(19)中华人民共和国国家知识产权局

(12)发明专利

(10)授权公告号 CN 107451225 B
(45)授权公告日 2021.02.05

(21)申请号 201710589656.5
(22)申请日 2012.12.21
(65)同一申请的已公布的文献号
申请公布号 CN 107451225 A
(43)申请公布日 2017.12.08
(30)优先权数据
61/580,193 2011.12.23 US
(62)分案原申请数据
201280068938.6 2012.12.21
(73)专利权人 亚马逊科技公司
地址 美国内华达
(72)发明人 N·宾科尔特 S·哈里佐保罗斯
M·A·沙赫 B·A·索维尔
D·茨罗吉安尼斯
(74)专利代理机构 中国贸促会专利商标事务所有限公司 11038
代理人 郑宗玉
(51)Int.Cl.
G06F 16/81(2019.01)
G06F 16/84(2019.01)
(56)对比文件
CN 102193970 A,2011.09.21
CN 101542475 A,2009.09.23
CN 1841380 A,2006.10.04
WO 2004021177 A1,2004.03.11
WO 2011005967 A1,2011.01.13
雷雪 等.异构分布式信息检索系统整合研究.《中国图书馆学报》.2008,第34卷(第174期),
审查员 刘蕾蕾

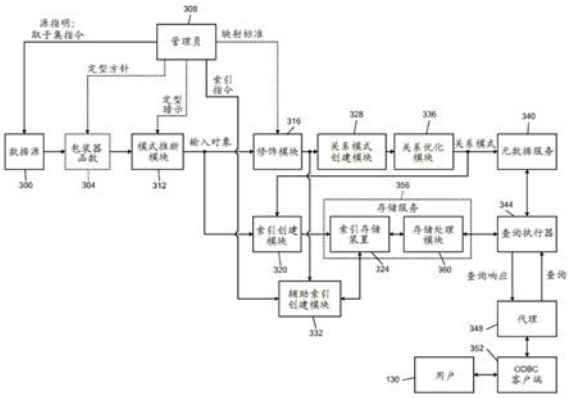
权利要求书5页 说明书42页 附图12页

(54)发明名称

用于半结构化数据的可缩放分析平台

(57)摘要

用于半结构化数据的可缩放分析平台。一种操作查询系统的方法包括从数据源检索对象,其中所述所检索到的对象中的每一个包括(i)数据和(ii)描述所述数据的元数据。所述方法包括通过以下方式动态地创建累积模式:从所述所检索到的对象中的每一个推断模式,并且将所述所推断出的模式与所述累积模式合并。所述方法包括将所述所检索到的对象中的每一个的所述数据存储在存储服务中。所述方法包括从用户接收查询,并且基于所述存储服务所存储的数据来对所述查询做出响应。



CN 107451225 B

1.一种由查询系统的处理器执行的方法,所述方法包括:

从数据源检索对象,其中所检索到的对象中的每一个包括数据和描述所述数据的元数据;

对于所检索到的对象中的每一个新对象,通过以下方式动态地创建累积模式:

基于对象的元数据和对象的元素的推断类型并且通过利用定型函数动态地对新对象进行定型,从新对象推断模式,其中,对新对象进行定型包括确定新对象的类型,以及定型函数是用于确定新对象的类型的函数;

通过将描述新对象的所推断出的模式与先前存在的累积模式合并来创建统一模式,所述先前存在的累积模式是根据先前所检索到的对象的所推断出的模式之间的先前合并操作而形成的;以及

存储统一模式作为累积模式;以及

将所检索到的对象中的每一个的数据存储在存储服务中。

2.如权利要求1所述的方法,还包括:

将所述累积模式转换为关系模式;

将所述关系模式呈现给用户;

从所述用户接收在所述关系模式上构造的查询;

基于由所述存储服务存储的数据对所述查询作出响应,其中来自所述用户的查询是在所述关系模式上构造的。

3.如权利要求2所述的方法,还包括:

将所检索到的对象中的第一对象的数据存储在第一索引和数组索引的至少一个中,其中所述存储服务包括第一索引和数组索引;以及

基于来自第一索引和数组索引中的至少一个的数据对所述查询做出响应。

4.如权利要求3所述的方法,还包括:将来自第一对象的数据作为键-值对存储在第一索引中,其中所述键-值对的值是所述数据,并且其中所述键-值对的键是基于与所述关系模式一致的数据的路径和第一对象的唯一识别符。

5.如权利要求4所述的方法,其中所述键-值对的键被构造为使得第一索引首先通过所述路径并且接着通过所述唯一识别符并置键-值对。

6.如权利要求3所述的方法,其中将作为数组的一部分的数据存储在所述数组索引中。

7.如权利要求3所述的方法,还包括:将第一索引存储在保序索引存储区中,其中所述存储服务包括所述保序索引存储区。

8.如权利要求2所述的方法,还包括:选择性地将所述累积模式的对象识别为映射。

9.如权利要求8所述的方法,其中基于对象的字段在所检索到的对象内的出现频率将所述累积模式的对象识别为映射,所述方法还包括以下中的至少一个:

在动态地创建所述累积模式的同时跟踪所述出现频率;

响应于所述出现频率的平均值低于阈值而将所述累积模式的对象识别为映射。

10.如权利要求8所述的方法,还包括:

将对应于所述映射的数据作为键-值对存储到映射索引中,其中所述键-值对的值是所述数据,并且其中所述键-值对的键是基于与所述关系模式一致的数据的路径、所检索到的对象中的提供所述数据的第一对象的唯一识别符、所述映射的联接键和所述映射中的数据

的映射键，其中所述键-值对的键被构造为使得所述映射索引首先通过所述路径、接下来通过所述唯一识别符、接下来通过所述联接键并且接着通过所述映射键来并置键-值对；或者

将对应于所述映射的数据作为键-值对存储到辅助映射索引中，其中所述键-值对的值是所述数据，并且其中所述键-值对的键是基于与所述关系模式一致的数据的路径、所述映射中的数据的映射键、所检索到的对象中的提供所述数据的第一对象的唯一识别符和所述映射的联接键，其中所述键-值对的键被构造为使得所述辅助映射索引首先通过所述路径、接下来通过所述映射键、接下来通过所述唯一识别符并且接着通过所述联接键来并置键-值对。

11.如权利要求2所述的方法，其中将所述累积模式转换为所述关系模式包括创建根表格，其中用于每个元素的列在所述累积模式的顶端中。

12.如权利要求11所述的方法，其中将所述累积模式转换为所述关系模式还包括针对所述累积模式中的每个数组在所述关系模式中创建额外表格，并且其中所述额外表格包括联接键列、索引列和针对所述数组中的每个标量类型的数据的值列，所述方法还包括：当所述数组存在于所述累积模式的所述顶端处时，将联接键列插入到所述额外表格中，并且将联接键列插入到所述根表格中；以及当所述数组嵌套在所述累积模式中位于所述顶端下方时，将联接键列插入到所述额外表格中，并且将联接键列插入到中间表格中。

13.如权利要求11所述的方法，其中将所述累积模式转换为所述关系模式还包括针对被确定为存在于所述累积模式中的每个映射在所述关系模式中创建额外表格，其中所述额外表格包括联接键列、键列和对于所述映射中的每个标量类型的数据的值列，所述方法还包括：当所述映射存在于所述累积模式的顶端处时，将联接键列插入到所述额外表格中，并且将联接键列插入到所述根表格中；以及当所述映射嵌套在所述累积模式中位于所述顶端下方时，将联接键列插入到所述额外表格中，并且将联接键列插入到中间表格中。

14.如权利要求2-13中任一项所述的方法，其中所述关系模式是结构化查询语言SQL模式并且所述查询是SQL查询，或者其中所述查询是Hive-QL查询、jagl查询和XQuery之一。

15.一种存储处理器可执行的指令的非暂时性计算机可读介质，其中所述指令在一个或多个处理器上执行时使得所述一个或多个处理器：

从数据源检索对象，其中所检索到的对象中的每一个包括数据和描述所述数据的元数据；

动态地创建累积模式，其中，为了动态地创建所述累积模式，所述指令使得所述一个或多个处理器对于所检索到的对象中的每一个新对象：

基于对象的元数据和对象的元素的推断类型并且通过利用定型函数动态地对新对象进行定型，从新对象推断模式，其中，对新对象进行定型包括确定新对象的类型，以及定型函数是用于确定新对象的类型的函数；

通过将描述新对象的所推断出的模式与先前存在的累积模式合并来创建统一模式，所述先前存在的累积模式是根据先前所检索到的对象的所推断出的模式之间的先前合并操作而形成的；以及

存储统一模式作为累积模式；以及

将所检索到的对象中的每一个的数据存储在存储服务中。

16.一种由模式推断系统的处理器执行的方法，包括：

监视数据源以检测对所述数据源的一个或多个更新；

响应于对所述数据源的所述一个或多个更新的检测：

更新第一数据库模式，其中，更新包括将通过利用定型函数动态地对所述数据源的所述一个或多个更新进行定型而从对所述数据源的所述一个或多个更新推断出的模式与第一数据库模式合并，其中，第一数据库模式是至少部分地基于从对第一数据库的先前更新推断出的一个或多个模式的，以及其中，对所述数据源的所述一个或多个更新进行定型包括确定所述数据源的所述一个或多个更新的类型，以及定型函数是用于确定所述数据源的所述一个或多个更新的类型的函数；和

存储一个或多个第一数据库模式更新。

17.如权利要求16所述的方法，还包括：

进一步响应于对所述数据源的所述一个或多个更新的检测：

从所述数据源获得更新；和

将更新存储到存储服务。

18.如权利要求17所述的方法，还包括：

接收对所述存储服务的查询；和

访问更新后的第一数据库模式，以在所述存储服务处执行所述查询。

19.根据权利要求16所述的方法，其中，对所述数据源的更新将一个或多个新的数据对象存储在所述数据源处。

20.根据权利要求16所述的方法，其中，对所述数据源的更新修改所述数据源处的一个或多个现有的数据对象。

21.如权利要求16所述的方法，还包括：

监视所述数据源以检测对所述数据源的其他更新；

响应于对所述数据源的其他更新的检测：

确定对于其他更新的推断出的模式不更新第一数据库模式；

从所述数据源获得其他更新；和

将其他更新存储到存储服务。

22.如权利要求16所述的方法，还包括：

接收对要监视的数据源的指定；和

响应于接收到所述指定来执行对数据源的监视。

23.一种模式推断系统，包括：

存储器，用于存储程序指令，所述程序指令如果被至少一个处理器执行则使所述至少一个处理器执行方法以至少：

监视数据源以检测对所述数据源的一个或多个更新；

响应于对所述数据源的所述一个或多个更新的检测：

更新第一数据库模式，其中，更新包括将通过利用定型函数动态地对所述数据源的所述一个或多个更新进行定型而从对所述数据源的所述一个或多个更新推断出的模式与第一数据库模式合并，其中，第一数据库模式是至少部分地基于从对第一数据库的先前更新推断出的一个或多个模式的，以及其中，对所述数据源的所述一个或多个更新进行定型包括确定所述数据源的所述一个或多个更新的类型，以及定型函数是用于确定所述数据源的

所述一个或多个更新的类型的函数;和

存储一个或多个第一数据库模式更新。

24.根据权利要求23所述的模式推断系统,其中,所述程序指令进一步使所述至少一个处理器执行所述方法以至少:

进一步响应于对所述数据源的所述一个或多个更新的检测:

从所述数据源获得更新;和

将更新存储到存储服务。

25.根据权利要求24所述的模式推断系统,其中,所述程序指令进一步使所述至少一个处理器执行所述方法以至少:

接收对所述存储服务的查询;和

访问更新后的第一数据库模式,以在所述存储服务处执行所述查询。

26.根据权利要求23所述的模式推断系统,其中,对所述数据源的更新将一个或多个新的数据对象存储在所述数据源处。

27.根据权利要求23所述的模式推断系统,其中,对所述数据源的更新修改所述数据源处的一个或多个现有的数据对象。

28.根据权利要求23所述的模式推断系统,其中,所述程序指令进一步使所述至少一个处理器执行所述方法以至少:

监视所述数据源以检测对所述数据源的其他更新;

响应于对所述数据源的其他更新的检测:

确定对于其他更新的推断出的模式不更新第一数据库模式;

从所述数据源获得其他更新;和

将其他更新存储到存储服务。

29.根据权利要求23所述的模式推断系统,其中,所述程序指令进一步使所述至少一个处理器执行所述方法以至少:

接收对要监视的数据源的指定;和

响应于接收到所述指定来执行对数据源的监视。

30.一种存储处理器可执行指令的非暂态计算机可读介质,其中所述处理器可执行指令在一个或多个处理器上执行时使所述一个或多个处理器:

监视数据源以检测对所述数据源的一个或多个更新;

响应于对所述数据源的所述一个或多个更新的检测:

更新第一数据库模式,其中,更新包括将通过利用定型函数动态地对所述数据源的所述一个或多个更新进行定型而从对所述数据源的所述一个或多个更新推断出的模式与第一数据库模式合并,其中,第一数据库模式是至少部分地基于从对第一数据库的先前更新推断出的一个或多个模式的,以及其中,对所述数据源的所述一个或多个更新进行定型包括确定所述数据源的所述一个或多个更新的类型,以及定型函数是用于确定所述数据源的所述一个或多个更新的类型的函数;和

存储一个或多个第一数据库模式更新。

31.如权利要求30所述的非暂态计算机可读介质,其中,所述处理器可执行指令使所述一个或多个处理器进一步实施:

进一步响应于对所述数据源的所述一个或多个更新的检测：

从所述数据源获得更新；和

将更新存储到存储服务。

32.如权利要求31所述的非暂态计算机可读介质，其中，所述处理器可执行指令使所述一个或多个处理器进一步实施：

接收对所述存储服务的查询；和

访问更新后的第一数据库模式，以在所述存储服务处执行所述查询。

33.如权利要求30所述的非暂态计算机可读介质，其中，对所述数据源的更新将一个或多个新的数据对象存储在所述数据源处。

34.如权利要求30所述的非暂态计算机可读介质，其中，对所述数据源的更新修改所述数据源处的一个或多个现有的数据对象。

35.如权利要求30所述的非暂态计算机可读介质，其中，所述处理器可执行指令使所述一个或多个处理器进一步实施：

监视所述数据源以检测对所述数据源的其他更新；

响应于对所述数据源的其他更新的检测：

确定对于其他更新的推断出的模式不更新第一数据库模式；

从所述数据源获得其他更新；和

将其他更新存储到存储服务。

用于半结构化数据的可缩放分析平台

[0001] 本申请是申请号为201280068938.6、发明名称为“用于半结构化数据的可缩放分析平台”、国际申请日为2012年12月21日的专利申请的分案申请，其全部内容通过引用合并于此。

[0002] 相关申请的交叉参考

[0003] 本申请主张2012年12月21日提交的美国实用申请第13/725,399号和2011年12月23日提交的美国临时申请第61/580,193号的优先权。上述申请的全部公开内容以引用的方式并入本文中。

技术领域

[0004] 本公开涉及一种可缩放交互式数据库平台，并且更具体地说，涉及一种并入有存储和计算的用于半结构化数据的可缩放交互式数据库平台。

背景技术

[0005] 本文中所提供的背景描述是出于概括地呈现本公开的上下文的目的。目前所提及的发明人的作品(就本背景部分中所描述的方面来说)以及在提交时可能尚未取得现有技术资格的背景描述的若干方面既不明确地也不隐含地被认为是本公开的现有技术。

[0006] 传统的数据库系统以与基础存储后端紧密集成的查询执行引擎为特点，所述基础存储后端通常由不具有计算能力的可成块寻址的持久性存储装置组成。这些装置(硬盘驱动器和/或固态驱动器)的特征在于(a)依据是顺序地还是随机地存取数据而显著不同的存取时间、(b)以块粒度设定的具有固定最小大小的存取单元以及(c)显著比主存储器慢(几个数量级)的存取时间。从存储管理到查询执行到查询优化，这些特征连同存储后端不具有任何非平凡计算能力的假设已对数据库系统的设计具有重要影响。

[0007] 数据库最初充当管理商家日常活动的操作存储区。随着数据库技术在性能和成本两方面有所改进，商家认为需要保持越来越多的操作历史和商业状态以供稍后分析。此类分析帮助商家洞察其过程并对它们进行优化，进而提供竞争优势和越来越多的利润。

[0008] 数据仓库由于这种需要而产生。商业数据通常被很好地结构化，从而容易填入关系表格中。数据仓库实质上是供应结构化查询语言(SQL)来对这种商业数据进行离线分析的可缩放关系数据库系统，并且针对主读工作负荷进行优化。举例来说，数据仓库包括如Teradata等传统系统以及例如Vertica、Greenplum和Aster Data等较新供应商。他们提供SQL接口、索引和快速列式访问。

[0009] 通常，周期性地(例如，每夜或每周)向数据仓库加载从各种源和操作系统摄取的数据。对这种数据进行清理、策划并统一成单个模式并且将其加载到仓库中的过程被称为提取-转换-加载(ETL)。随着源和数据的种类增加，ETL过程的复杂性也增加。成功地实施ETL(包括定义恰当的模式并且将输入数据匹配于预定模式)可能需要专业人员花费数周到数月，并且可能很难或不可能实施改变。市场上有很多工具(例如Abinitio、Informatica和Pentaho)来辅助ETL过程。然而，ETL过程大体上仍是麻烦的、脆弱的并且昂贵的。

[0010] 数据分析市场已经爆发出使得商业用户易于对仓库中的数据执行特别迭代分析的许多商业智能和可视化工具。商业智能工具构建仓库数据的多维集合并且允许用户导航通过并观看这种数据的各种片段和投影。举例来说，商业用户可能想要通过产品种类、地区和商店查看总月度销售。然后，他们可能想要针对特定种类深挖到每周销售或者上升到查看整个国家的销售。多维集合还可称为在线分析处理(OLAP)立方体。例如Business Objects和Cognos等许多商业智能(BI)工具实现此类分析，并且支持用于查询立方体的称为多维表达式(MDX)的语言。还有例如MicroStrategy、Tableau和Spotfire等许多可视化工具，其允许商业用户直观地浏览这些立方体和数据仓库。

[0011] 最近，商家想要分析的数据类型已经改变。随着传统实体商业放到网上并且形成新的在线商业，这些商家需要分析例如Google和Yahoo等领先的公司所充斥着的数据类型。这些数据类型包括例如网页、页面浏览量日志、点击流、RSS(丰富站点摘要)馈入、应用程序日志、应用服务器日志、系统日志、事务日志、传感器数据、社交网络馈入、新闻馈入和博客贴子等数据类型。

[0012] 这些半结构化数据不能很好地适合传统仓库。它们具有某种固有结构，但结构可能是不一致的。结构可随着时间迅速地改变，并且可随不同源而变化。它们并不是自然列成表格的，并且用户想要对这些数据执行的分析——聚合、分类、预测等——不容易用SQL来表达。用于有效利用这些数据的现有工具是麻烦并且不足的。

[0013] 因而，出现了新的高度可伸缩的存储和分析平台——Hadoop，其由在Google处实施以用于管理网络爬行和搜索的技术激发。就其核心来说，Hadoop供应用于可靠地存储其数据的群集式文件系统——HDFS(Hadoop分布式文件系统)以及基本的并行分析引擎——MapReduce来支持较复杂的分析。从这些部分开始，Hadoop生态系统已经发展成包括带索引的操作存储区——HBase和新的查询接口——Pig和Hive，其依赖于MapReduce。

[0014] Hive是在Hadoop之上添加查询层的Apache项目，其没有在传统仓库中找到的用于查询优化、高速缓存和索引的任何优化。代替地，Hive简单地将类SQL语言(称为Hive-QL)的查询转换为待对Hadoop群集运行的MapReduce工作。传统的商业用户对于Hive有三个主要问题。Hive不支持标准的SQL，并且不具有动态模式。另外，Hive不够快速来允许交互式查询，因为每个Hive查询需要重新剖析所有源数据的MapReduce工作，并且通常需要多次遍历源数据。

[0015] Impala是Cloudera的Hadoop实施方案的用于Hive-QL查询的实时引擎。其对Hive的顺序文件提供分析，并且可最终支持嵌套模型。然而，其不具有动态模式，而是需要用户仍提前针对待查询的数据提供模式。

[0016] Pig是另一个Apache项目并且提供用于在Hadoop中处理日志文件的无模式脚本语言。如同Hive，Pig将每件事转换成映射-化简工作。同样，其不利用任何索引，并且不够快速来进行交互。

[0017] Jaql是用于分析JavaScript对象表示法(JSON)日志的无模式声明性语言(与如SQL等声明性语言相反)。如同Pig，其编译成Hadoop上的映射-化简程序，并且共有许多相同缺点，包括非交互性速度。

[0018] Hadoop本身正在相当迅速地流行起来，并且在云中现成可用。Amazon提供弹性的映射-化简，其可实际上等效于在云中运行的Hadoop的MapReduce实施方案。其对存储于

Amazon的基于云的S3(简单存储服务)中的数据起作用,并且向S3输出结果。

[0019] Hadoop生态系统的优点有三项。首先,系统缩放到极限大小并且能够存储任何数据类型。第二,与传统仓库相比,其具有极低成本(便宜多达20倍)。第三,其是开放源,这避免了与单个供应商锁定。用户想要针对恰当工作挑选恰当工具的能力,并且避免在系统之间移动数据来完成其工作。虽然Hadoop较灵活,但使用Hadoop需要具有深厚知识的专门技术管理员和程序员,通常很难找到这些人员。此外,Hadoop太慢而不具交互性。即使最简单的查询也需要数分钟到数小时来执行。

[0020] Dremmel是在Google处内部开发的工具,其对嵌套关系或半结构化数据提供基于SQL的分析查询。原始版本处理呈ProtoBuf格式的数据。Dremmel需要用户提前针对所有记录定义模式。BigQuery是Dremmel的基于云的商业化,并且扩展到处理CSV和JSON格式。Drill是Dremmel的开放源版本。

[0021] Asterix是用于使用抽象数据模型(ADM)和注释查询语言(AQL)管理和分析半结构化数据的系统,ADM是JSON的普遍化。Asterix不支持标准SQL,也没有本公开所给予的快速访问。

发明内容

[0022] 一种操作查询系统的方法包括:从数据源检索对象,其中所述所检索到的对象中的每一个包括(i)数据和(ii)描述所述数据的元数据。所述方法包括通过以下方式动态地创建累积模式:从所述所检索到的对象中的每一个推断模式,并且将所述所推断出的模式与所述累积模式合并。所述方法包括将所述所检索到的对象中的每一个的所述数据存储在存储服务中。所述方法包括从用户接收查询,并且基于所述存储服务所存储的数据来对所述查询做出响应。

[0023] 所述方法还包括将所述累积模式转换为关系模式,并且将所述关系模式呈现给所述用户,其中来自所述用户的所述查询是在所述关系模式上构造的。所述方法还包括将所述所检索到的对象中的每一个的所述数据存储在(i)第一索引和(ii)数组索引的至少一个中,其中所述存储服务包括所述第一索引和所述数组索引。所述方法还包括基于来自所述第一索引和所述数组索引中的至少一个的数据来对所述查询做出响应。

[0024] 所述方法还包括将来自所检索到的对象的数据作为键-值对存储在所述第一索引中,其中所述键-值对的值是所述数据,并且其中所述键-值对的键是基于(i)与所述关系模式一致的数据的路径和(ii)所述所检索到的对象的唯一识别符。所述键-值对的键被构造为使得所述第一索引首先通过所述路径并且接着通过所述唯一识别符并置键-值对。将作为数组的一部分的数据存储在所述数组索引中。不将作为数组的一部分的数据存储在所述第一索引中。

[0025] 将所述数据作为键-值对存储在所述数组索引中,其中所述键-值对的值是所述数据,并且其中所述键-值对的键是基于(i)与所述关系模式一致的数据的路径、(ii)所述所检索到的对象的唯一识别符和(iii)所述数据在所述数组中的位置的索引。所述键-值对的键被构造为使得所述数组索引首先通过所述路径、接下来通过所述唯一识别符并且接着通过所述索引来并置键-值对。所述键-值对的键进一步基于联接键。所述键-值对的键被构造为使得所述数组索引首先通过所述路径、接下来通过所述唯一识别符、接下来通过所述联

接键并且接着通过所述索引来并置键-值对。所述方法还包括选择性地将所述数据存储在辅助数组索引中。

[0026] 将所述数据作为键-值对存储在所述辅助数组索引中，其中所述键-值对的值是所述数据，并且其中所述键-值对的键是基于(i)与所述关系模式一致的数据的路径、(ii)所述数据在所述数组中的位置的索引和(iii)所述对象的唯一识别符。所述键-值对的键被构造为使得所述辅助数组索引首先通过所述路径、接下来通过所述索引并且接着通过所述唯一识别符来并置键-值对。所述键-值对的键进一步基于联接键。所述键-值对的键被构造为使得所述辅助数组索引首先通过所述路径、接下来通过所述索引、接下来通过所述唯一识别符并且接着通过所述联接键来并置键-值对。

[0027] 所述方法还包括将所述第一索引存储在保序索引存储区中，其中所述存储服务包括所述保序索引存储区。所述方法还包括将所述数组索引存储在所述保序索引存储区中。所述关系模式是结构化查询语言(SQL)模式，并且所述查询是SQL查询。所述查询是Hive-QL查询、jaql查询和XQuery中的一个。

[0028] 所述方法还包括选择性地将所述累积模式的对象识别为映射。基于所述对象的字段在所述所检索到的对象内的出现频率来将所述累积模式的对象识别为映射。所述方法还包括在动态地创建所述累积模式的同时跟踪所述出现频率。响应于所述出现频率的平均值低于阈值而将所述累积模式的所述对象识别为映射。

[0029] 所述方法还包括将对应于所述映射的数据作为键-值对存储到映射索引中，其中所述键-值对的值是所述数据，并且其中所述键-值对的键是基于(i)与所述关系模式一致的数据的路径、(ii)提供所述数据的所述所检索到的对象的唯一识别符、(iii)所述映射的联接键和(iv)所述映射中的所述数据的映射键。所述键-值对的键被构造为使得所述映射索引首先通过所述路径、接下来通过所述唯一识别符、接下来通过所述联接键并且接着通过所述映射键来并置键-值对。

[0030] 所述方法还包括将对应于所述映射的数据作为键-值对存储到辅助映射索引中，其中所述键-值对的值是所述数据，并且其中所述键-值对的键是基于(i)与所述关系模式一致的数据的路径、(ii)所述映射中的所述数据的映射键、(iii)提供所述数据的所述所检索到的对象的唯一识别符和(iv)所述映射的联接键。所述键-值对的键被构造为使得所述辅助映射索引首先通过所述路径、接下来通过所述映射键、接下来通过所述唯一识别符并且接着通过所述联接键来并置键-值对。

[0031] 将所述累积模式转换为所述关系模式包括创建根表格，其中用于每个元素的列在所述累积模式的顶端中。将所述累积模式转换为所述关系模式包括针对所述累积模式中的每个数组在所述关系模式中创建额外表格。所述额外表格包括(i)联接键列、(ii)索引列和(iii)针对所述数组中的每个标量类型的数据，值列。

[0032] 所述方法还包括当所述数组存在于所述累积模式的所述顶端处时将联接键列插入到所述额外表格中并且插入到所述根表格中。所述方法还包括当所述数组嵌套在所述累积模式中位于所述顶端下方时将联接键列插入到所述额外表格中并且插入到中间表格中。将所述累积模式转换为所述关系模式包括针对所述累积模式中的每个映射在所述关系模式中创建额外表格。

[0033] 所述额外表格包括(i)联接键列、(ii)键列和(iii)对于所述映射中的每个标量类

型的数据，值列。所述键列是字符串类型。所述方法还包括当所述映射存在于所述累积模式的顶端处时将联接键列插入到所述额外表格中并且插入到所述根表格中。

[0034] 所述方法还包括当所述映射嵌套在所述累积模式中位于所述顶端下方时将联接键列插入到所述额外表格中并且插入到中间表格中。所述方法还包括选择性地将所检索到的对象的数据值作为键-值对存储在值索引中，其中所述键-值对的键是基于(i)与所述关系模式一致的数据值的路径和(ii)所述数据值，其中所述键-值对的值是基于所述所检索到的对象的唯一识别符，并且其中所述存储服务包括所述值索引。

[0035] 所述键-值对的键被构造为使得所述值索引首先通过所述路径并且接着通过所述数据值来并置键-值对。当所述数据值是数组的一部分时，所述键-值对的值进一步基于所述数组中的所述数据值的索引。所述键-值对的值进一步基于所述数组的联接键。当所述数据值是映射的一部分时，所述键-值对的值进一步基于所述映射中的所述数据值的映射键。

[0036] 所述键-值对的值进一步基于所述映射的联接键。所述方法还包括通过将元数据添加到从所述数据源获得的原始数据来产生所述所检索到的对象。推断用于所检索到的对象的模式是基于所述所检索到的对象的所述元数据和所述所检索到的对象的元素的推断类型来执行的。对于所述所检索到的对象中的每一个，所述所检索到的对象的所述数据包括值，并且所述所检索到的对象的所述元数据包括所述值的名称。

[0037] 所述所检索到的对象中的每一个是JavaScript对象表示法(JSON)对象。所述累积模式是JavaScript对象表示法(JSON)模式。所述方法还包括选择性地将所述所检索到的对象中的每一个存储在行索引中，其中所述存储服务包括所述行索引。将所检索到的对象作为键-值对存储在所述行索引中，其中所述键-值对的键是所述所检索到的对象的唯一识别符，并且其中所述键-值对的值是所述整个所检索到的对象的串行化。

[0038] 本公开的进一步适用性区域将从下文提供的详细描述中变得显而易见。应当理解，详细描述和特定实施例意图仅用于说明目的并且并不意图限制本公开的范围。

附图说明

[0039] 将从详细描述和附图中更全面地理解本公开，其中：

[0040] 图1A描绘利用云资源的用于半结构化数据的可缩放分析平台的示例性网络架构；

[0041] 图1B描绘在用户端处具有服务器设备的用于半结构化数据的可缩放分析平台的示例性网络架构；

[0042] 图1C是服务器系统的功能框图；

[0043] 图2A是用于半结构化数据的示例性可缩放分析平台的功能框图；

[0044] 图2B是用于半结构化数据的可缩放分析平台的示例性查询系统的功能框图；

[0045] 图3是描绘并入所摄取的数据的示例性方法的流程图；

[0046] 图4是描绘推断模式的示例性方法的流程图；

[0047] 图5是描绘合并两个模式的示例性方法的流程图；

[0048] 图6是描绘折叠模式的示例性方法的流程图；

[0049] 图7是描绘用数据填充索引的示例性方法的流程图；

[0050] 图8是描绘执行映射修饰的示例性方法的流程图；并且

[0051] 图9是描绘根据JSON模式创建关系模式的示例性方法的流程图。

[0052] 在附图中,可重复使用参考数字来识别相似和/或相同的元件。

具体实施方式

[0053] 概述

[0054] 本公开描述一种分析平台,其能够提供顺应SQL(结构化查询语言)的接口来用于查询半结构化数据。仅出于说明目的,以JSON(JavaScript对象表示法)格式表示半结构化数据。根据本公开的原理,可使用其它自描述性半结构化格式。源数据不需要是自描述性的。描述可与数据分离,正如像协议缓冲等某物的情况一样。只要存在用以将标签施加到数据的规则、试探法或包装器函数,就可将任何输入数据转换为类似于JSON格式的对象。

[0055] 在根据本公开的分析平台的各种实施方案中,实现以下一些或全部优点:

[0056] 速度

[0057] 分析平台提供快速查询响应时间来支持特别、探索性和交互性分析。用户可使用这个系统来迅速地在数据中发现隐藏见解,而不必提交查询并且稍后在当天或第二天返回观看结果。分析平台依赖于索引存储区,将所有所摄取的数据存储在索引中,这实现了快速响应时间。

[0058] 使用两个主要索引,即BigIndex(BI)和ArrayIndex(AI),下文中更详细地对其进行描述。这些索引是路径索引与面向列的存储区之间的交叉。如同面向列的存储区,它们允许查询仅在相关字段中检索数据,进而减少I/O(输入/输出)需求并且改善性能。然而,不同于列存储区,这些索引适合于具有许多字段的复杂嵌套对象和集合。对于其它访问型式,分析平台引擎维持辅助索引,下文中更详细地对其进行描述,包括ValueIndex(VI)。如同传统的数据库索引,ValueIndex针对特定字段值或值范围提供快速对数访问。这些索引显著减少了检索以满足查询所必要的数据,进而改善响应时间。

[0059] 动态模式

[0060] 分析平台自身从数据中推断模式,使得用户不必先验地知道预期模式并且在可加载数据之前预先声明所述模式。半结构化数据可具有随时间和随不同源变化的结构。因而,在数据到达时,引擎动态地从数据计算并更新模式(或结构)。向用户呈现基于这种计算得到的模式的关系模式,用户可使用所述关系模式来撰写查询。

[0061] 不同于需要程序员在查询之前指定数据集合的模式的先前分析引擎,本平台在所有所摄取的对象当中计算(或推断)基础模式。由于动态模式性质,存在大量灵活性。生成源数据的应用程序可随着所述应用程序演进而改变结构。分析者可聚合并查询来自各个时期的数据而不需要指定模式在各个时期之间如何变化。此外,不需要设计并强制执行全局模式,设计并强制执行全局模式可能花费数月,并且常常需要排除不适合所述模式的数据。

[0062] 有时被描述为“无模式”的如MapReduce或Pig等其它分析系统具有两个主要缺点。首先,它们需要用户知道模式以便查询数据,而非自动向用户呈现所推断出的模式。第二,它们对每次查询剖析并解译对象及其结构,而分析平台在加载时对对象进行剖析和编索引。这些索引允许快速得多地运行后续查询,如上文所提及。先前引擎不提供从底层数据自动推断精确且简洁的模式。

[0063] SQL

[0064] 分析平台暴露标准SQL查询接口(例如,符合ANSI SQL 2003的接口),使得用户可

利用现有SQL工具(例如,报告、可视化和BI工具)以及专门知识。因而,熟悉SQL或SQL工具的商业用户可直接访问和查询半结构化数据而不需要加载数据仓库。由于传统的基于SQL的工具不处理JSON或其它半结构化数据格式,所以分析平台呈现JSON对象的所计算得到的模式的关系视图。其呈现正规化视图并且并入优化以保持视图的大小易于管理。虽然关系视图可在所述模式中呈现若干张表格,但这些表格未必被物化。

[0065] 为了更好地适应以表格形式表示半结构化数据,分析平台可自动识别"映射"对象。映射是可在其中搜索并查询字段名称和值的对象(或嵌套对象)。举例来说,对象可含有作为字段名称的日期以及如针对值的页面浏览量的统计信息。在关系视图中,将映射提取到单独表格中,并枢转数据以使得键在键列中并且值在值列中。

[0066] 缩放和弹性

[0067] 分析平台进行缩放以处理大数据集大小。分析平台可自动且动态地在独立节点上分布内部数据结构和处理。

[0068] 分析平台是针对虚拟化"云"环境来设计和构建的,所述"云"环境包括例如Amazon网络服务等公用云以及例如由用户的组织管理或由第三方(例如Rackspace)提供的虚拟化服务器环境等专用云。可利用Amazon网络服务的各个部件,包括S3(简单存储服务)、EC2(弹性计算云)和弹性块存储(EBS)。分析平台是弹性的,这意味着其可按照需要放大缩小到任意大小,并且可通过将其内部数据结构存储在长期存储区(例如Amazon S3)上来休眠。分析平台还具有多租户和多用户支持。

[0069] 分析平台使用基于服务的架构,其具有四个部件:代理、元数据服务、查询执行器和存储服务。为了缩放分析平台引擎来支持较大数据集、提供较快速的响应以及支持较多用户,对执行引擎进行并行化并且在独立的低成本服务器节点上分割存储服务。这些节点在托管环境中可以是真实服务器或虚拟化服务器。由于执行器与存储服务分离,所以它们可独立缩放。这种分离的向外扩展架构允许用户利用如AWS的云环境提供的用于存储和计算的按需弹性。

[0070] 存储服务可配置有各种分割策略。此外,可将底层数据结构(索引和元数据)迁移到长期存储装置,如Amazon S3,以使系统在不使用时休眠,进而降低成本。

[0071] 同步化

[0072] 分析平台可被配置来自动地使其内容与来自如HDFS(Hadoop分布式文件系统)、Amazon S3(简单存储服务)和noSQL存储区(例如MongoDB)等储存库的源数据同步并且进而复制所述源数据。可连续监视这些源以查看变化、添加和更新,使得分析平台可摄取已改变的数据。这允许查询结果是相对最新的。

[0073] 模式推断

[0074] 分析平台响应于源中出现的数据而采取以下动作:(1)从所述数据推断统一半结构化(例如JSON)模式,(2)创建所述模式的关系视图,(3)用数据填充物理索引,以及(4)执行利用所述索引的查询。可对动作1、2和3的一部分或全部进行管线化以允许仅单遍通过来自数据源的数据。

[0075] 首先描述第一个动作——模式推断。

[0076] 对半结构化数据的介绍

[0077] JSON是越来越流行的自描述性半结构化数据格式,并且非常普遍用于互联网上的

数据交换。再次，尽管本文中描述JSON是出于说明目的并且为了提供使用JSON格式的稍后实施例的上下文，但本公开不限于JSON。

[0078] 简要地说，JSON对象由字符串字段(或列)和潜在不同类型的对应值构成：数字、字符串、数组、对象等。JSON对象可为嵌套的，并且字段可为多值的，例如数组、嵌套数组等。可在http://JSON.org处找到说明。可在“A JSON Media Type for Describing the Structure and Meaning of JSON Documents”(IETF(Internet Engineering Task Force)draft-zyp-json-schema-03,2010年11月22日，可在http://tools.ietf.org/html/draft-zyp-json-schema-03处获得)中找到额外细节，所述文献的全部公开内容以引用的方式并入本文中。JSON的一般化包括更多类型，例如BSON(二进制JSON)。此外，如XML、Protobuf、Thrift等其它半结构化格式都能转换为JSON。当使用XML时，查询可遵照XQuery而非SQL。

[0079] 以下是示例性JSON对象：

[0080]
```
{ "player": { "fname": "George", "lname": "Ruth", "nickname" :
   "Babe"}, "born": "February 6, 1985",
   "avg": 0.342, "HR": 714,
   "teams": [ { "name": "Boston Red Sox", "years": "1914-1919" },
   { "name": "New York Yankees", "years": "1920-1934" },
   { "name": "Boston Braves", "years": "1935" } ] }
```

[0081] 半结构化对象的结构可在对象之间有所变化。因而，在同一棒球数据中，可找到以下对象：

[0082]
```
{ "player": { "fname": "Sandy", "lname":  "Koufax"}, "born":
   "December 30, 1935",
    "ERA": 2.76, "strikeouts": 2396,
   "teams": [ { "name": "Brooklyn / LA Dodgers", "years": "1955-
   1966" } ] }
```

[0083] 模式描述在数据集合中找到的可能结构和数据类型。这个模式包括字段的名称、对应值的类型和嵌套关系。因此，以上两个对象的模式将为：

[0084]
```
{ "player": { "fname": string, "lname": string, "nickname": string
   }, "born": string, “avg": number, "HR": number, "ERA": number,
   "strikeouts": number,
   "teams": [ { "name": string, "years": string } ] }
```

[0085] 虽然以上是本文献全文中用来说明模式的表示法，但更完整的说明是可在http://JSON-schema.org处获得的JSON模式。举例来说，JSON模式中的类型通常用引号括起来，如呈字符串或“整数(int)”形式。出于本公开的简明和易读性，将省略引号。

[0086] 半结构化对象可替代地视为树，其中字段作为节点并且叶子作为原子值。对象或模式中的路径是这棵树中的路径，例如"player.fname"、"teams[].name"。

[0087] 迭代模式推断

[0088] 在用户可问数据集问题之前，其需要知道模式——即，什么字段或维度可用于查询。在许多情况下，分析者不负责生成数据，所以其不知道什么内容已被记录并且可用。举例来说，在以上棒球实施例中，如果已经仅在集合中观测到击球员，那么分析者可能不知道“ERA”字段是可用的。因而，分析平台从所摄取的数据计算(或推断)统一模式，并且呈现所述模式的关系视图来辅助分析者制定查询。

[0089] 分析平台旨在产生针对优化模式的精度和简洁性的模式。一般来说，精确意味着模式表示所观测或摄取的数据中的所有结构并且不允许尚未看见的结构。简洁意味着模式足够小以使得可由人类阅读和理解。

[0090] 动态创建模式的一般方法是以从过去对象推断出的“当前”模式开始并且随着摄取新对象而扩大模式。简单地将当前模式(S_curr)与新对象(O_new)的模式(类型)合并来得到新模式(S_new)：

[0091] S_new＝merge(S_curr,type(O_new))

[0092] 概略地说，合并过程采用两个模式的联合，折叠共用字段、子对象和数组，并且在出现新字段、子对象和数组时进行添加。

[0093] 对象

[0094] 以下一些实施例使用类似来自推特(Twitter)的数据流(称为消防带)的输出的数据。推特消防带给出表示“已推送”推文的JSON对象以及关于那些推文的元数据(例如，用户、位置、主题等)的流(无休止序列)。这些推文类似于许多其它类型的事件日志数据，例如现代网络框架(例如，Ruby on Rails)、移动应用程序、传感器和装置(能量计、恒温器)等所产生的数据。虽然类似于推特数据，但以下实施例出于解释目的而偏离实际推特数据。

[0095] 简单地处理基本JSON对象；仅推断对象中所看到的类型。举例来说，考虑以下对象：

[0096]
```
{ "created_at": "Thu Nov 08", "id": 266353834,
 "source": "Twitter for iPhone",
 "text": "@ilstavrachi: would love dinner. Cook this:
   http://bit.ly/955Ffo",
 "user": { "id": 29471497, "screen_name": "Mashah08" },
 "favorited": false}
```

[0097] 从那个对象推断出的模式将为：

[0098]
```
{ "created_at": string, "id": number, "source": string, "text":
   string,

 "user": { "id": number, "screen_name": string }, "favorited":
  boolean }
```

[0099] 随着新对象到来，可通过对字段集执行联合来添加新字段。有时，将重复字段，但其类型改变，称为类型多态性的情形。模式使用具有相同键的多个属性来表示类型多态性。

[0100] 日志格式经常变化，并且开发者可添加新字段或改变字段类型。作为具体实施例，考虑标识推文的"id"字段，其最初是数字。然而，随着推文数目增长，某些编程语言无法处理这类大数字，所以已经将"id"字段改变为字符串。因而，假设看到以下形式的新记录

[0101]
```
{ "created_at": "Thu Nov 10", "id": "266353840",
 "source": "Twitter for iPhone",
 "text": "@binkert: come with me to @ilstavrachi place",
 "user": { "id": 29471497, "screen_name": "Mashah08" },
 "retweet_count": 0 }
```

[0102] 由于现在已经看到字符串"id"，并且新字段"retweet_count"已经出现，所以如下

扩大模式：

[0103]
```
{ "created_at": string, "id": number, "id": string, "source":
   string, "text": string,
   "user": { "id": number, "screen_name": string },
   "retweet_count": number }
```

[0104] 请注意，"id"出现两次，一次作为字符串并且一次作为数字。有时，嵌套对象的结构变化。举例来说，假设添加用户的更多简档信息：

[0105]
```
{ "created_at": "Thu Nov 10", "id": "266353875",
   "source": "Twitter for iPhone",
   "text": "@binkert: come with me to @ilstavrachi place",
   "user": { "id": "29471755", "screen_name": "mashah08",
   "location": "Saratoga, CA", "followers_count": 22 },
   "retweet_count": 0 }
```

[0106] 在这种情况下，平台递归地合并"user"嵌套模式以得到以下模式：

[0107]
```
{ "created_at": string, "id": number, "id": string, "source":
   string, "text": string,
   "user": { "id": number, "id": string, "screen_name": string,
   "location": string, "followers_count": number },
   "retweet_count": number }
```

[0108] 空字段和空对象

[0109] JSON记录中可存在空对象或空字段。举例来说，人员坐标(纬度和经度)的记录可为：

[0110] {"coordinates":{}}

[0111] 模式具有相同类型：

[0112] {"coordinates":{}}

[0113] 严格地说，{}称为实例，并且类型是对象。为易于解释，本公开中的实施例和解释不同于严格的JSON。

[0114] 类似地，以下对象

[0115] {"geo":null}

[0116] 具有相同类型：

[0117] {"geo":null}

[0118] 如果后续记录具有所述对象的值，那么通过应用合并来填写空对象。举例来说，记录：

[0119] {"coordinates":{}}

[0120] {"coordinates":{"type":"Point"})

[0121] 将产生模式

[0122] {"coordinates":{"type":string}}

[0123] 空类型类似地由所观测到的类型替换。举例来说，记录

[0124] {"geo":null}

[0125] {"geo":true}

[0126] 将产生模式：

[0127] {"geo":boolean}

[0128] 数组

[0129] 推文通常含有例如主题标签(突出的主题词)、url和其它推特用户的提及等项目。举例来说,推特消防带可自动地剖析和提取这些项目以包括在推特的JSON对象中。在以下实施例中,主题标签元数据用来说明如何推断数组的模式。

[0130] 首先,考虑在以下推文(或字符串)中提取和记录主题标签的起始偏移量列表:

[0131] "#donuts #muffins #biscuits"

[0132] 那些偏移量可用如下数组来表示:

[0133] {"offsets":[0,8,17]}

[0134] 源数据中的数组在模式中表示为含有在源数组中找到的元素的类型的数组,其没有特定次序。因此,用于以上对象的模式为:

[0135] {"offsets":[number]}

[0136] 可能想要包括主题标签连同偏移量以用于稍后处理。在这种情况下,推文对象可如下在数组中枚举主题标签和偏移量:

[0137] {"tags":[0,"donuts",8,"muffins",17,"biscuits"]}

[0138] 对应的模式将在数组中包括两种类型:

[0139] {"tags":[number,string]}

[0140] 替代地,可如下颠倒标签与偏移量:

[0141] {"tags":["donuts",0,"muffins",8,"biscuits",17]}

[0142] 并且,因为"tags"数组可含有字符串或数字,所以所得模式为:

[0143] {"tags":[string,number]}

[0144] 事实上,标签文字和标签偏移量可包括在邻近对象中:

[0145] {"tags":["donuts","muffins","biscuits"]},

[0146] {"tags":[0,8,17]}

[0147] 现在具有"tags"的两个模式:

[0148] {"tags":[string]}和{"tags":[number]}

[0149] 在这种情况下,数组处于相同深度并且可进行合并来产生如上相同模式:

[0150] {"tags":[string,number]}

[0151] 另外,请注意以下模式是相同的:

[0152] {"tags":[string,number]}

[0153] {"tags":[number,string]}

[0154] 这是因为类型列表被视为一个集合。在可能的情况下合并数组元素的类型,并且针对数组内部的对象和数组进一步执行合并。在各种其它实施方案中,可保持类型的次序以及类型之间的相依性。然而,这可能会使模式不简洁得多。

[0155] 嵌套对象

[0156] 为了说明嵌套对象,假设如下记录开始偏移量和结束偏移量:

[0157] {"tags":[{"text":"donuts","begin":0},{"text":"donuts",

[0158] "end":6}]}

[0159] 所得模式为:

[0160] {"tags":[{"text":string,"begin":number,

[0161] "end":number}]}

[0162] 如图所示,合并对象类型来代替单独地对数组元素进行定型。

[0163] 类似地,在标签字符串和偏移量处于嵌套数组中的情况下:

[0164] {"tags":[["donuts","muffins"],[0,8]]}==>

[0165] {"tags":[[string],[number]]},

[0166] 模式进一步化简为:

[0167] {"tags":[[string,number]]}

[0168] 这是在本公开的各种实施方案中在模式的精度与模式的简洁性之间所作的折衷。

[0169] 如下处理空对象和空数组。因为如上所述来填写空对象,所以以下示例性模式化简是可能的:

[0170] {"parsed":{"tag":{},"tag":{"offset":number}}}

[0171] =>{"parsed":{"tag":{"offset":number}}

[0172] 类似地,针对数组使用所述合并规则,进行以下模式化简:

[0173]
```
{ "tags": [[], [ number ]] } => { "tags": [[ number ]] }
{ "tags": [[], [[]]] } => { "tags": [[[]]] }
{ "tags": [[], [[]], [number]] } => { "tags": [[[]], [number]] }
   => { "tags": [[[], number]]] }
```

[0174] 合并程序

[0175] 为了从先前模式和新对象创建新模式,分析平台首先对新对象进行定型(即,计算其模式)。这个程序意图指定用于定型的规范语义学,而不描述任何特定实施方案。在以下描述中,变量v、w、v_i、w_j涉及任何有效JSON值,而j、k、j_m、k_n涉及有效字符串。用于定型的基本规则为:

[0176]
```
type(scalar v) = scalar_type of v
type({ k_1: v_1, ..., k_n: v_n }) =
   collapse({ k_1: type(v_1), ..., k_n: type(v_n) })
type([ v_1, ..., v_n ]) =
   collapse([ type(v_1), ..., type(v_n) ])
```

[0177] 第一条规则简单地陈述,针对例如3或“a”等标量,直接从值本身(针对3的数字或针对“a”的字符串)推断对应类型。第二条规则和第三条规则使用折叠函数递归地对对象和数组进行定型。

[0178] 折叠函数重复地合并对象中的相同字段的类型,并且合并数组内部的对象、数组和共用类型。其持续递归进行,直至达到标量类型为止。对于对象,折叠函数为:

[0179]
```
collapse({ k_1: v_1, ..., k_n: v_n }):
    while k_i == k_j:
        if v_i, v_j are scalar types and v_i == v_j OR
           v_i, v_j are objects OR v_i, v_j are arrays:
           replace {..., k_i: v_i, ..., k_j: v_j, ...}
           with    {..., k_i: merge(v_i, v_j), ...}
```

[0180] 对于数组,折叠函数为:

[0181]
```
collapse([ v_1, ..., v_n ]):
    while v_i, v_j are scalar types and v_i == v_j OR
      v_i, v_j are objects OR v_i, v_j are arrays:
        replace [..., v_i, ..., v_j, ...]
        with    [..., merge(v_i, v_j), ...]
```

[0182] 合并函数描述如何成对组合值来去除重复并且组合数组/映射。对于对象，合并仅递归地调用折叠来折叠共用字段：

[0183]
```
merge(v, v) = v
merge({}, { k_1: v_1, ..., k_n: v_n }) = { k_1: v_1, ..., k_n: v_n
   }
merge({ j_1: v_1, ..., j_n: v_n }, { k_1: w_1, ..., k_m: w_m } )
    = collapse({ j_1: v_1, ..., j_n: v_n, k_1: w_1, ..., k_m: w_m
   })
```

[0184] 类似地，对于数组：

[0185] merge([ ],[v_1,...,v_n])＝[v_1,...,v_n]

[0186] merge([v_1,...,v_n],[w_1,...,w_m])

[0187] ＝collapse([v_1,...,v_n,w_1,...,w_m])

[0188] 保留空值，例如此处所展示：

[0189] merge({"coordinates":{}},{"coordinates":null},

[0190] {"coordinates":[]})

[0191] ＝{"Coordinates":{},"coordinates":[],"coordinates":null}

[0192] JSON空值是一个值，就如同数字9是一个值。在关系式中，空值指示没有指定值。在SQL中，空值被呈现为tags<null>:boolean，其中如果存在空值，那么布尔值为真，否则空值。为了为SQL用户简化模式，如果用户不需要区分JSON空值与SQL空值，那么可省略coordinates<null>列。

[0193] 累积实施例

[0194] 使用以上简单规则，有可能对深嵌套JSON记录进行定型。举例来说，考虑表示网页的页面浏览量统计信息的复杂假设记录：

[0195] {"stat":[10,"total_pageviews",{"counts":[1,[3]],

[0196] "page_attr":7.0},{"page_attr":["internal"]}]}

[0197] 将产生以下模式：

[0198]
```
{ "stat": [number,
           string,
           { "counts": [number, [number]],
             "page_attr": number,
             "page_attr": [string]
           }]}
```

[0199] 在各种实施方案中，可使用JSON模式格式来编码所推断出的模式。这种格式被标准化，并且可易于扩展为并入额外元数据(例如，对象是否为映射)。然而，这是非常冗长并且浪费空间的，所以在本公开中不用于实施例。举例来说，按照JSON模式格式，如下表示以上模式：

[0200]

```
{
  "type": "object",
  "properties": {
    "stat": {
      "items": {
        "type": [
          "number",
          "string",
          {
            "type": "object",
            "properties": {
              "counts": {
```

[0201]

```
                    "items": {
                      "type": [
                        "number",
                        {
                          "items": {
                            "type": "number"
                          },
                          "type": "array"
                        }
                      ]
                    },
                    "type": "array"
                  },
                  "page_attr": {
                    "type": [
                      "number",
                      {
                        "items": {
                          "type": "string"
                        },
                        "type": "array"
                      }
                    ]
                  }
                }
              }
            ]
          },
          "type": "array"
        }
      ]
    }
```

[0202] 映射修饰

[0203] 开发者和分析者可出于许多不同目的来使用JSON对象和数组。明确地说,JSON对象常常被用作对象和“映射”。举例来说,开发者可能创建一个对象,其中字段是日期并且值是如页面浏览量等收集统计信息。另一个实施例是当字段是用户id并且值是简档时。在这些情况下,对象更像映射数据结构而非静态对象。用户并不总是知道可能的字段名称,因为存在如此多的名称,并且字段名称是动态地创建的。因而,用户可能想要以查询值的相同方式查询字段。

[0204] 为了支持这种使用,分析平台能够识别映射。分析平台并入有试探法来识别映射,并且还允许用户指定应当将哪些嵌套对象视为映射以及不应将哪些嵌套对象视为映射。将对象标记为映射称为修饰。

[0205] 一般来说,在初始加载之后执行修饰——也就是说,不必在初始摄取时识别映射。可稍后在第二遍时或在已经摄取更多数据之后执行修饰。另外,如果需要的话,可简单地将

映射恢复回到对象。

[0206] 默认地，将JSON对象视为对象(或者，按C命名法，结构体)。这可在JSON模式中通过用"obj_type":object表示对象来明确地指示。以下实施例中所使用的速记表示法为O{}。

[0207] 为了标记映射，试探法寻找作为一个群组与其包含对象(容器)相比相对不频繁出现的字段。对于映射，使用速记M{}。

[0208] 当在第一遍时计算模式时，跟踪字段出现的频率。考虑在数据集中以频率F出现的对象(或嵌套对象)。令v_i为对象中的字段i的频率，并且N为对象的唯一字段的数目(不管其类型如何)。比率(sum(v_i)/N)/F是平均字段频率与容器的频率的比率。如果这个比率低于阈值(例如0.01，其可为可由用户配置的)，那么包含对象被指明为映射。在各种实施方案中，JSON模式中的空对象被视为映射。

[0209] 创建关系模式

[0210] 在推断出源数据集中的JSON对象的模式之后，分析平台产生可暴露给SQL用户和基于SQL的工具的关系模式。目标是创建在JSON模式中表示包含关系的简洁模式，同时给予用户标准SQL的力量。这个关系模式是从被修饰的JSON模式产生的，并且是底层半结构化数据集的视图。此处首先呈现如何将JSON模式转换为关系视图的一些实施例，之后论述用于执行转换的一般化程序。

[0211] 对象

[0212] 最简单的实施例是具有简单标量类型的对象，例如以下模式：

[0213] {"created_at":string,"id":number,"text":string,

[0214] "source":string,"favorited":boolean}

[0215] 在这种情况下，对象的字段直接转化为关系式的列：

[0216] Root(created_at:str,id:num,text:str,source:str,favorited:

[0217] bool)

[0218] 顶端对象的关系式(或表格)在此处称为“根”，但其可由(例如)源集合的名称替换，如果此类名称存在的话。为了空间和易读性，已经将类型名称字符串、数字和布尔值缩写为str、num和bool。

[0219] 可将类型添加到属性名称以支持类型多态性。举例来说，考虑以下模式：

[0220] {"created_at":string,"id":number,"id":string,"text":

[0221] string,"source":string,"favorited":boolean}

[0222] 所得的关系模式将接着具有单独"id"和"id"列：

[0223] Root(cteated_at:str,id<num>:num,id<str>:str,

[0224] source:str,text:str,favorited:bool)

[0225] 嵌套对象

[0226] 嵌套对象产生具有外键关系的新关系式。举例来说，考虑JSON模式：

[0227]
```
{ "created_at": string, "id": number, "source": string, "text":
  string,
  "user": { "id": number, "screen_name": string },
  "favorited": boolean }
```

[0228] 对应的关系模式为

[0229] Root(created_at:str,id:num,source:str,text:str,favorited:

[0230] bool,user:join_key)

[0231] Root.user(id_jk:join_key,id:num,screen_name:str)

[0232] 嵌套对象被“正规化”为由其路径命名的单独关系式,在这种情况下为"Root.user"。表示子对象的新表格中的列"Root.user"."id_jk"是用于列"Root.user"(表格"Root"中的"user"列)的外键。类型被指定为“联接键”来将其与其它列区别开,但在实际实施方案中,join_key类型通常是整数。

[0233] 对象可被嵌套几级深。举例来说,转推对象可包括转推状态对象,其包括转推的用户的简档,从而得到以下模式:

[0234]
```
{ "created_at": string, "id": number, "source": string, "text":
   string,
   "user": { "id": number, "screen_name": string },
   "retweeted_status": { "created_at": string, "id": number,
   "user": { "id": number, "screen_name": string } },
   "favorited": boolean }
```

[0235] 对应的关系视图为:

[0236]
```
Root(created_at: str, id: num, source: str,
   text: str, favorited: bool,
   user: join_key, retweeted_status: join_key)
Root.user(id_jk: join_key, id: num, screen_name: str)
Root.retweeted_status(id_jk: join_key, created_at: str, id: num,
   user: join_key)
Root.retweeted_status.user(id_jk: join_key, id: num, screen_name:
   str)
```

[0237] 请注意,"Root.user"、"Root.retweeted_status"和"Root.retweeted_status.user"全部被分离成不同表格。

[0238] 优化1对1关系

[0239] 在嵌套对象关系中,通常在主表格中的行与嵌套对象的表格中的行之间存在1对1关系。因而,这些可使用列名称的点分表示法来1对1地折叠成单个表格。

[0240] 举例来说,以上多关系式实施例展平为:

[0241] Root(created_at:str,id:num,source:str,

[0242] text:str,favorited:bool,

[0243] user.id:num,user.screen_name:str)

[0244] 并且,对于三级嵌套对象实施例,

[0245]
```
Root(created_at: str, id: num, source: str,
   text: str, favorited: bool,
   user.id: num, user.screen_name: str,
   retweeted_status.created_at: str,
   retweeted_status.id: num,
   retweeted_status.user.id: num,
   retweeted_status.user.screen_name: str)
```

[0246] 请注意,由于关系模式仅仅是JSON模式的视图,所以可在不修改底层数据的情况下由分析平台按照需要向用户呈现展平、部分展平或分离(未展平)的关系模式。仅有的限

制是不向用户呈现冲突的表格定义。

[0247] 映射

[0248] 在不将字段集指明为映射的情况下,对应的关系模式可包括大量列。另外,用户可能想要查询字段名称;举例来说,他们可能想要查找12月的平均页面浏览量。

[0249] 为了解决这些问题,可"枢转"被修饰为映射的(嵌套)对象的表格。举例来说,考虑用于记录网站上的每个页面的各种量度(每日页面浏览量、点击量、花费时间等)的以下模式:

[0250]
```
O{ "page_url": string, "page_id": number,
   "stat_name": string,
   "metric": M{ "2012-01-01": number, "2012-01-02": number, ...,
   "2012-12-01": number, ...}}
```

[0251] 可在关系式中将字段和值存储为键-值对,而不是产生具有针对每一天的单独列的表格:

[0252] Root(page_url:str,page_id:num,stat_name:str,metric<map>:

[0253] join_key)

[0254] Root.metric<map>(id_jk:join_key,key:string,val:num)

[0255] 在这种情况下,id列是外键;指示每个映射条目最初存在于哪个记录内。对于一年的页面浏览量,并不是在表格"Root.metric"中具有365列,而是只有两列。"key"列存储字段名称,并且"val"列存储值。举例来说,对于以上模式,数据库可含有"www.noudata.com/jobs"(page_id 284)的这些记录:

[0256] Root("www.noudata.com/jobs",284,"page_views",3),

[0257] Root.metric<map>(3,"2012-12-01",50),

[0258] Root.metric<map>(3,"2012-12-02",30),...

[0259] 当映射中具有类型多态性时,枢转仍然起作用。举例来说,假设量度表示情绪,其含有类别和指示类别的强度的得分:

[0260]
```
{ "page_url": "www.noudata.com/blog", "page_id": 285, "stat_name":
   "sentiment"
   "metric": { "2012-12-01": "agreement", "2012-12-01": 5,
               "2012-12-05": "anger", "2012-12-05": 2, ... } }
```

[0261] JSON模式将为:

[0262]
```
O{ "page_url": string, "page_id": number,
   "stat_name": string,
   "metric": M{ "2012-12-01": string, "2012-12-01": number, ...,
                "2012-12-05": string, "2012-12-05": number, ...}}
```

[0263] 当创建关系模式时,可将新"val"列添加到映射关系式以包括新类型。其它"val"列同样可附加有其类型以区别列名称,如所示:

[0264]
```
Root(page_url: str, page_id: num, stat_name: str, metric<map>:
   join_key)
Root.metric<map>(id_jk: join_key, key: string,
   val<str>: str, val<num>: num)
```

[0265] 从以上JSON对象得到的条目将呈现为:

[0266] Root.metric<map>(4,"2012-12-01","agreement",NULL),

[0267] Root.metric<map>(4,"2012-12-01",NULL,5),

[0268] Root.metric<map>(4,"2012-12-05","anger",NULL),

[0269] Root.metric<map>(4,"2012-12-05",NULL,2)...

[0270] 一旦这些映射被枢转,用户就可向键列施加判断和函数,正如他们将对任何其它列所作。

[0271] 嵌套映射

[0272] 基本原理对于嵌套映射来说是相同的。考虑每天和每小时的统计信息列表:

[0273]
```
M{"2012-12-01": M{ "12:00": number,
                   "01:00": number,
                   "02:00": number,
                    ... },
  "2012-12-02": M{ ... },
   ... }
```

[0274] 所得模式将为

[0275] Root(id_jk:join_key,key:string,val<map>:join_key)

[0276] Root.val<map>(id_jk:join_key,key:string,val<num>:num)

[0277] 对象也可嵌套在映射内部:

[0278]
```
M{"2012-12-01": O{ "sentiment": string,
                   "strength": number }
  "2012-12-02": O{ ... }
   ... }
```

[0279] 所得的展平关系模式为:

[0280] Root(id_jk:join_key,key:string,val<map>:join_key)

[0281] Root.val<map>(id_jk:join_key,sentiment:string,

[0282] strength:number)

[0283] 空元素

[0284] 空对象有时出现在数据中。考虑以下模式:

[0285] {"created_at":string,"id":number,"source":string,"text":

[0286] string,

[0287] "user":{"id":number,"screen_name":string}}

[0288] 可在没有用户信息的情况下接收JSON对象,如此处所示:

[0289]
```
{ "created_at": "Thu Nov 08",
  "id": 266353834,
  "source": "Twitter for iPhone",
  "text": "@ilstavrachi: would love dinner. Cook this:
  http://bit.ly/955Ffo",
  "user": { } }
```

[0290] 空用户对象可用以下关系元组表示:

[0291] Root("Thu Nov 08",266353834,"Twitterfor iPhone","@ilstavrachi:

[0292] would love dinner.Cook this:http://bit.ly/955Ffo",join_key)

[0293] Root.user(join_key,NULL,NULL)

[0294] 如果所有所摄取的用户对象在所摄取的流中具有空对象,那么所得的JSON模式将包括空对象。举例来说,见这个模式中的最终字段("user"):

[0295] {"id":number,"user":{}}

[0296] 在这种情况下,空对象"user"可被视为映射,从而得到以下关系模式:

[0297] Root(id:num,user<map>:join_key)

[0298] Root.user<map>(id_jk:join_key,key:string)

[0299] 请注意,Root.user<map>没有任何值列,并且最初是空的。然而,这种设计使得稍后在模式随着摄取新对象而改变的情况下便于添加列,因为Root中的每个记录将已经被指派联接键。

[0300] 数组

[0301] 类似于映射来处理数组,所以模式转化相当类似。主要差异是映射的字符串"key"字段由对应于数组索引的整数(int)类型的"index"字段替换。简单的实施例是:

[0302] {"tags":[string]}

[0303] 这得到以下关系模式:

[0304] Root(tags<arr>:join_key)

[0305] Root.tags<arr>(id_jk:join_key,index:int,val<str>:str)

[0306] 类型多态性和嵌套数组以针对映射的相同方式起作用。考虑以下模式:

[0307] {"tags":[number,string]}

[0308] 这得到以下关系模式:

[0309] Root(tags<arr>:join_key)

[0310] Root.tags<arr>(id_jk:join_key,index:int,val<num>:num,val<str>:str)

[0311] 对象可嵌套在数组内,如此处:

[0312] {"tags":[{"text":string,"offset":number}]}

[0313] 所得的关系模式可被创建为:

[0314] Root(tags<arr>:join_key)

[0315] Root.tags<arr>(id_jk:join_key,index:int,val:join_key)

[0316] Root.tags<arr>.val(id_jk:join_key,text:str,offset:num)

[0317] 使用1对1展平优化,关系模式变为:

[0318] Root(tags<arr>:join_key)

[0319] Root.tags<arr>(id_jk:join_key,index:int,val.text:str,val.offset:num)

[0320] 嵌套数组和空数组

[0321] 可按与映射类似的方式针对嵌套数组和空数组创建关系模式。对于以下模式:

[0322] {"tags":[string,[number]],"urls":[]}

[0323] 关系模式将为:

[0324]
```
Root(tags<arr>: join_key, urls<arr>: join_key)
Root.tags<arr>(id_jk: join_key, index: int,
                val<str>: str, val<arr>: join_key)
Root.tags<arr>.val<arr>(id_jk: join_key, index: int,
                        val<num>: num)
Root.urls<arr>(id_jk: join_key, index: int)
```

[0325] 请注意,对于嵌套数组,创建单独表格,其中"val"附加到表格名称。对于空数组,创建只有"index"列而没有"val"列的单独表格,稍后一旦观测到数组的内容并对其进行定型,就可添加"val"列。

[0326] 对原子值的类型推断

[0327] 以上类型推断和转换为关系模式的程序依赖于JSON中可用的基本类型。相同的程序同样适用于所选择的任何定型系统。换句话说,分析平台可推断像整数、浮点数和时间等较窄标量类型,只要可从值推断出原子标量类型。BSON和XML具有这些扩展类型系统。此外,可使用各种试探法(例如正规表达式)来检测例如日期和时间等较复杂类型。

[0328] 由于ANSI SQL不支持与JSON相同的类型,所以将所推断出的类型转换为迄今针对关系视图所见到的最具体类型。举例来说,如果针对字段"freq"仅已经看到整数,那么将在关系模式中针对"freq"把数字类型转换为整数。类似地,如果已经观测到整数和浮点数,那么关系模式将把"freq"列展示为浮点数。同样,在关系模式中字符串字段转换为字符变化类型。换句话说,可跟踪比基本JSON类型更具体的类型。

[0329] 替代方案是依赖于类型多态性并且使用较具体定型系统来推断数据值的类型。也就是说,代替使用JSON原始类型,使用ANSI SQL的原始类型。

[0330] 以下是在摄取期间跟踪的类型列表(在左侧)以及如何针对SQL模式对其进行转换(在右侧)。大多数SQL数据库支持客户端可根据需要使用的包括文字在内的额外类型。请注意:ObjectId类型专用于BSON。

[0331] int32,-->INTEGER

[0332] int64,-->INTEGER

[0333] double,-->DOUBLE PRECISION

[0334] string,-->VARCHAR

[0335] date,-->DATE

[0336] bool,-->BOOLEAN

[0337] object_id,(BSON)-->VARCHAR(24)

[0338] time-->TIME

[0339] timestamp-->TIMESTAMP

[0340] 程序

[0341] 可使用嵌套JSON模式结构的递归解包来完成从JSON模式转换为关系模式。此处展示示例性实施方案的伪码表示。

[0342]

```
Call for every attribute in topmost object:
attr_schema, "Root", attr_name

create_schema(json_schema, rel_name, attr_name):

/*如果其被修饰为对象，那么创建表格（关系式）*/
if json_schema is object:
   Add join key called attr_name to relation rel_name
   new_rel = rel_name + "." + attr_name

   Create relation new_rel
   add (id_jk: join_key) to new_rel

   /*递归地将属性添加到表格（关系式）*/
   for attr, attr_schema in json_schema:
     create_schema(attr_schema, new_rel, attr)

/*针对（嵌套）映射创建恰当的属性和表格*/
else if json_schema is map:
   Add join key called 'attr_name + <map>' to relation rel_name
   new_rel = rel_name + "." + attr_name<map>

   Create relation new_rel
   Add (id_jk: join_key) and (key: string) to new_rel

   /*递归地将属性添加到表格（关系式）*/
   for each distinct value type val_type in json_schema:
     create_schema(val_type, new_rel, "val")

/*针对数组创建恰当的属性和表格*/
else if json_schema is array:
   Add join key called 'attr_name + <arr>' to relation rel_name
   new_rel = rel_name + "." + attr_name<arr>

   Create relation new_rel

   Add (id_jk: join_key) and (index: int) to new_rel

   /*递归地将属性添加到表格（关系式）*/
   for each distinct item type item_type in json_schema:
     create_schema(item_type, new_rel, "val")

/*原始类型，将列添加到表格（关系式）*/
else:
   If attr_name does not exist in relation rel_name:
     Add column (attr_name, attr_name's type) to relation
     rel_name
   else
     Rename attribute attr_name to attr_name + "<orignal
     attr_name's type>" in relation rel_name
     Add column (attr_name + "<" + attr_name's type + ">",
     attr_name's type) to relation rel_name
```

[0343] 以上程序将在没有1对1优化的情况下创建关系模式。可通过所述关系模式执行第二遍，识别具有1对1关系的对象表格并且将其折叠。或者，可内联执行1对1优化，但出于清楚起见而未展示这点。当具有嵌套对象的模式的子树未被数组或映射“打断”时，整个对象子树可折叠成具有由其通往子树的根的路径命名的属性的单张表格。作为映射或对象的属

性保持在单独表格中，但内部所含有的任何子对象可递归地折叠。这些原理适用于嵌套对象的任何任意深度。

[0344] 用数据填充索引

[0345] 一旦已经响应于新对象来更新了JSON和关系模式，就可将对象中所含有的数据存储在索引中，如下文所描述。

[0346] 分析平台中的索引依赖于存储键-值对的保序索引。索引支持以下操作：lookup(prefix)、insert(key,value)、delete(key)、update(key,value)以及用于范围搜索的get_next()。许多数据结构和低级库支持此类接口。实施例包括BerkeleyDB、TokyoCabinet、KyotoCabinet、LevelDB等等。这些在内部使用保序二级存储数据结构，如B树、LSM(日志结构化合并)树和分形树。可能存在特殊情况，其中例如针对对象ID，使用非保序索引(例如散列表格)。使用非保序索引，可能会牺牲get_next()和进行范围搜索的能力。

[0347] 在各种实施方案中，分析框架使用LevelDB，其实施LSM树，进行压缩，并且用高插入速率向数据集提供良好性能。LevelDB还做出可能与分析框架的常用模型一致的性能折衷。举例来说，当分析例如日志数据等数据时，将频繁地添加数据，但将很少或从不改变现有数据。有利的是，以较慢的数据删除和数据修改为代价来优化LevelDB以实现快速数据插入。

[0348] 保序索引具有按键次序并置键-值对的性质。因此，当在某个键附近搜索键-值对或按序检索项目时，将比在无序检索项目时快速得多地返回响应。

[0349] 分析平台可针对每个源集合维持多个键-值索引，并且在一些实施方案中，针对每个源集合维持两个到六个索引。分析平台使用这些索引来评估对关系模式的SQL查询(不需要物化SQL模式)。每个对象被指派由tid指示的唯一id。可从中重构其它索引和模式的两个索引为BigIndex(BI)和ArrayIndex(AI)。

[0350] BigIndex(BI)

[0351] BigIndex(BI)是存储数据中的未嵌入在数组中的所有字段的基本数据存储区。可基于col_path和tid通过键从BI检索值(val)。

[0352] <col_path,tid>->val

[0353] col_path是从根对象到字段的路径，其附加有字段的类型。举例来说，对于以下记录：

[0354] 1:{"text":"Tweet this","user":{"id":29471497,

[0355] "screen_name":"Mashah08"}}

[0356] 2:{"text":"Tweet that","user":{"id":27438992,

[0357] "screen_name":"binkert"}}

[0358] 以下键-值对被添加到BI：

[0359] (root.text<str>,1)-->"Tweet this"

[0360] (root.text<str>,2)-->"Tweet that"

[0361] (root.user.id<num>,1)-->29471497

[0362] (root.user.id<num>,2)-->27438992

[0363] (root.user.screen_name<str>,1)-->"Mashah08"

[0364] (root.user.screen_name<str>,2)-->"binkert"

[0365] 在各种实施方案中,底层索引存储区(例如LevelDB)不知道键的片段的重要性。换句话说,尽管"root.text<str>,1"表示根表格中的字符串文字字段的第一个元素,但索引存储区仅仅可看到无差别的多字符键。作为简单的实施例,可简单地通过串联col_path和tid(重要的是,按那个次序)来创建键。举例来说,上文所示范的第一个键可被传递到索引存储区作为"root.text<str>1"。索引存储区将并置第二键("root.text<str>2")与第一键,这不是因为对路径相似性的任何理解,而是仅仅因为前14个字符是相同的。即使列名称和类型被存储为每个键的一部分,但由于种类排序,可使用压缩(例如基于前缀的压缩)来降低存储成本。

[0366] 在BI中,源数据的所有列被组合成单个结构,这不同于针对每个新列创建单独列文件的传统列存储区。BI方法允许单个索引实例并且还使得能够延迟映射检测。由于新字段仅出现为BI中的条目,所以未能枢转映射并不会招致为稍后变为映射的每个字段创建大量物理文件的物理成本。

[0367] 在BI中,并置每个属性或"列"的键-值对。因此,如同列文件,BI允许查询执行器集中于查询中的关注字段上,而非迫使其扫过含有查询中未提及的字段的数据。

[0368] ArrayIndex(AI)

[0369] 虽然可将针对数组的来自正规化表格的字段添加到BI,但数组索引接着将来自其对应值。代替地,可将数组字段添加到保留索引信息并且允许同一数组中的条目由索引存储区并置的单独ArrayIndex(AI),这为许多查询提供良好性能。可使用以下签名将数组值存储在AI中:

[0370] <col_path,tid,join_key,index)->val

[0371] col_path是数组字段的路径:举例来说,用于标签数组中的元素的"root.tags"或用于标签数组内部的对象中的"text"字段的"root.tags.text"。join_key和索引是数组的外键和值的索引。还存储tid来避免必须针对每个数组将单独条目存储在BI中。可使用tid来查找同一对象中的对应列的值。考虑不同推文中的表示主题标签的对象:

[0372] 1:{"id":3465345,"tags":["muffins""cupcakes"]}

[0373] 2:{"id":3465376,"tags":["curry""sauces"]}

[0374] 对于这些对象,标签表格具有以下模式:

[0375] Root.tags<arr>(id_jk:join_key,index:int,val:string)

[0376] 对于那张表格,AI中的条目将为:

[0377] (root.tags<art>,1,1,0)-->"muffins"

[0378] (root.tags<arr>,1,1,1)-->"cupcakes"

[0379] (root.tags<arr>,2,2,0)-->"curry"

[0380] (root.tags<arr>,2,2,1)-->"sauces"

[0381] 数组索引允许迅速迭代通过数组字段的值。举例来说,当对这些字段执行统计(例如,总和、平均值、方差等)、查找特定值等时,这是有用的。

[0382] 嵌套数组实施例

[0383] 请注意,对于根对象中的数组(顶端数组),tid和join_key字段是冗余的(见上文)并且可被优化掉。然而,对于嵌套数组,需要单独join_key,并且其不是多余的。举例来说,考虑这个JSON对象:

[0384] 1:{"id":3598574,"tags":[[8,25,75],["muffins","donuts",

[0385] "pastries"]]}

[0386] 对应的关系模式是：

[0387] Root.tags<arr>(id_jk:join_key,index:int,val<arr>:join_key)

[0388] Root.tags<arr>.val<arr>(id_jk:join_key,index:int,val<num>:

[0389] num,val<str>:str)

[0390] 记得AI使用以下键-值对

[0391] col_path,tid,join_key,index->val

[0392] 这得到这些AI条目

[0393] tags<arr>.val<arr>,1,1,0->1

[0394] tags<arr>.val<arr>,1,1,1->2

[0395] (数字数组)

[0396] tags<arr>.val<arr>.val<num>,1,1,0->8

[0397] tags<arr>.val<arr>.val<num>,1,1,1->25

[0398] tags<arr>.val<arr>.val<num>,1,1,2->75

[0399] (字符串数组)

[0400] tags<arr>.val<arr>.val<str>,1,2,0->"muffins"

[0401] tags<arr>.val<arr>.val<str>,1,2,1->"donuts"

[0402] tags<arr>.val<arr>.val<str>,1,2,2->"pastries"

[0403] 请注意，假如从嵌套数组键-值对中移除联接键，那么将不可能知道muffins是第一嵌套数组还是第二嵌套数组的一部分。因此，联接键对于顶端数组是冗余的，而对于嵌套数组的情况不是冗余的。

[0404] 数组索引2(AI2)

[0405] 虽然这两个索引(BI和AI)足以重构所有所摄取的数据，但它们并不有效地支持多种访问型式。对于这些访问型式，引入以下索引，可任选地创建所述索引来以额外空间为代价改善性能。

[0406] 这具有以下签名：

[0407] (col_path,index,tid,join_key)->val

[0408] 其允许迅速地找到数组的特定索引元素。举例来说，使用AI2返回索引10处的所有标签(tags[10])是简单且快速的。

[0409] 映射索引(MI)

[0410] 映射索引在其功能性和签名方面类似于数组索引：

[0411] (col_path,tid,join_key,map_key)->val

[0412] 主要差异是映射索引不是在初始摄取期间构建的，而是异步构造的。在初始加载期间，映射将被处理为对象并且照常被插入到BI中。一旦两者被填充，BI和MI两者中有许多条目可用以实现较有效的查询处理。BI条目保持相关，以防用户或管理员要求解除修饰映射。只需要改变关系模式，并且接着将在查询中使用对应于未映射数据的原始BI条目。

[0413] 如同AI，当迭代通过映射的元素时，MI是有用的：用于应用统计函数、用于局限于特定字段名称等。再次考虑维持页面浏览量统计信息的对象：

[0414] 1:{"url":"noudata.com/blog",

[0415] "page_views":{"2012-12-01":10,"2012-12-02":12,..."2012-

[0416] 12-15":10}

[0417] 2:{"url":"noudata.com/jobs",

[0418] "page_views":{"2012-12-01":2,"2012-12-02":4,..."2012-

[0419] 12-15":7}

[0420] 在标记为映射的情况下用于page_views表格的关系模式为：

[0421] Root.page_views<map>(id_jk:join_key,key:string,val:num)

[0422] 其中键是映射的键，并且值是相关联的值。对于以上对象，MI中的条目将为：

[0423] (root.page_views<map>,1,1,"2012-12-01")-->10

[0424] (root.page_views<map>,1,1,"2012-12-02")-->12

[0425] ...

[0426] (root.page_views<map>,1,1,"2012-12-15")-->10

[0427] (root.page_views<map>,2,2,"2012-12-01")-->2

[0428] (root.page_views<map>,2,2,"2012-12-02")-->4

[0429] ...

[0430] (root.page_views<map>,2,2,"2012-12-05")-->7

[0431] 这种排序允许针对每个页面并置page_views映射中的值，而在BI中，将通过日期来并置所述值。

[0432] 映射索引2(MI2)

[0433] 另外，可实施辅助映射索引。映射索引在其功能性和签名方面类似于数组索引：

[0434] (col_path,map_key,tid,join_key)->val

[0435] 这允许有效搜索特定映射元素，例如“对应于映射键2012-12-05的所有不同值”。可如下书写AI2和MI2两者的一般表示：

[0436] (col_path,key,tid,join_key)->val

[0437] 其中键对应于数组的索引或映射的map_key。

[0438] ValueIndex(VI)

[0439] 虽然以上索引对于查找特定字段的值并且迭代通过那些值是有用的，但如果查询正仅查找特定值或值范围，那么其不允许快速访问。举例来说，查询可要求返回由"mashah08"书写的推文的文字。为了辅助此类查询，可针对模式中的一些或所有字段构建ValueIndex。ValueIndex可在摄取数据时构建或稍后异步构建。值索引的键为：

[0440] (col_path,val)

[0441] 其中val是源数据中的属性的值。VI中的那个键的对应值取决于在哪里出现所述值的字段。对于以上每个索引，其改变：

[0442] BI:(col_path,val)-->tid

[0443] AI:(col_path,val)-->tid,join_key,index

[0444] MI:(col_path,val)-->tid,join_key,key

[0445] 举例来说，推文：

[0446] 1:{"text":"Tweet this","user":{"id":29471497,

[0447] "screen_name":"mashah08"}}

[0448] 2:{"text":"Tweet that","user":{"id":27438992,

[0449] "screen_name":"binkert"}}

[0450] 被存储为：

[0451]
```
(root.text<string>, "Tweet this")           --> 1
(root.text<string>, "Tweet that")           --> 2
(root.user.id<num>, 29471497)               --> 1
(root.user.id<num>, 27438992)               --> 2
(root.user.screen_name<string>, "Mashah08") --> 1
(root.user.screen_name<string>, "binkert")  --> 2
```

[0452] 使用VI,可通过查找键:(root.user.screen_name,"mashah08")并且检索所有关联tid来搜索由"mashah08"创作的所有推文。接着,可使用所检索的tid来搜索BI以返回每条推文的对应文字。

[0453] 索引尤其是值索引的成本是额外存储空间以及在将新对象添加到系统时对其进行更新所需要的执行时间。由于空间或更新开销的缘故,用户可能由于这些原因而不想对所有可能路径进行索引。因而,用户可指定在VI中对哪些路径进行索引。

[0454] RowIndex(RI)

[0455] 为了有助于重现整个所摄取的对象(类似于在传统的基于行的存储区中索取记录),可实施RowIndex(RI)。RowIndex存储键-值对

[0456] tid-->JSON object

[0457] JSON对象可被存储为字符串表示、BSON或任何其它串行化格式,例如用于JSON对象的内部表示的树结构。对于上文相对于VI所论述的两条推文,对应的RI条目将为：

[0458] 1-->{"text":"Tweet this","user":{"id":29471497,

[0459] "screen_name":"mashah08"}}

[0460] 2-->{"text":"Tweet that","user":{"id":27438992,

[0461] "screen_name":"binkert"}}

[0462] 实施例

[0463] BI、AI、MI和VI的实施例。考虑与上文类似的推文,其中添加了"retweet_freq"属性,所述属性记录一条推文在一天内被转推多少次：

[0464] 1:{"text":"Love#muffins and#cupcakes:bit.ly/955Ffo",

[0465] "user":{"id":29471497,"screen_name":"mashah08"},

[0466] "tags":["muffins","cupcakes"],

[0467] "retweet_freq":{"2012-12-01":10,"2012-12-02":13

[0468] "2012-12-03":1}}

[0469] 2:{"text":"Love#sushi and#umami:bit.ly/955Ffo",

[0470] "user":{"id":28492838,"screen_name":"binkert"},

[0471] "tags":["sushi","umami"],

[0472] "retweet_freq":{"2012-12-04":20,"2012-12-05":1}}

[0473] 这些记录的模式为：

[0474]

```
O{ "text": string, "user": O{ "id": number,
   "screen_name": string }, "tags": [ string ],
   "retweet_freq": M{ "2012-12-01": number ... "2012-12-05":
   number } }
```

[0475] 这些记录的JSON模式将为

[0476]

```
{
  "type": "object",
  "obj_type": "object",
  "properties": {
    "text": {
      "type": "string"
    },
    "user": {
      "type": "object",
      "obj_type": "object",
      "properties": {
         "id": {
            "type": "number",
          },
         "screen_name": {
            "type": "string",


            }
      }
    },
    "tags": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "retweet_freq": {
      "type": "object",
      "obj_type": "map",
      "properties": {
        "2012-12-01": {
          "type": "number"
        },
    ...
         "2012-12-05": {
          "type": "number"
         }
      }
    }
  }
}
```

[0477] 如果retweet_freq未被视为映射，那么关系模式为：

```
Root (text: str,
      user.id: num, user.screen_name: str,
      tags<arr>: join_key,
      retweet_freq.2012-12-01: num,
      retweet_freq.2012-12-02: num,
      retweet_freq.2012-12-03: num,
      retweet_freq.2012-12-04: num,
      retweet_freq.2012-12-05: num)
Root.tags<arr> (id_jk: join_key,
                index: int,
                val: str)
```

[0479] 在这种情况下，以上示例性记录将如下填充这些关系式：

```
Root:
    ("Love #muffins ...", 29471497, mashah08, 1, 10, 13, 1, NULL,
   NULL)

    ("Love #sushi ...", 28492838, binkert, 2, NULL, NULL, NULL,
   20, 1)

Root.tags<arr>:
    (1, 0, "muffins")
    (1, 1, "cupcakes")
    (2, 0, "sushi")
    (2, 1, "umami")
```

[0481] 请注意，这些是在假如对这些表格运行"select*"查询的情况下查询将返回的元组。这些元组不必在存储引擎中如此物化。也就是说，这可仅为底层数据的虚拟视图，而不如所描绘那样进行物理存储。

[0482] 如果retweet_freq被识别为映射，那么关系模式变得更简洁(并且更适应额外数据)，如下：

```
Root (text: str,
      user.id: num, user.screen_name: str,
      tags<arr>: join_key,
      retweet_freq<map>: join_key)
Root.tags<arr> (id_jk: join_key,
                index: int,
                val: str)
Root.retweet_freq<map> (id_jk: join_key,
                        key: str,
                        val: num)
```

[0484] 对应的元组为：

[0485]
```
Root:
    ("Love #muffins ...", 29471497, mashah08, 1, 1)
    ("Love #sushi ...", 28492838, binkert, 2, 2)
Root.tags<arr>:
    (1, 0, "muffins")
    (1, 1, "cupcakes")
    (2, 0, "sushi")
    (2, 1, "umami")
Root.retweet_freq<map>:
    (1, "2012-12-01", 10)
    (1, "2012-12-02", 13)
    (1, "2012-12-03", 1)
    (2, "2012-12-04", 20)
    (2, "2012-12-05", 1)
```

[0486] 添加到BI的键-值对为：

[0487] (root.retweet_freq.2012-12-01,1)-->10

[0488] (root.retweet_freq.2012-12-02,1)-->13

[0489] (root.retweet_freq.2012-12-03,1)-->1

[0490] (root.retweet_freq.2012-12-04,2)-->20

[0491] (root.retweet_freq.2012-12-05,2)-->1

[0492] (root.text,1)-->"Love#muffins and#cupcakes"

[0493] (root.text,2)-->"Love#sushi and#umami"

[0494] (root.user.id,1)-->29471497

[0495] (root.user.id,2)-->28492838

[0496] (root.user.screenname,1)-->mashah08

[0497] (root.user.screen_name,2)-->binkert

[0498] 添加到AI的键-值对如下。请注意，在这种情况下，联接键是冗余的(与tid一样)，因为没有嵌套数组。

[0499] (root.tags<arr>,1,1,0)-->"muffins"

[0500] (root.tags<arr>,1,1,1)-->"cupcakes"

[0501] (root.tags<arr>,2,2,0)-->"sushi"

[0502] (root.tags<arr>,2,2,1)-->"umami"

[0503] RI将具有以下两个条目

[0504] 1-->{"text":"Love#muffins and#cupcakes:bit.ly/955Ffo",

[0505] "user":{"id":29471497,"screen_name":"mashah08"},"tags":

[0506] ["muffins","cupcakes"],"retweet_freq":{"2012-12-01":

[0507] 10,"2012-12-02":13,"2012-12-03":1}}

[0508] 2-->{"text":"Love#sushi and#umami:bit.ly/955Ffo","user":{

[0509] "id":28492838,"screen_name":"binkert"},"tags":["sushi",

[0510] "umami"],"retweet_freq":{"2012-12-04":20,"2012-12-05":1

[0511] }}

[0512] 如果构建MI并且在构建MI时,MI将具有以下条目:

[0513] (root.retweet_freq<map>,1,1,"2012-12-01")-->10

[0514] (root.retweet_freq<map>,1,1,"2012-12-02")-->13

[0515] (root.retweet_freq<map>,1,1,"2012-12-03")-->1

[0516] (root.retweet_freq<map>,2,2,"2012-12-04")-->20

[0517] (root.retweet_freq<map>,2,2,"2012-12-05")-->1

[0518] 类似地,VI将具有以下条目(如果所有路径被索引并且映射被像映射一样处理):

```
(root.retweet_freq<map>, 1)      --> 2, 2, "2012-12-05"
(root.retweet_freq<map>, 1)      --> 1, 1, "2012-12-03"
```

[0519]

```
(root.retweet_freq<map>, 10)     --> 1, 1, "2012-12-01"
(root.retweet_freq<map>, 13)     --> 1, 1, "2012-12-02"
(root.retweet_freq<map>, 20)     --> 2, 2, "2012-12-04"
(root.tags<arr>, "cupcakes")     --> 1, 1, 1
(root.tags<arr>, "muffins")      --> 1, 1, 0
(root.tags<arr>, "sushi")        --> 2, 2, 0
(root.tags<arr>, "umami")        --> 2, 2, 1
(root.text<str>, "Love #muffins and #cupcakes")  --> 1
(root.text<str>, "Love #sushi and #umami")       --> 2
(root.user.id, 29471497)                         --> 1
(root.user.id, 28492838)                         --> 2
(root.user.screen_name, "mashah08")              --> 1
(root.user.screen_name, "binkert")               --> 2
```

[0520] 虽然分阶段描述以上动作,但以上动作可被管线化以允许在单个遍次中执行摄取,加载BI、AI和RI,并且计算JSON模式。可异步构建其它索引,并且可根据需要启用和停用其它索引。

[0521] 系统架构

[0522] 分析平台被建造为面向服务的。在各种实施方案中,存在五项主要服务:代理、元数据服务、查询执行器、存储服务和摄取服务。

[0523] 这种分离方法可具有若干优点。由于这些服务仅通过外部API(远程程序调用)进行通信,所以可对服务进行多路复用并且独立共享每个服务。举例来说,可每个执行器使用多个代理并且每个存储服务使用多个执行器。还可在执行器和存储服务的多个实例上共享元数据服务。

[0524] 执行器、存储和摄取服务被并行化,并且可在专用或公用基础设施中在虚拟化机器实例中运行单独片段。这允许独立地中止并缩放这些服务。这对于通过基于需求波动调整服务能力来降低成本为有用的。举例来说,公用云的弹性可用以高度并行化摄取服务来实现快速夜间加载,同时针对日常查询工作负荷保持执行和存储服务在大小上减小。

[0525] 代理是通往客户端的网关并且支持一个或多个标准协议,例如ODBC(开放式数据库互连)、libpq、JDBC(Java数据库互连)、SSL(安全套接字层)等。网关充当内部服务的防火墙、验证服务和控制点。举例来说,客户端连接(例如网络套接字)可在代理处保持打开,而

中止支持的执行和存储服务以节省成本。当客户端连接再次变得有效时，可按照需要以相对较短的启动等待时间唤醒所需要的服务。

[0526] 元数据服务通常由其它服务的许多实例共享。其存储元数据，包括模式、源信息、分割信息、客户端用户名、键、统计信息(直方图、值分布等)以及关于每个服务的当前状态的信息(实例数目、IP地址等)。

[0527] 存储服务管理索引并且服务读取和写入请求。另外，查询执行器可将许多函数下推到存储服务。在各种实施方案中，存储服务可评估判断和UDF(用户自定义函数)以过滤结果，评估本地联接(例如，以重构对象)，评估下推联接(例如，广播联接)，并且评估本地聚合。

[0528] 存储服务可通过称为分割并行的技术来并行化。在这种方法中，创建存储服务的许多实例或分区并且在分区当中划分所摄取的对象。每个分区存储每个类型的索引，就像它是单个整体实例一样。然而，每个分区仅对所摄取的数据的子集进行索引。

[0529] 分析引擎支持一个或多个分割策略。简单但有效的策略是通过tid来分割对象并且将其相应条目存储在本地索引中。以此方式，不在不同实例上分裂所摄取的对象，当查询依赖于对象的多个部分时，所述分裂可能会消耗显著的网络带宽。可按多种方式指派tid，包括散裂指派、轮循或基于范围的指派。这些特定指派在所有实例上分布最新近的数据，进而分摊负荷。

[0530] 另一种策略是通过另一字段值(或字段值组合)(例如用户id或会话id)来分割。替代分割字段(列)使得便于与其它表格或集合(例如，用户简档)执行本地联接。分割策略可为散列分割或使用取样和范围分割。前者用于有效点查找，并且后者用于支持有效范围搜索。

[0531] 不管分割策略如何，应当能够本地重构对象或对象的任何子集。因此，存储服务分区在查询处理期间没有串话，并且仅需要经由其API来与执行服务通信。

[0532] 存储服务具有高速缓存。可增加每个分区中的高速缓存大小或增加分区数目来改善存储服务的性能。存储服务可将索引高速缓存在存储器中或本地磁盘上，并且索引可驻存于如Amazon S3等外部存储装置上。这种特征允许关闭和毁坏存储服务节点并且在必要时对其进行重新部署。此外，其允许极端弹性：以低成本使存储服务休眠到S3并且随着需求波动而改变存储服务能力的能力。

[0533] 查询执行服务执行由查询规划阶段生成的查询计划。其实施查询运算符，例如联接、统一、分类、聚合等。这些运算中的许多运算可被下推到存储服务，并且在可能时这样做。这些包括判断、投影、用以重构所投影的属性的列式联接以及使用分组依据声明的分配和代数聚合函数的部分聚合。

[0534] 查询执行服务接受来自存储服务的数据并且计算非本地运算：非本地联接、需要重新分割的分组依据声明、分类等。所述执行器类似于分割的并行执行器。其使用交换运算符来在查询执行步骤之间重新分割，并且采用本地存储来溢出中间结果。对于许多查询，有可能在存储服务中运行大多数查询并且仅需要单个执行器节点来收集结果并执行任何小型非本地运算。

[0535] 摄取服务

[0536] 摄取服务负责将半结构化数据加载到存储服务中，在存储服务处可查询所述数

据。用户提供来自多种平台(例如,MongoDB、Amazon S3、HDFS)的呈多种格式(例如,JSON、BSON、CSV)的数据,所述数据任选地用压缩机制(例如,GZIP、BZIP2、Snappy)进行压缩。基本程序都适用而不管所使用的格式如何。

[0537] 摄取任务可粗略地划分为两个部分:初始摄取任务,其加载大量新用户数据;以及增量摄取,其在新数据可用时周期性地发生。

[0538] 初始摄取

[0539] 初始摄取过程可被分解为若干个步骤。首先,将输入数据分割为多个数据块。用户在文件集合中或通过提供通往其数据源的直接连接来提供初始数据。在元数据服务中记录这些文件的位置和格式。用户可提供已经例如由于日志文件旋转而被分割的数据,但是如果没有,那么可将文件分割为多个数据块以支持并行加载。这些数据块通常为大约几百兆字节并且独立进行处理。

[0540] 用于分割输入文件的确切机制取决于数据格式。对于由换行符分离记录的未压缩格式(例如,JSON或CSV),可使用数目等于目标数据块数目的过程来并行处理单个文件。处理在文件中的恰当偏移量(file_size/total_num_chunks)*chunk_num处开始,并且接着进行搜索,直至找到换行符为止。对于压缩数据或呈像BSON等二进制格式的数据,可能需要依序扫描每个输入文件。将每个数据块的位置(文件,偏离量,大小)存储在元数据服务中。

[0541] 一旦将数据划分为多个数据块,就执行实际模式推断和摄取。作为这个过程的一部分,启动两个服务:摄取服务和存储服务。这两个服务可采用多个服务器来工作。所述两个服务还可共同位于任何给定机器上。摄取服务是短暂的并且仅在摄取过程期间使用,而存储服务保持实际数据并且必须是持久的。摄取中所涉及的服务器将数据发送到存储服务服务器,并且摄取服务器的数目独立于存储服务器的数目,其中选择数目来使每个服务的吞吐量之间的不平衡减到最小。在摄取服务器之间分割数据块。每个摄取服务器负责用于指派给它的每个数据块的以下步骤:(i)剖析和类型推断,(ii)与存储服务通信,以及(iii)计算局部模式和统计信息。

[0542] 首先,将数据记录剖析成内部树表示。可针对所有源格式(JSON、BSON等)使用一致的内部表示。取决于输入格式,还可执行类型推断。举例来说,JSON没有日期的表示,所以通常将日期存储为字符串。由于日期是非常常见的,所以其是对在摄取期间检测的类型的例子,使得用户可利用日期发布询问。对于CSV输入文件,由于没有对列进行定型,所以必须还检测例如整数等基本类型。

[0543] 一旦已经对记录进行剖析并且推断出类型,就将剖析树的压缩表示发送到存储服务。这采用树的前序遍历的形式。存储服务负责确定待存储在每个索引(BI、AI等)中的值,并且产生元组id和联接键。键产生被提交给存储服务,使得可依序产生键,这对底层索引存储区改善摄取性能。

[0544] 在摄取记录时,使用上述规则更新局部JSON模式。所述模式将反映单个摄取机器所看到的记录,并且不同机器可具有不同模式。

[0545] 除了计算模式之外,维持统计信息,所述统计信息对查询处理以及识别映射有用。这些统计信息包括像每个属性出现的次数以及其以字节数计的平均大小等量度。举例来说,以下记录

[0546] {id:3546732984}

[0547] {id:"3487234234"}

[0548] {id:73242342343}

[0549] {id:458527434332}

[0550] {id:"2342342343"}

[0551] 将产生模式{id:int,id:string},并且可用计数3标注id:int并且用计数2标注id:string。每个摄取机器存储其在元数据服务中计算出的模式和统计信息。

[0552] 一旦已经摄取所有数据块,就计算总体模式,其将由查询引擎使用并且呈现给用户。这可使用单个过程来完成,所述单个过程从元数据服务读取部分模式,使用上述方法对其进行合并,并且将结果存储回元数据服务中。由于模式的数目限于摄取机器的数目,所以这个过程不是性能关键的。

[0553] 确定映射是任选的。如先前所描述,可连同存储在元数据服务中的统计信息一起使用试探法来确定应当在MI中将哪些属性存储为映射。记得这对查询处理来说是不必要的,但这使得一些查询表达起来较自然并且改善效率。一旦已经识别了映射,每个存储服务器就接收识别哪些属性应当是映射的消息。存储服务器接着扫描这些列并且将它们插入到MI中。

[0554] 增量更新

[0555] 一些用户可预先加载其大部分数据,但大多数用户将随时间周期性加载新数据,通常作为规则(例如,每小时或每天)过程的一部分。摄取这个数据很大程度上类似于初始摄取。将新数据分裂成数据块,针对每个数据块计算模式,并且将局部模式与元数据服务中所维持的全局模式合并。

[0556] 在添加新数据时,系统自动地检测新数据。方法取决于源数据平台。举例来说,对于S3文件,最简单的情况是检测S3桶中的变化。特殊过程周期性地扫描桶以查找新键-值对(即,新文件),并且将所找到的任何键-值对添加到元数据服务。在已经找到某个数目的新文件或已经过去某个时间段之后,过程启动新摄取过程来加载所述数据。

[0557] 在MongoDB中执行的操作可存储在称为操作日志(或oplog)的特殊集合中。操作日志提供在内部用于复制的由MongoDB使用的写入操作的一致记录。读取操作日志并且将其用于在存储新记录的S3中创建一组平面文件。以上方法可接着用以摄取新数据。

[0558] 增量摄取过程可处理新数据(例如,新JSON文件)以及对现有文献的更新(例如,现有JSON文献中的新属性或现有属性的新值)。每个数据源平台在暴露源文件中的更新的方面具有不同能力。将这种类型的信息称为“德耳塔”,并且其可采用平面文件或日志文件(例如,MongoDB)的形式。增量摄取过程处理来自“德耳塔”文件的信息,并且将所述信息与现有模式信息进行组合来产生发送到存储服务的新数据。

[0559] 构造数据子集

[0560] 尽管此处描述的用于摄取数据并且进行增量更新的系统可从源摄取全部数据,但通过预先指定想要摄取的数据的JSON模式(或关系模式),仅摄取子集是相对简单的。这通过提供JSON模式本身或通过提供指定子集的查询来进行。以此方式,分析平台可被认为是源数据的物化视图。

[0561] 还有可能指定用户不想摄取的数据。可提供JSON模式或关系模式,其描述不应摄取的数据部分。接着,只不过是在元数据服务中记录那个信息,其告诉摄取过程简单地跳过

所有未来行的那些元素。如果这在已经摄取了数据之后进行，那么已经摄取的数据简单地变为不可用并且可成为由后台任务收集的垃圾。这个垃圾收集过程将被并入到索引存储区(例如，LevelDB)的压缩过程中。

[0562] 容错性

[0563] 尽管有可能在初始摄取期间重新开始加载过程，但增量摄取过程不应破坏系统中的现有数据，以便防止用户必须从头开始重新加载所有数据。由于摄取文件不是幂等操作，所以归因于id生成，可基于获取底层存储系统的快照来实施简单容错方案。

[0564] 当底层存储系统支持在一个时间点获取一致快照时，获取快照可为简单的，正如LevelDB那样。使用这个基元，用于递增加载的步骤如下。单个过程指导每个存储服务器在本地获取快照并且在加载的持续时间内将所有查询指向这个快照。如上所述加载每个数据块。当完成时，负责加载数据块的摄取服务器将其标记为在元数据服务中完成。

[0565] 一个过程监视元数据服务。当已经加载所有数据块时，其自动将查询重新指向状态的更新版本。接着可丢弃在第一个步骤中获取的快照。在失败的情况下，快照成为状态的规范版本，并且丢弃状态的部分经更新的(并且可能被破坏的)原始版本。接着重新开始摄取过程。另外，可在服务器发生故障的情况下使用存储系统磁盘卷的快照来进行恢复。

[0566] 查询执行

[0567] 示例性查询

[0568] 为了展示示例性执行，考虑以下简单查询：

[0569] select count(*)from table as t where t.a＞10;

[0570] 首先，代理接收查询并且将其发送到执行器节点以用于规划。接下来，执行器节点创建查询计划，其调用元数据服务来确定哪些集合和存储节点可供使用。所述执行器节点通常将所述计划分配给其它执行器节点，但此处，仅需要单个执行器节点。

[0571] 执行节点接着向存储服务节点发出RPC调用，下推t.a＞10判断和计数函数。接下来，存储节点计算子计数并且将其返回到执行器节点。执行器节点接着在代理取下一个结果值时将结果返回到代理。

[0572] 动态定型

[0573] 数据库系统(例如，PostgreSQL)的存储引擎是强定型的，这意味着列(或属性)的所有值必须具有完全相同的类型(例如，整数、字符串、时戳等)。在大数据分析的上下文中，这是一个重要限制，因为经常应用程序需要改变一条特定信息(属性)的表示以及因而他们使用来存储所述信息(属性)的数据类型。举例来说，应用程序可最初使用整数来存储特定属性的值并且接着切换到使用浮点数。数据库系统未被设计来支持此类操作。

[0574] 一种处理这个问题的方式是针对每个属性使用多个关系列-每个不同数据类型使用一个。举例来说，如果已经看到具有值31432和"31433"(即，整数和字符串)的属性"user.id"，那么可将"user.id<int>"和"user.id<string>"存储为单独列。单个记录将具有仅针对这些列中的对应于那个记录中的"user.id"的类型的一个列的值。针对那个记录的其它列的值将为空值。

[0575] 针对同一属性呈现多个列通常对于用户使用来说太复杂。为了简化用户体验，分析平台可在查询时动态地推断用户期望使用的类型。为此，存储服务记录多个类型。举例来说，存储服务针对数字使用一般数据类型，称为“数字”，其涵盖整数和浮点数。当使用“数

字”类型时，将较具体的数据类型存储为值的一部分。举例来说，将属性"Customer.metric"的整数值10在BI中存储为键-值对，其中(Customer.metric,<NUMBER>,tid)是键并且(10,INTEGER)是值。同一属性的浮点值10.5将被存储为键：(Customer.metric,<NUMBER>,TID)、值：(10.5,FLOAT)。

[0576] 最后，在查询时，分析平台可根据查询的性质(判断、挑选操作等)在数据类型之间执行动态挑选，只要这些操作不导致信息损耗。虽然“数字”不是ANSI SQL类型，但灵活的定型系统允许客户端根据查询上下文将其处理为标准ANSI SQL浮点数、整数或数字类型。举例来说，考虑以下查询：

[0577] select user.lang from tweets where user.id＝'31432'

[0578] 在具有"user.id<int>"和"user.id<string>"两者的情况下，系统在查询时任选地将整数(例如，31432)转换为单个字符串表示(例如，"31432")，进而允许用户对具有ANSI SQL类型VARCHAR的单个列"user.id"起作用。

[0579] 虽然提到ANSI(美国国家标准协会)/ISO(国际标准化组织)SQL:2003作为实施例，但在其它实施方案中，可遵守其它标准，即SQL或其它。仅举例来说，暴露的接口可符合ANSI/ISO SQL:2011。

[0580] 图式

[0581] 在图1A中，示出分析平台的示例性基于云的实施方案。使用分析框架的组织的局域网(LAN)或广域网(WAN)100连接到互联网104。这个实施方案中的计算需求和存储需求均由基于云的服务提供。在所示出的特定实施方案中，计算服务器与存储服务器分离。具体地说，计算云108包括多个服务器112，其提供分析框架的处理能力。服务器112可为离散硬件实例或可为虚拟化服务器。

[0582] 服务器112还可具有其自己的存储装置，处理能力对所述存储装置进行操作。举例来说，服务器112可实施查询执行器和存储服务两者。尽管传统的列式存储系统将数据作为列存储在磁盘上，但当将所述数据读取到存储器中时，从列式数据重组行。然而，本公开的索引在磁盘上和在存储器中均作为列式存储进行操作。由于索引的独特配置，可用相对较少的损失来实现快速列式访问的利益。

[0583] 存储云116包括用于索引数据的存储阵列120，因为根据本公开，将数据存储在索引中而非存储在物化表格中。当使用服务器112的存储资源时，存储阵列120可用于备份和近线存储，而非用于对每个查询做出响应。

[0584] 在各种实施方案中，存储阵列124可包括数据，分析框架将对所述数据进行操作。仅举例来说，存储阵列124可保持相关数据，例如日志数据，用户可能想要使用分析框架来查询所述数据。虽然存储阵列120和存储阵列124被示出为在同一存储云116中，但其可位于不同云中，包括专用外部托管云、公用云和组织特定的内部托管虚拟化环境。

[0585] 仅举例来说，存储云116可为Amazon网络服务(AWS)S3云，商家已经正在使用所述云来在存储阵列124中存储数据。因而，将数据传送到存储阵列120中可用高吞吐量和低成本来实现。计算云108可由AWS EC2提供，在所述情况下，计算云108和存储云116由共同提供商托管。用户130使用标准SQL工具构造查询，在计算云108中运行那个查询，并且向用户130返回响应。SQL工具可为已经安装在用户130的计算机134上的工具，并且不必为了与本分析框架一起工作来进行修改。

[0586] 在图1B中，示出另一种示例性部署方法。在这种情况下，物理服务器设备180连接到商家的LAN/WAN 100。服务器设备180可现场进行托管或可异地进行托管，并且例如使用虚拟专用网络，连接到LAN/WAN 100。服务器设备180包括计算能力以及存储装置，并且从源接收输入数据，所述源可在LAN/WAN 100本地。仅举例来说，计算机或服务器184可存储日志，例如网络流量日志或入侵检测日志。

[0587] 服务器设备180检索并存储索引数据以用于对用户130的查询做出响应。存储云116可包括额外数据源188，其可保持另外其它数据且/或可为用于较旧数据的近线数据存储设施。为了满足用户查询，服务器设备180可从额外数据源188检索额外数据。例如出于备份目的，服务器设备180还可将数据存储在存储云116中。在各种其它实施方案中，额外数据源188可为云中的Hadoop实施方案的一部分。

[0588] 本公开的分析框架是灵活的，使得许多其它部署情形是有可能的。仅举例来说，可向商家提供软件，并且可将那个软件安装在自己的或托管的服务器上。在另一实施方案中，可提供虚拟机实例，其可通过虚拟化环境来例示。此外，用户可在浏览器中具备用户接口，并且SQL部分可由服务提供商(例如Nou Data)托管，并且在其系统上或在云中实施。

[0589] 在图1C中，示出服务器200的硬件部件。处理器204执行来自存储器208的指令，并且可对存储在存储器208中的数据进行操作(读取和写入)。一般来说，为了速度，存储器208是易失性存储器。潜在地经由芯片组212，处理器204与非易失性存储装置216通信。在各种实施方案中，非易失性存储装置216可包括充当高速缓存的快闪存储器。容量较大且成本较低的存储装置可用于二级非易失性存储装置220。举例来说，磁性存储介质(例如硬盘驱动器)可用以将底层数据存储在二级非易失性存储装置220中，所述二级非易失性存储装置的有效部分高速缓存在非易失性存储装置216中。

[0590] 输入/输出功能性224可包括例如键盘和鼠标等输入以及例如图形显示器和音频输出等输出。服务器200使用连网卡228与其它计算装置通信。在各种实施方案中或在各种时间处，输入/输出功能性224可休眠，其中服务器200与外部行动者之间的所有交互是经由连网卡228进行的。为了易于说明，未示出额外众所周知的特征和变化，例如(仅举例来说)非易失性存储装置216与存储器208之间或连网卡228与存储器208之间的直接存储器访问(DMA)功能性。

[0591] 在图2A中，过程图示出如何将数据摄取到分析框架中以使得其可由用户130查询的一个实施例。数据源300提供数据，分析框架对所述数据进行操作。如果那个原始数据不是自描述性的，那么任选的用户自定义包装器函数304可将原始数据转换为自描述性半结构化数据，例如JSON对象。

[0592] 管理员308(其可为以不同能力进行操作的用户130)能够指明用于实施这些包装器函数的指导方针。管理员308还可指明将使用哪些数据源300以及将从那些数据源检索什么数据。在各种实施方案中，检索数据可包括取子集操作和/或其它计算。仅举例来说，当一个数据源300是Hadoop时，可在针对分析框架检索数据之前请求MapReduce工作。

[0593] 所检索的数据由模式推断模块312进行处理，所述模式推断模块基于所接收数据的所观测结构来动态地构造模式。在各种实施方案中，管理员308可具有向模式推断模块312提供定型暗示的能力。举例来说，定型暗示可包括对辨认特定格式(例如日期、时间或其它管理员自定义类型)的请求，所述格式可由(例如)正规表达式指定。

[0594] 将数据对象和由模式推断模块312产生的模式提供到修饰模块316以及索引创建模块320。输入对象包括源数据以及描述所述源数据的元数据。由索引创建模块320将源数据存储在索引存储装置324中。

[0595] 修饰模块316在由模式模块312产生的模式中识别映射。在不需要映射识别的实施方案中，可省略修饰模块316。管理员308可能够指定映射标准来调整在识别映射的过程中使用的由修饰模块316执行的试探法。

[0596] 在已经识别了映射之后，关系模式创建模块328产生关系模式，例如顺应SQL的模式。另外，将所识别的映射提供给辅助索引创建模块332，其能够创建额外索引，例如MapIndex，以及ValueIndex中的映射条目，如上所述。辅助索引还可存储在索引存储装置324中。

[0597] 管理员308可具有请求创建映射索引的能力，并且可指定将哪个列添加到值索引。另外，管理员可能够指定应当将哪些对象处理为映射，并且可动态地改变是否将对象处理为映射。此类改变将导致关系模式的改变。

[0598] 关系优化模块336优化关系模式以向用户130呈现较简洁的模式。举例来说，关系优化模块336可识别表格之间的一对一关系，并且将那些表格展平为单个表格，如上所述。将所得的关系模式提供给元数据服务340。

[0599] 查询执行器344与元数据服务340介接以执行来自代理348的查询。代理348与顺应SQL的客户端(例如ODBC客户端352)交互，所述客户端在没有特殊配置的情况下能够与代理348交互。用户130使用ODBC客户端352来将查询发送到查询执行器344并且接收对那些查询的响应。

[0600] 经由ODBC客户端352，用户130还可看到由元数据服务340存储的关系模式并且在所述关系模式上构造查询。不要求用户130或管理员308知道预期模式或帮助创建模式。代替地，基于所检索到的数据来动态地创建模式，并且接着呈现所述模式。虽然示出ODBC客户端352，但除了ODBC之外的机构也是可用的，包括JDBC和直接postgres查询。在各种实施方案中，图形用户接口应用程序可有助于方便用户使用ODBC客户端352。

[0601] 查询执行器344对来自存储服务356的数据进行操作，所述存储服务包括索引存储装置324。存储服务356可包括其自己的本地存储处理模块360，查询执行器344可将各种处理任务委派给所述本地存储处理模块360。接着由存储处理模块360将所处理的数据提供给查询执行器344以构造对所接收的查询的响应。在基于云的实施方案中，存储服务356和查询执行器344均可在计算云中实施，并且索引存储装置324可存储在计算实例中。索引存储装置324可镜射到近线存储装置，例如在如图1A所示的存储云116中。

[0602] 在图2B中，高级功能图示出具有多个节点402-1、402-2和402-3(统称为节点402)的存储服务356。虽然示出三个节点402，但可使用更多或更少节点，并且所使用的数目可基于分析框架的需要而动态地变化。节点402的数目可随着需要存储更多数据以及响应于需要额外处理来执行查询且/或提供冗余性而增加。查询执行器344被示出为具有节点406-1、406-2和406-3(统称为节点406)。节点406的数目也可基于查询负荷来动态地变化，并且独立于节点402的数目。

[0603] 代理348提供ODBC客户端352与查询执行器344之间的接口。查询执行器344与元数据服务340交互，元数据服务340存储用于驻存在存储服务356中的数据的模式。

[0604] 图3示出用于数据摄取的示例性过程。控制在504处开始，在该处可例如由用户或管理员指明数据源。另外，可选择来自数据源的某些数据集，并且可请求对数据源进行某些取子集和化简操作。控制在508处继续，在该处监视所指明的数据源以查找新数据。

[0605] 在512处，如果已经将新数据对象添加到数据源，那么控制转移到516；否则，控制返回到504，以允许在需要时修改数据源。在516处，推断新对象的模式，这可根据定型函数来执行，例如图4中所示。在520处，将来自516的所推断出的模式与已经存在的模式合并。所述合并可根据合并函数来执行，例如图5中所示。

[0606] 在524处，如果需要修饰，那么控制转移到528；否则，控制转移到532。在528处，在数据内识别映射，例如图8中所示。在536处，如果未识别到新映射，那么控制在532处继续；否则，如果已经识别到新映射，那么控制转移到540。在540处，如果需要映射索引，那么控制转移到544；否则，控制在532处继续。在544处，对于BigIndex或ArrayIndex中的与新映射属性相关联的每个值，将那个值添加到映射索引。另外，如果用户和/或管理员需要，那么针对特定属性，将值添加到值索引。控制接着在532处继续。

[0607] 在各种实施方案中，在524处的修饰可进行等待，直到处理了第一轮对象为止。举例来说，在初始摄取时，可延迟修饰，直到摄取了所有初始对象为止。以此方式，将已经收集到足够的统计信息来由映射试探法使用。对于额外对象的增量摄取，可在每组新的额外对象之后执行修饰。

[0608] 在532处，如果JSON模式已经由于新对象而改变，那么控制转移到548，在该处将所述模式转换为关系模式。控制在552处继续，在该处对关系视图进行优化，例如通过展平一对一关系。控制接着在556处继续。如果模式尚未在532处改变，那么控制将直接转移到556。在556处，用新对象的数据填充索引，这可如图7所示来执行。控制接着返回到504。

[0609] 虽然在556处将索引的填充示出为在548处将所推断出的模式转换为关系模式之后执行，但在各种实施方案中，可在产生关系模式之前填充索引，因为不需要关系模式。所述程序可使用所推断出的JSON模式来产生路径和联接键。关系模式充当底层半结构化数据的关系视图。

[0610] 图4示出依赖于递归的定型函数的示例性实施方案。控制在604处开始，在该处如果待定型的对象是标量，那么控制转移到608。在608处，确定标量的类型，并且在612处返回那个标量类型作为函数的输出。可基于所接收的对象中的自描述来确定标量类型。另外，可使用另外的定型规则，其可承认某些字符串表示例如日期或时间等数据。

[0611] 如果在604处对象不是标量，那么控制转移到616。在616处，如果对象是数组，那么控制转移到620，在该处对数组的每个元素递归地调用定型函数(图4)。在已经接收到这些定型函数的结果时，控制在624处继续，在该处对在620处所确定的元素类型的数组调用例如图6所示的折叠函数。当由折叠函数返回折叠数组时，在628处由定型函数返回那个折叠数组。

[0612] 如果在616处对象不是数组，那么控制转移到632。在632处，对对象的每个字段递归地调用定型函数(图4)。控制在636处继续，在该处对在632处确定的字段类型的并置调用折叠函数。接着在640处由定型函数返回由折叠函数返回的折叠对象。

[0613] 图5示出将两个模式元素合并为单个模式元素的合并函数的示例性实施方案。合并函数也是递归的，并且在第一次调用时，两个模式元素是先前存在的模式和从新近接收

的对象推断出的新模式。在合并函数的进一步递归调用中，模式元素将是这些模式的子元素。控制在704处开始，在该处如果待合并的模式元素是等效的，那么控制转移到708并且返回所述等效模式元素中的任一个。否则，控制转移到712，在该处如果待合并的模式元素均为数组，那么控制转移到716；否则，控制转移到720。

[0614] 在716处，如果待合并的数组之一是空的，那么在724处返回另一个数组。否则，控制在728处继续，在该处对含有两个待合并的数组的元素的数组调用如图6中所示的折叠函数。接着在732处由合并函数返回由折叠函数返回的折叠数组。

[0615] 在720处，如果待合并的模式元素之一是空的，那么在736处由合并函数返回另一个模式元素。如果待合并的模式元素都不是空的，那么控制在740处继续，在该处对含有两个待合并的模式元素的键-值对的对象调用折叠函数。接着在744处合并函数返回由折叠函数返回的折叠对象。

[0616] 图6示出折叠函数的示例性实施方案。控制在804处开始，在该处如果待折叠的对象是数组，那么控制转移到808；否则，控制转移到812。在808处，如果数组含有均为数组的一对值，那么控制转移到816；否则，控制在820处继续。在820处，如果数组含有均为对象的一对值，那么控制转移到816；否则，控制在824处继续。在824处，如果数组含有为相等标量类型的一对值，那么控制转移到816；否则，完成折叠并且从折叠函数返回数组。在816处，对由808、820或824识别的该对值调用例如图5中所示的合并函数。控制在828处继续，在该处用由合并函数返回的单个值替换该对值。

[0617] 在812处，如果对象中的任何键都是相同的，那么控制转移到832；否则，折叠完成并且返回对象。在832处，控制选择相同的一对键并且在836中继续。如果该对键的值均为数组或均为对象，那么控制转移到840；否则，控制转移到844。在840处，对该对键的值调用合并函数。控制在848处继续，在该处用具有由合并函数返回的值的单个键替换该对键。控制接着在852处继续，在该处如果任何额外键是相同的，那么控制转移到832；否则，折叠完成并且返回所修改的对象。在844处，如果该对键的值均为标量，那么控制转移到856；否则，控制转移到852。在856处，如果该对键的值的标量类型是相等的，那么控制转移到840来合并那几对键；否则，控制转移到852。

[0618] 图7示出用于用来自新近检索到的对象的数据填充索引的示例性过程。控制在904处开始，在该处如果需要RowIndex，那么控制转移到908；否则，控制转移到912。在908处，如上所述将对象添加到RowIndex，并且控制在912处继续。在912处，将对象展平为用于当前关系模式的关系元组，并且根据需要创建联接键。控制在916处继续，在该处控制确定是否存在待添加到索引的更多元组。如果是，那么控制转移到920；否则，索引已经被填充，并且因而控制结束。

[0619] 在920处，控制确定元组是否用于数组表格。如果是，那么控制转移到924；否则，控制转移到928。在924处，如果在数组表格中存在更多值列，那么控制转移到932。在932处，如果列值存在于原始检索对象中，那么在936处将值添加到ArrayIndex。控制接着在940处继续。如果针对所述列需要ValueIndex，那么控制转移到944；否则，控制返回到924。如果在932处列值不存在于原始检索对象中，那么控制返回到924。

[0620] 在928处，如果元组用于映射表格，那么控制转移到948；否则，控制转移到952。在948处，控制确定是否更多值列保留在映射表格中。如果是，那么控制转移到956；否则，控制

返回到916。在956处，控制确定列值是否存在于原始检索对象中。如果是，那么控制转移到960；否则，控制返回到948。在960处，将值添加到MapIndex，并且控制转移到964。在964处，如果针对列需要ValueIndex，那么在968中将值添加到ValueIndex；在任何情况下，控制接着返回到948。

[0621] 在952中，控制确定是否更多列存在于表格中。如果是，那么控制转移到972；否则，控制返回到916。在972处，控制确定列值是否存在于原始检索对象中。如果是，那么控制转移到976；否则，控制返回到952。在976处，将值添加到BigIndex并且控制在980处继续。在980处，如果针对所述列需要ValueIndex，那么控制转移到984，在该处将值添加到ValueIndex；在任何情况下，控制接着返回到952。

[0622] 图8示出用于识别映射的示例性过程。控制在1004处开始，在该处选择第一个对象。控制在1008处继续，在该处如果对象是空的，那么在1012处将包含对象指明为映射；否则，控制转移到1016。在1016处，控制如上所述确定平均字段频率与包含对象的频率的比率。控制在1020处继续，在该处如果比率低于阈值，那么控制转移到1012来将包含对象指明为映射；否则，控制转移到1024。仅举例来说，阈值可为用户可调整的，且/或可基于所观测的数据为动态的。在各种实施方案中，随着关系模式增大，可调整试探法来更容易地将字段识别为映射。在1012处，将包含对象指明为映射，并且控制在1024处继续。如果有更多对象要评估，那么控制转移到1028，在该处选择下一个对象，并且控制在1008处继续；否则，控制结束。

[0623] 图9示出依赖于递归来创建关系模式的create_schema函数的示例性实施方案。在调用create_schema函数时，控制将模式元素(Schema_Element)并入到表格(Current_Table)中。为此，控制在1104处开始，在该处如果Schema_Element是对象，那么控制转移到1108；否则，控制转移到1112。在1108处，如果对象是空对象，那么将对象视为映射并且控制转移到1116；否则，控制在1120处继续。在1120处，针对嵌套对象创建新表格(New_Table)。在1124处，将联接键(Join_Key)添加到Current_Table，并且在1128处，将对应的Join_Key添加到New_Table。控制接着在1132处继续，在该处针对嵌套对象中的每个字段，递归地调用create_schema函数来向表格添加字段。控制接着在1136处从create_schema函数的当前调用返回。

[0624] 在1112处，如果Schema_Element是映射，那么控制转移到1116；否则，控制转移到1138。在1116处，针对映射创建新表格(New_Table)。控制在1140处继续，在该处将Join_Key添加到Current_Table，并且在1144处，将对应的Join_Key添加到New_Table。在1148处，将具有字符串类型的键字段添加到New_Table。控制在1152处继续，在该处针对映射中的每个值类型，递归地调用create_schema函数来将值类型添加到New_Table。控制接着在1136处返回。

[0625] 在1138处，控制确定Schema_Element是否是数组。如果是，那么控制转移到1156；否则，控制转移到1160。在1156处，针对数组创建新表格(New_Table)，在1164处将Join_Key添加到Current_Table，并且在1168处将对应的Join_Key添加到New_Table。在1172处，将具有整数类型的索引字段添加到New_Table。控制在1176处继续，在该处针对数组中的每个项目类型，调用create_schema函数来将项目类型添加到New_Table。控制接着在1136处返回。

[0626] 在1160处，通过消元过程，Schema_Element是基元。如果在Current_Table中已经

存在具有与基元相同的名称的字段，那么控制转移到1180；否则，控制转移到1184。在1184处，简单地将名称字段添加到Current_Table，并且控制在1136处返回。在1180处，存在类型多态性，并且因此重命名Current_Table中的具有与基元相同的名称的现有字段以将其类型附加到字段名称。控制在1188处继续，在该处基于当前基元来添加新字段，其中类型附加到字段名称。控制接着在1136处返回。

[0627] 结论

[0628] 前述描述在本质上仅为说明性的，并且决不意图限制本公开、其应用或用途。本公开的广泛教示可按多种形式来实施。因此，尽管本公开包括特定实施例，但本公开的真实范围不应如此受限，因为在研究图式、说明书和所附权利要求书后，其它修改将变得显而易见。如本文中所使用，短语“A、B和C中的至少一个”应当理解为意指逻辑(A或B或C)，其中使用非排他性逻辑“或”。应当理解，一种方法内的一个或多个步骤可在不更改本公开的原理的情况下按不同次序(或同时)来执行。

[0629] 在本申请中，包括以下定义在内，术语“模块”可用术语“电路”替换。术语“模块”可指代以下各项、作为以下各项的一部分或包括以下各项：专用集成电路(ASIC)；数字、模拟或混合模拟/数字离散电路；数字、模拟或混合模拟/数字集成电路；组合逻辑电路；现场可编程门阵列(FPGA)；执行代码的处理器(共享、专用或群组)；存储由处理器执行的代码的存储器(共享、专用或群组)；提供所描述的功能性的其它合适的硬件组件；或以上各项的一些或全部的组合，例如呈芯片上系统的形式。

[0630] 如上文所使用的术语“代码”可包括软件、固件和/或微代码，并且可指代程序、例程、函数、类别和/或对象。术语“共享处理器”包含执行来自多个模块的一些或所有代码的单个处理器。术语“群组处理器”包含与额外处理器组合执行来自一个或多个模块的一些或所有代码的处理器。术语“共享存储器”包含存储来自多个模块的一些或所有代码的单个存储器。术语“群组存储器”包含与额外存储器组合存储来自一个或多个模块的一些或所有代码的存储器。术语“存储器”可为术语“计算机可读介质”的子集。术语“计算机可读介质”不包含传播穿过介质的暂时性电信号和电磁信号，并且因此可被视为有形且非暂时性的。非暂时性有形计算机可读介质的非限制性实施例包括非易失性存储器、易失性存储器、磁性存储装置和光学存储装置。

[0631] 本申请中所描述的设备和方法可部分地或全部地由一个或多个处理器执行的一个或多个计算机程序实施。计算机程序包括存储在至少一个非暂时性有形计算机可读介质上的处理器可执行指令。计算机程序还可包括且/或依赖于所存储的数据。

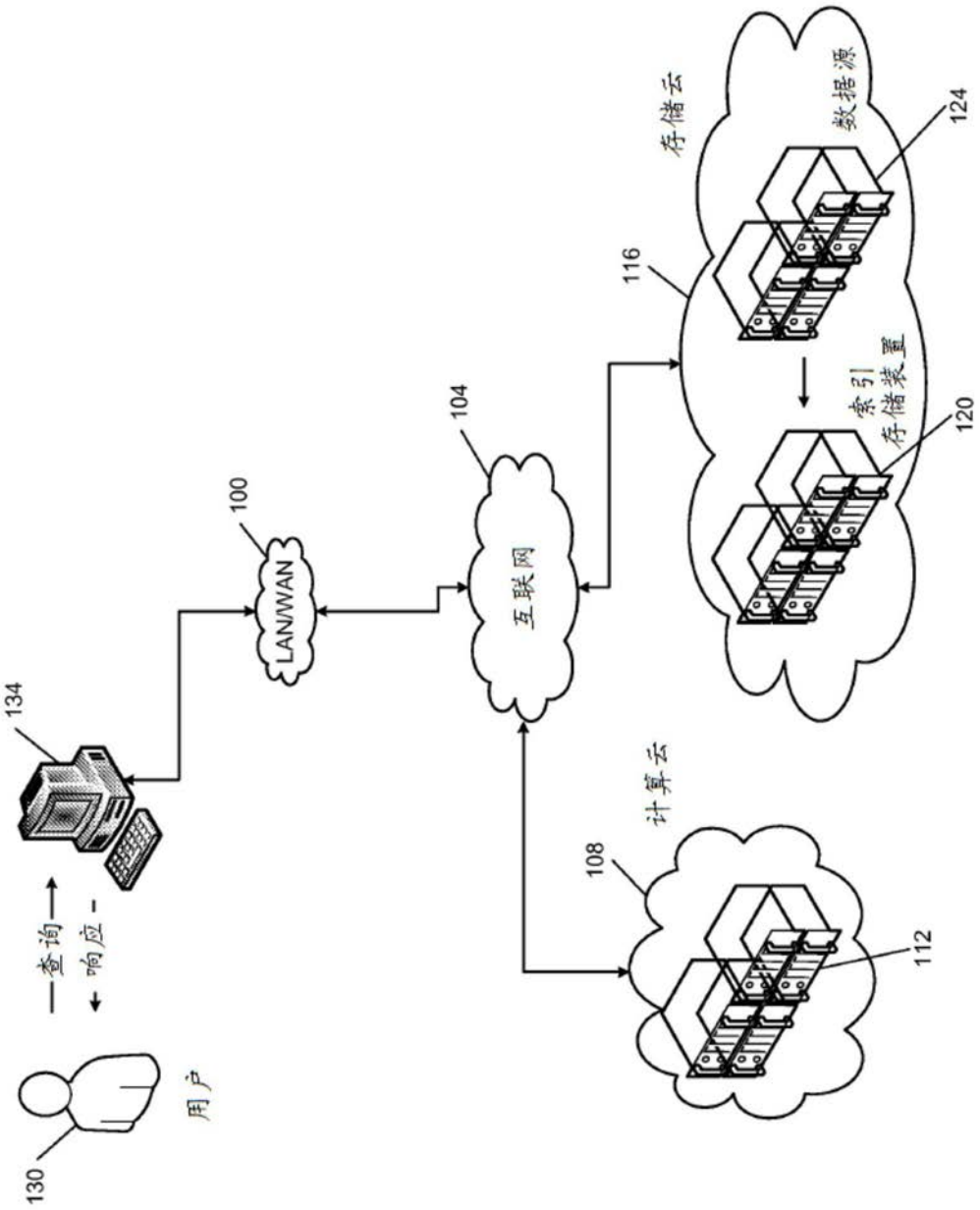

存储云
数据源
124
116
索引
存储装置
120
104
互联网
100
LAN/WAN
134
查询
响应
用户
130
计算云
108
112

图1A

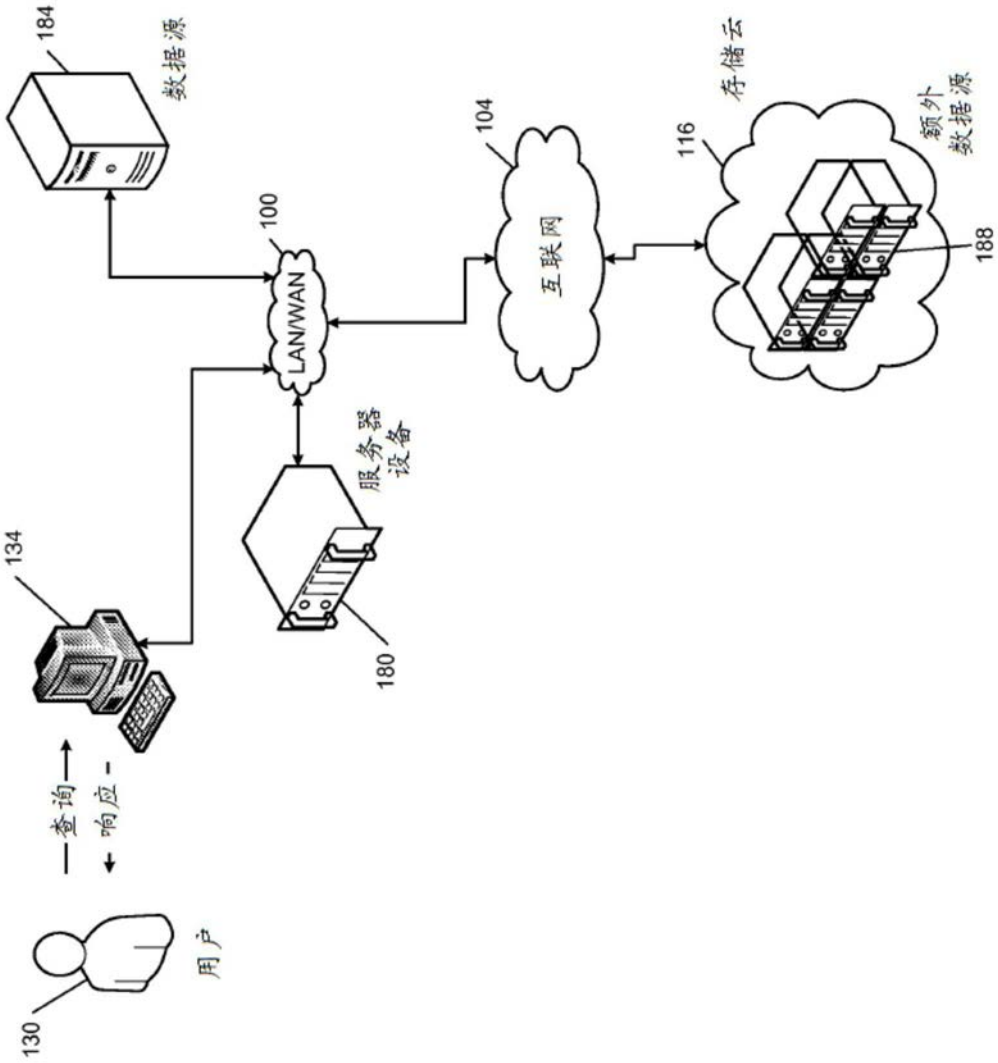

184
数据源
100
LAN/WAN
104
互联网
116
存储云
额外
数据源
188
服务器
设备
180
134
查询
响应
用户
130

图1B

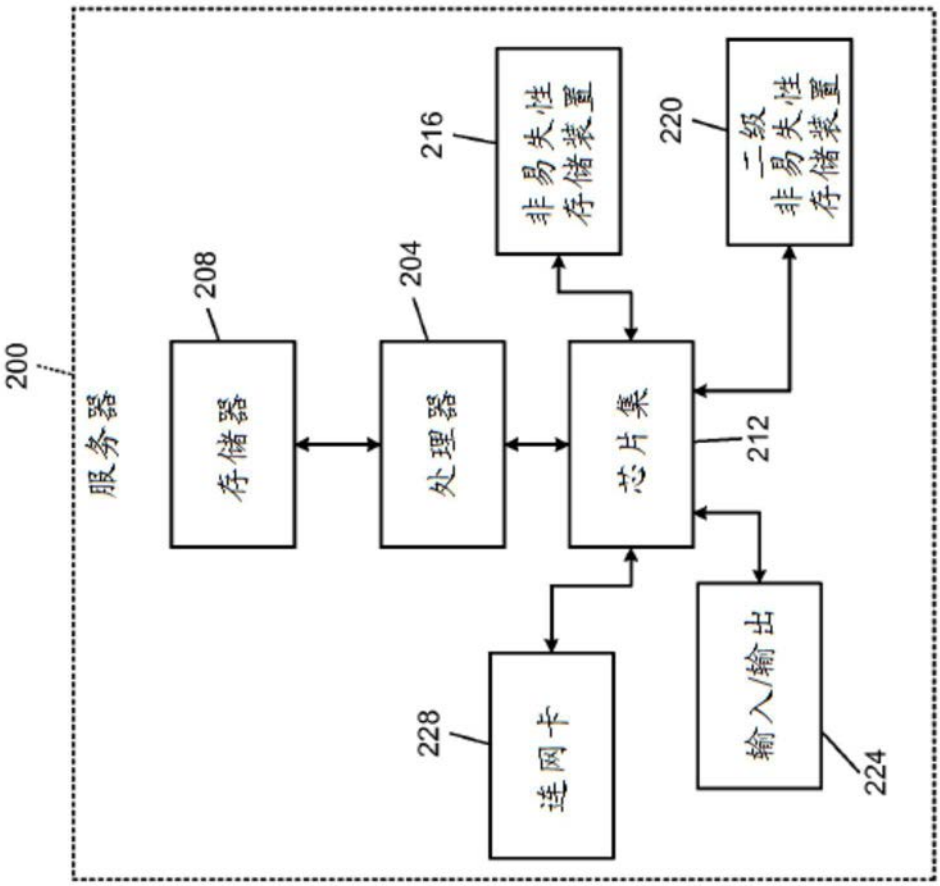

200
服务器
208
存储器
204
处理器
芯片集
212
216
非易失性存储装置
220
二级非易失性存储装置
228
连网卡
输入/输出
224

图1C

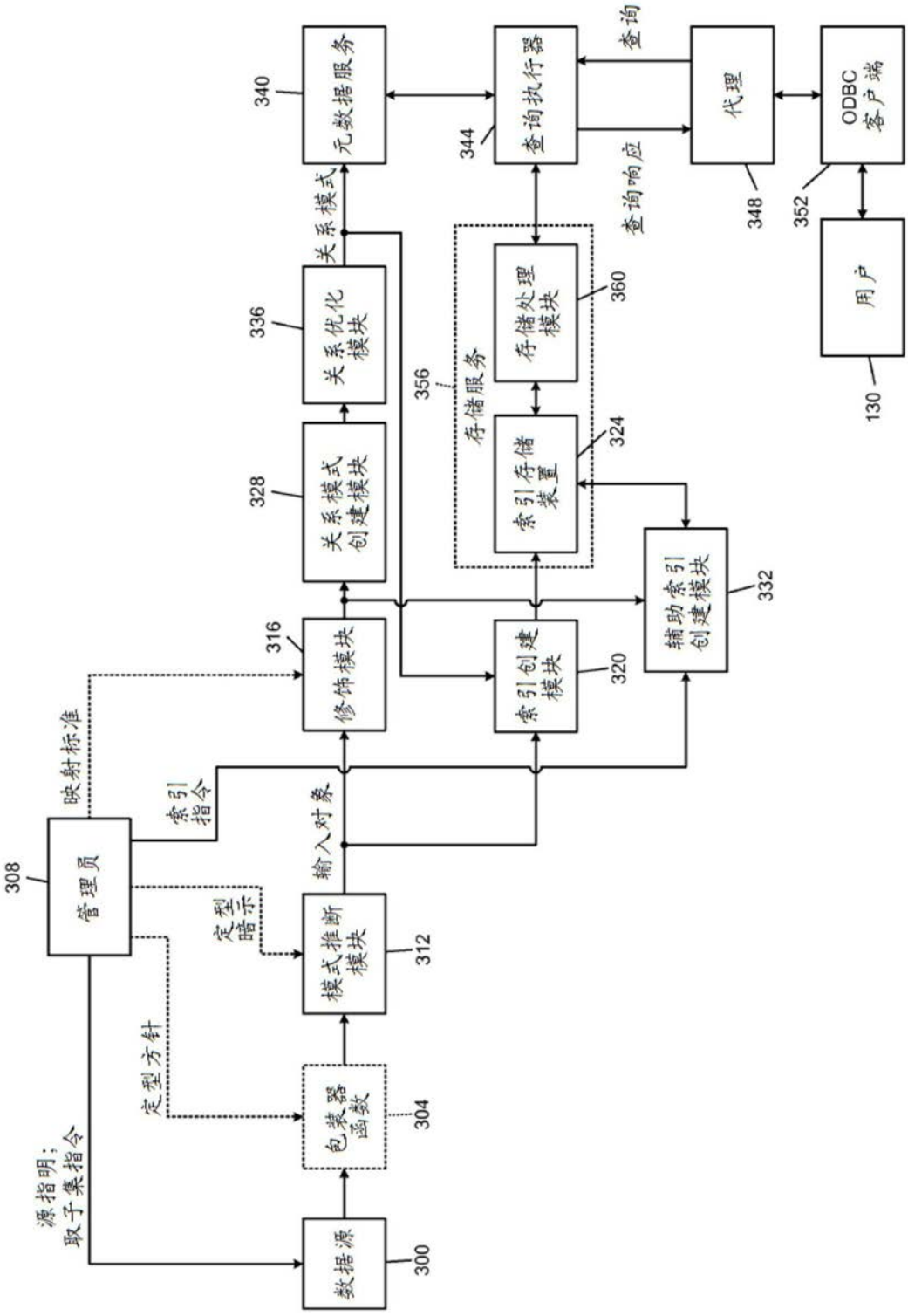

源指明；
取子集指令
数据源
300
包装器
函数
304
模式推断
模块
312
定型
暗示
定型方针
管理员
308
索引
指令
映射标准
输入对象
修饰模块
316
关系模式
创建模块
328
关系优化
模块
336
关系模式
元数据服务
340
索引创建
模块
320
辅助索引
创建模块
332
存储服务
356
索引存储
装置
324
存储处理
模块
360
查询执行器
344
查询
查询响应
代理
348
ODBC
客户端
352
用户
130

图2A

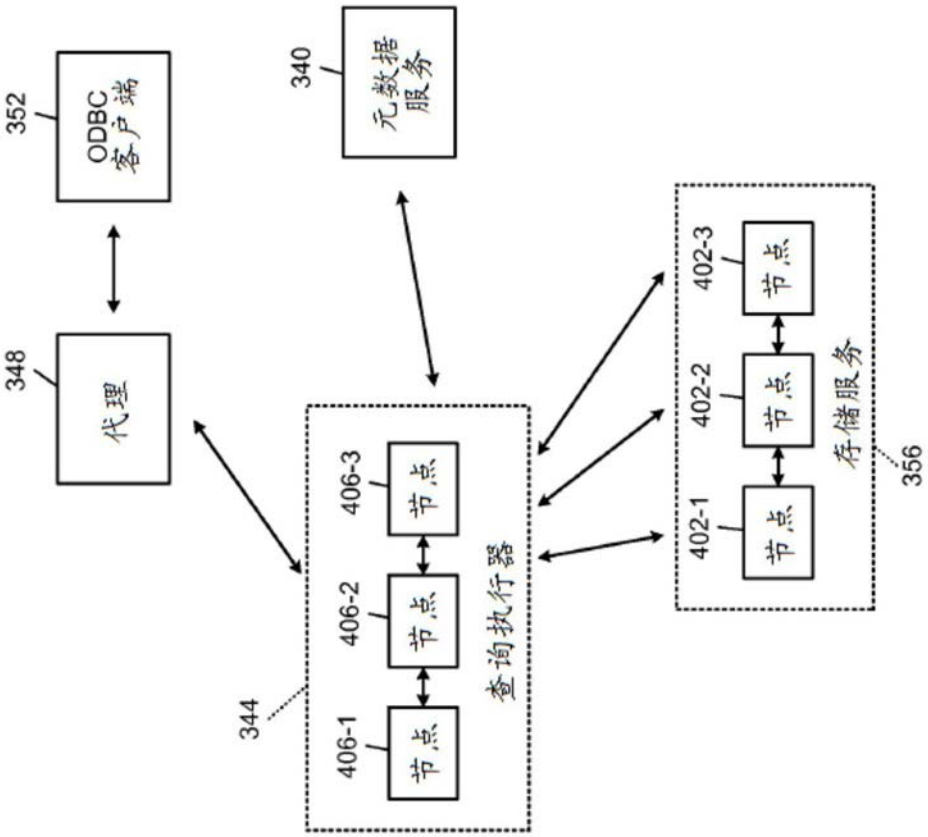

352
ODBC
客户端
340
元数据
服务
348
代理
406-3
节点
406-2
节点
406-1
节点
344
查询执行器
402-3
节点
402-2
节点
402-1
节点
存储服务
356

图2B

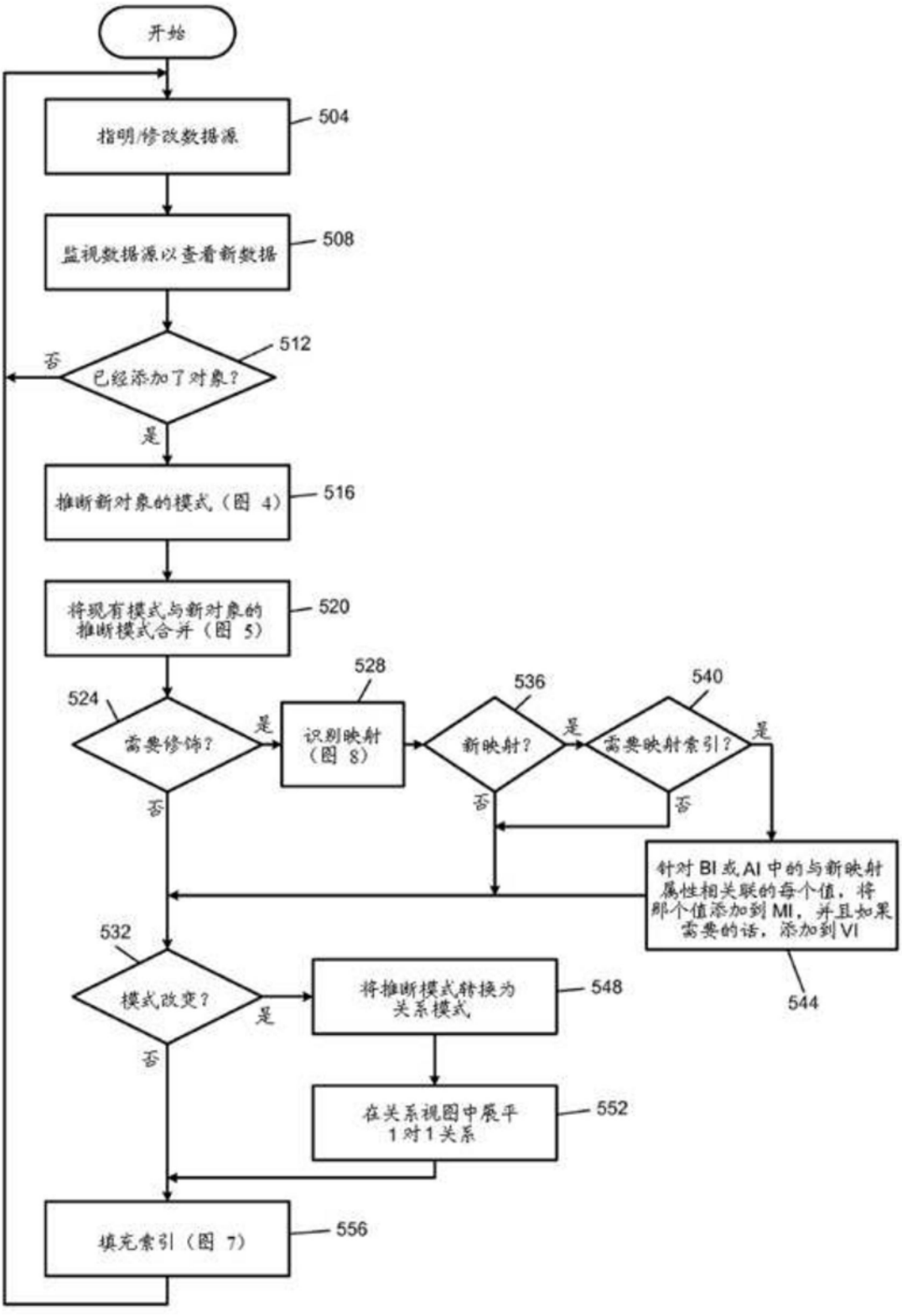

开始
指明/修改数据源
504
监视数据源以查看新数据
508
已经添加了对象?
512
否
是
推断新对象的模式（图 4）
516
将现有模式与新对象的推断模式合并（图 5）
520
524
需要修饰?
是
否
528
识别映射（图 8）
536
新映射?
是
否
540
需要映射索引?
是
否
针对 BI 或 AI 中的与新映射属性相关联的每个值，将那个值添加到 MI，并且如果需要的话，添加到 VI
544
532
模式改变?
是
否
将推断模式转换为关系模式
548
在关系视图中展平 1 对 1 关系
552
填充索引（图 7）
556

图3

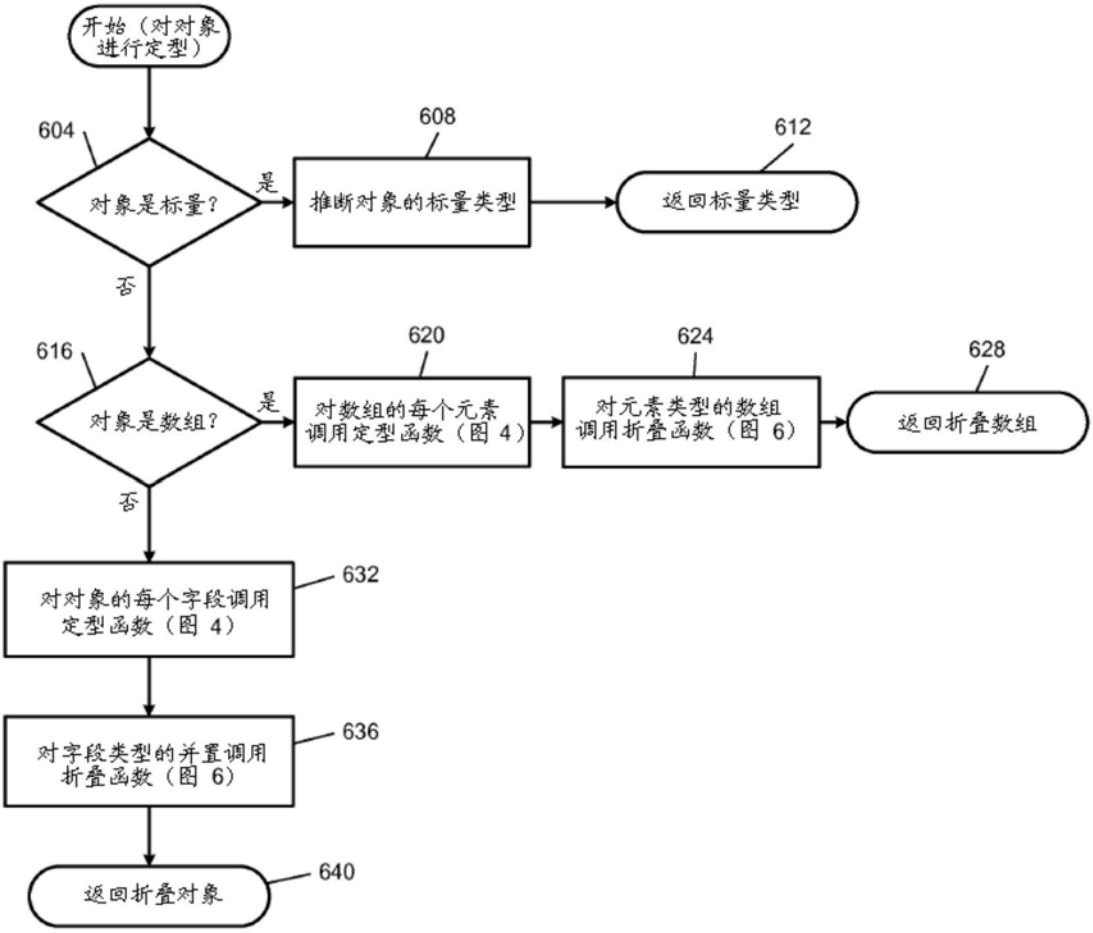

开始（对对象进行定型）
604
对象是标量？
是
608
推断对象的标量类型
612
返回标量类型
否
616
对象是数组？
是
620
对数组的每个元素调用定型函数（图 4）
624
对元素类型的数组调用折叠函数（图 6）
628
返回折叠数组
否
对对象的每个字段调用定型函数（图 4）
632
对字段类型的并置调用折叠函数（图 6）
636
返回折叠对象
640

图4

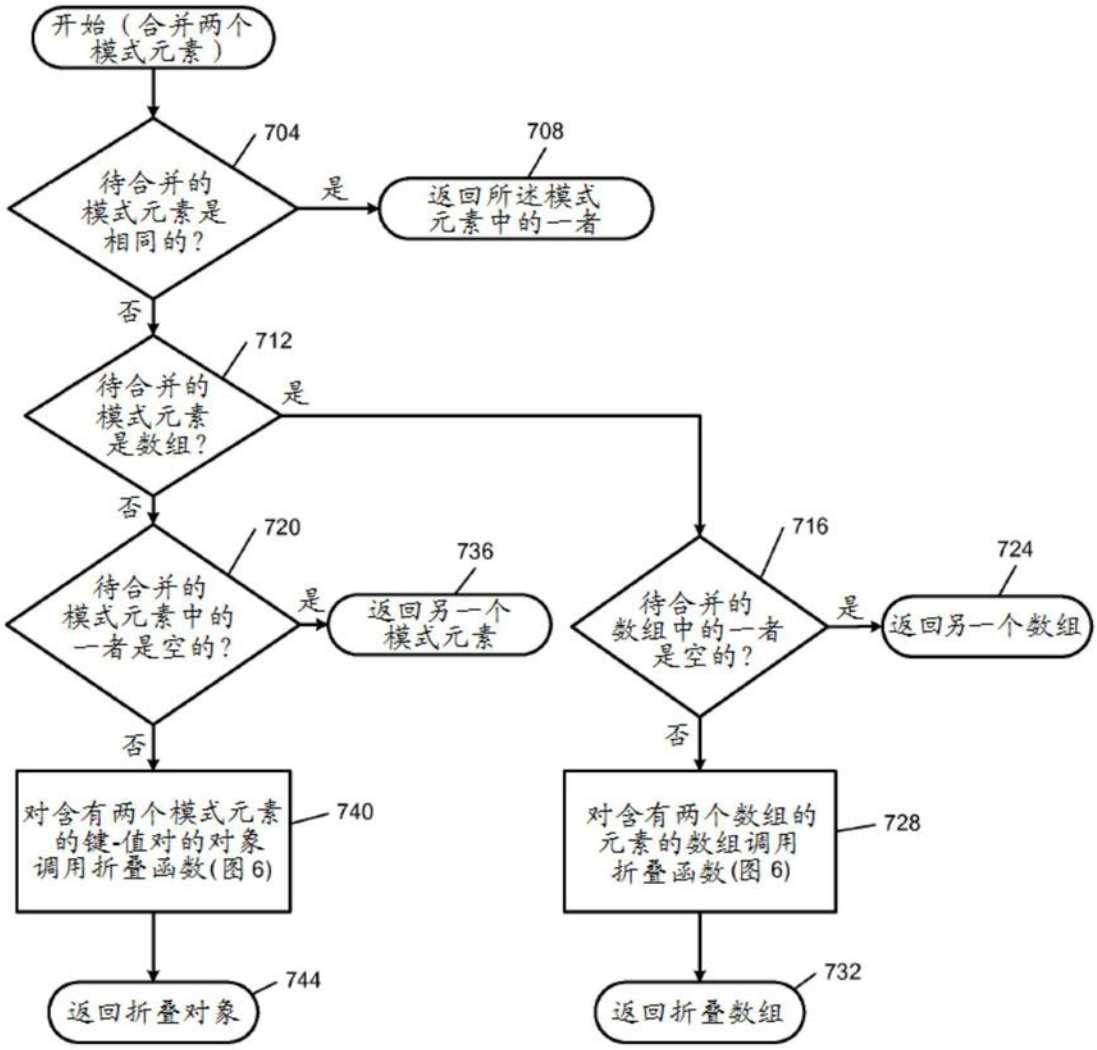

开始（合并两个模式元素）
704
待合并的模式元素是相同的？
是
708
返回所述模式元素中的一者
否
712
待合并的模式元素是数组？
是
否
720
待合并的模式元素中的一者是空的？
是
736
返回另一个模式元素
否
716
待合并的数组中的一者是空的？
是
724
返回另一个数组
否
对含有两个模式元素的键-值对的对象调用折叠函数(图 6)
740
对含有两个数组的元素的数组调用折叠函数(图 6)
728
返回折叠对象
744
返回折叠数组
732

图5

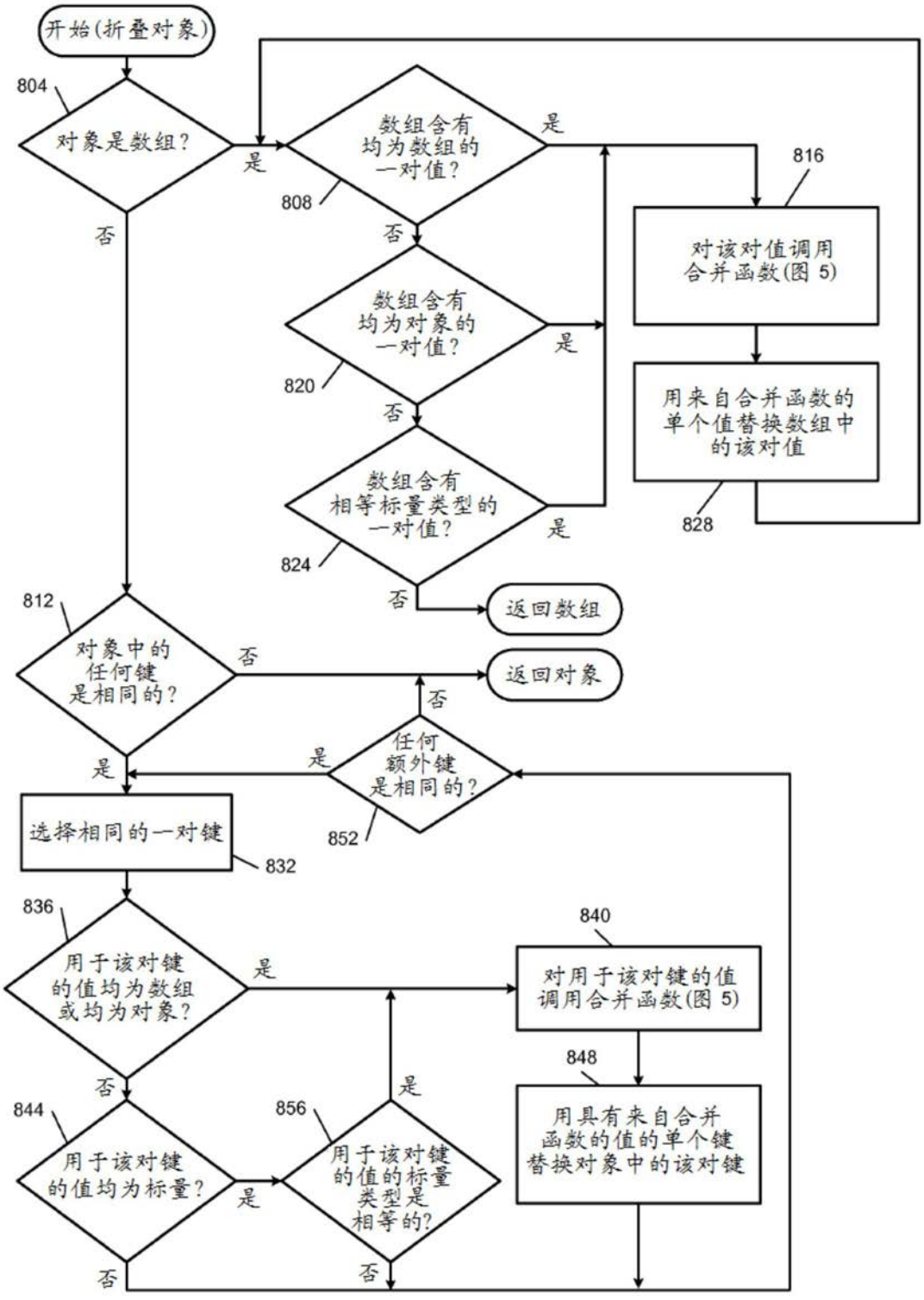

开始(折叠对象)
804
对象是数组?
是
否
数组含有均为数组的一对值?
808
是
否
816
对该对值调用合并函数(图 5)
数组含有均为对象的一对值?
820
是
否
用来自合并函数的单个值替换数组中的该对值
828
数组含有相等标量类型的一对值?
824
是
否
返回数组
812
对象中的任何键是相同的?
否
返回对象
否
任何额外键是相同的?
852
是
是
选择相同的一对键
832
836
用于该对键的值均为数组或均为对象?
是
840
对用于该对键的值调用合并函数(图 5)
848
否
是
844
用于该对键的值均为标量?
是
856
用于该对键的值的标量类型是相等的?
用具有来自合并函数的值的单个键替换对象中的该对键
否
否

图6

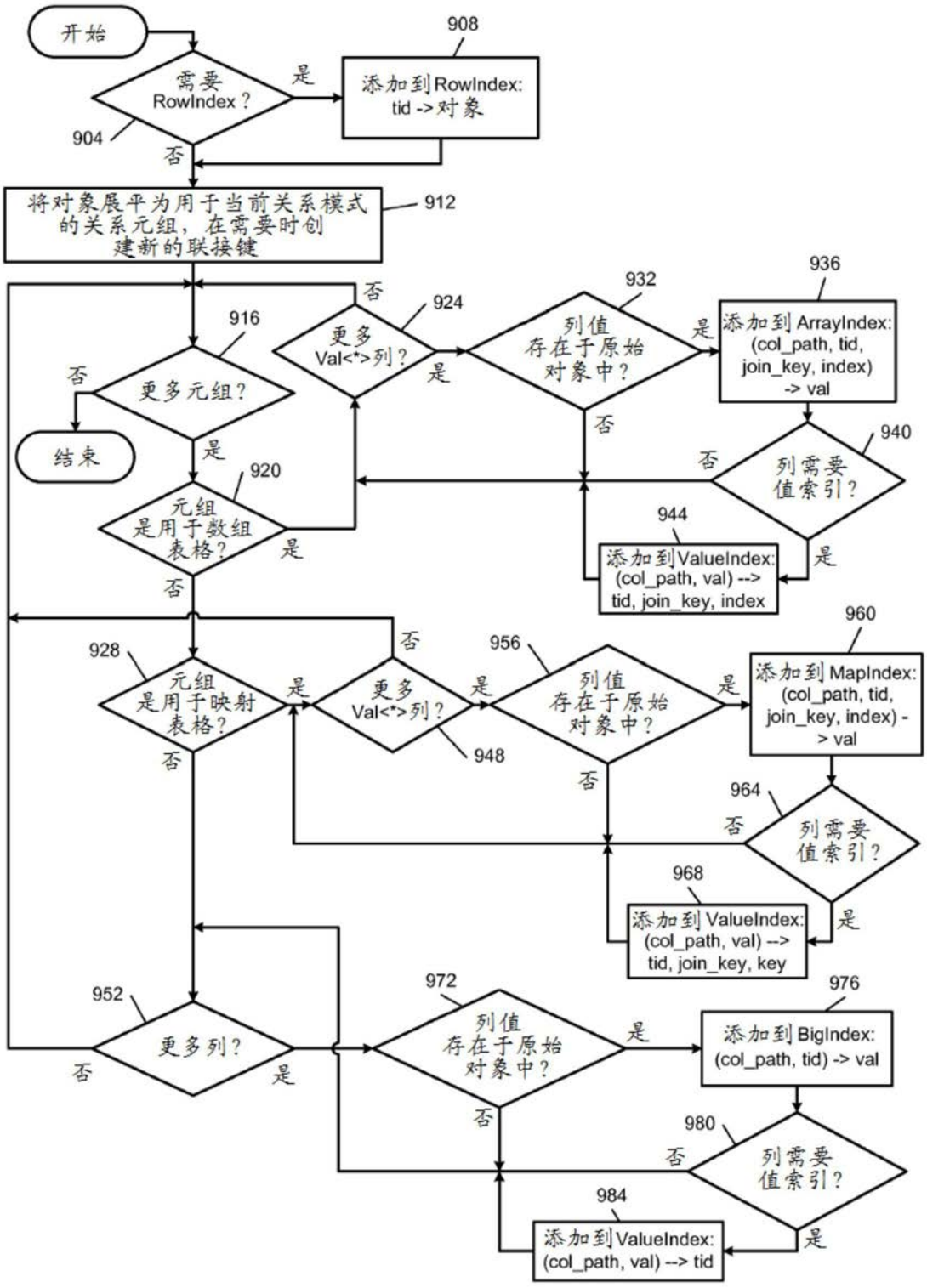

开始
需要RowIndex？
是
否
904
908
添加到RowIndex: tid -> 对象
将对象展平为用于当前关系模式的关系元组，在需要时创建新的联接键
912
更多Val<*>列？
是
否
924
列值存在于原始对象中？
是
否
932
添加到ArrayIndex: (col_path, tid, join_key, index) -> val
936
列需要值索引？
是
否
940
添加到ValueIndex: (col_path, val) --> tid, join_key, index
944
更多元组？
是
否
916
结束
元组是用于数组表格？
是
否
920
元组是用于映射表格？
是
否
928
更多Val<*>列？
是
否
948
956
列值存在于原始对象中？
是
否
添加到MapIndex: (col_path, tid, join_key, index) -> val
960
列需要值索引？
是
否
964
添加到ValueIndex: (col_path, val) --> tid, join_key, key
968
更多列？
是
否
952
列值存在于原始对象中？
是
否
972
添加到BigIndex: (col_path, tid) -> val
976
列需要值索引？
是
否
980
添加到ValueIndex: (col_path, val) --> tid
984

图7

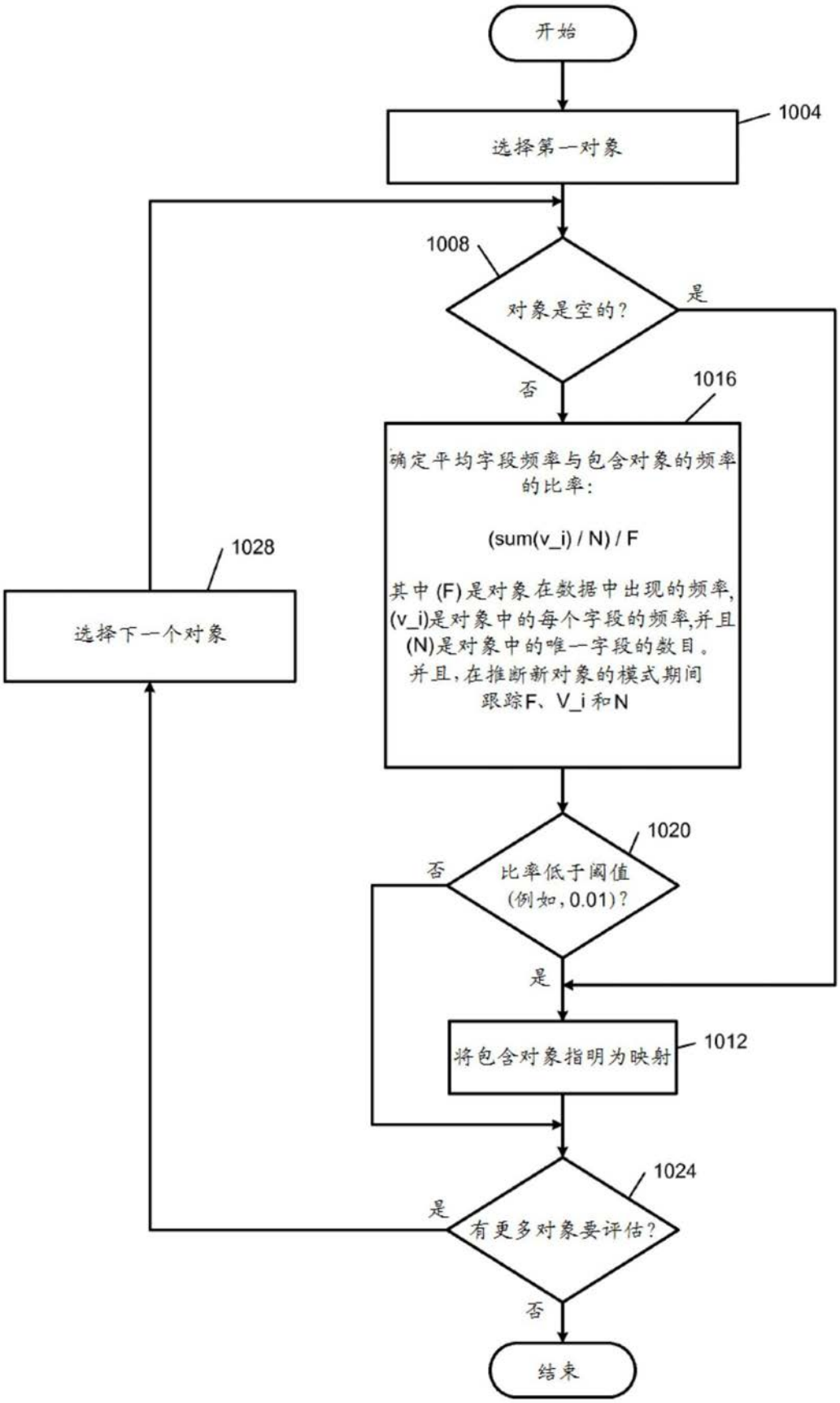

开始
选择第一对象
1004
1008
对象是空的?
是
否
1016
确定平均字段频率与包含对象的频率的比率:
(sum(v_i) / N) / F
其中(F)是对象在数据中出现的频率,(v_i)是对象中的每个字段的频率,并且(N)是对象中的唯一字段的数目。
并且,在推断新对象的模式期间跟踪F、V_i和N
1028
选择下一个对象
1020
否
比率低于阈值(例如,0.01)?
是
将包含对象指明为映射
1012
1024
是
有更多对象要评估?
否
结束

图8

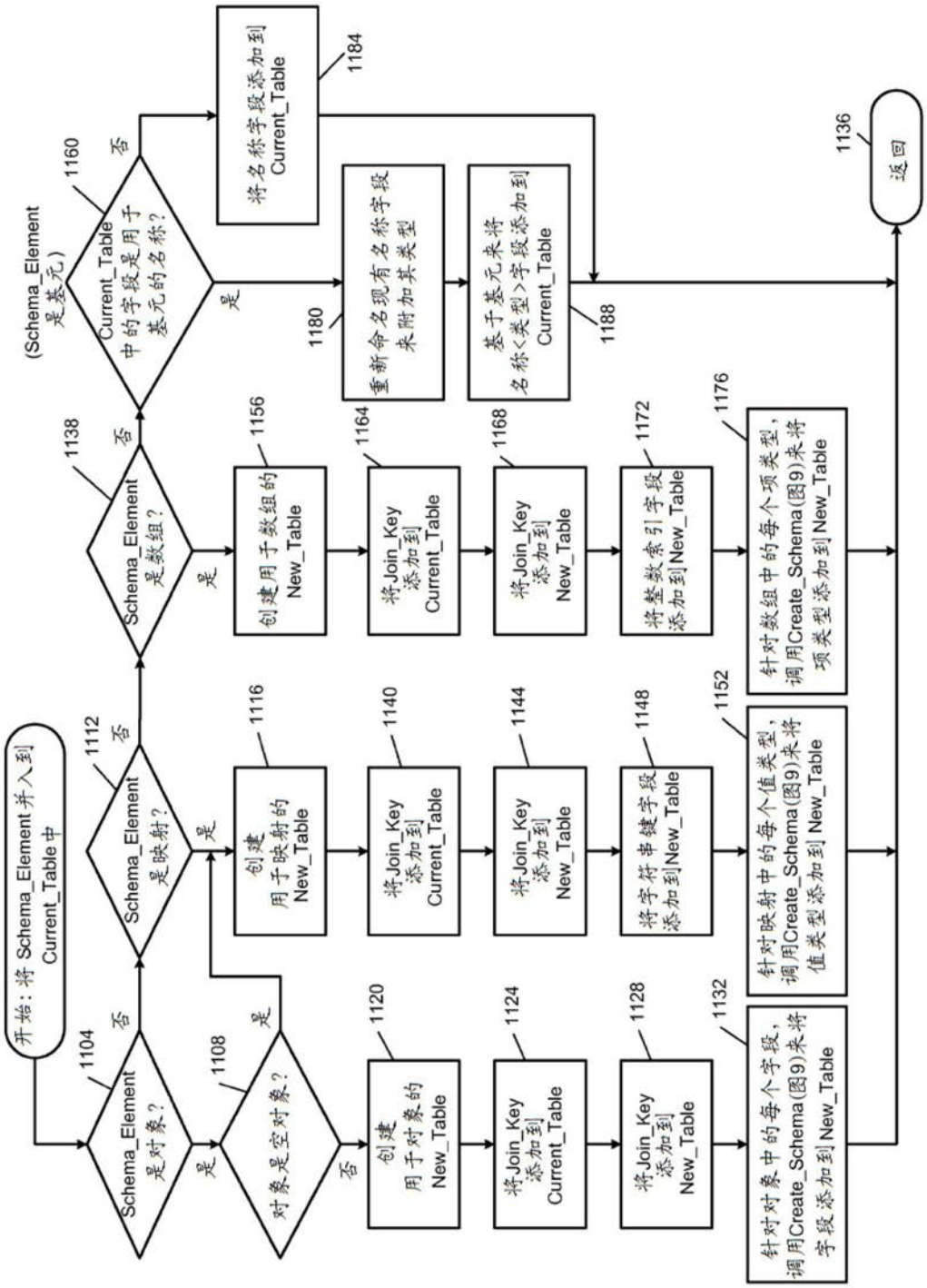

开始：将 Schema_Element 并入到 Current_Table 中
Schema_Element 是对象？
1104
否
是
对象是空对象？
1108
是
否
创建用于对象的 New_Table
1120
将Join_Key添加到 Current_Table
1124
将Join_Key添加到 New_Table
1128
针对对象中的每个字段，调用Create_Schema(图9)来将字段添加到 New_Table
1132
Schema_Element 是映射？
1112
否
是
创建用于映射的 New_Table
1116
将Join_Key添加到 Current_Table
1140
将Join_Key添加到 New_Table
1144
将字符串键字段添加到New_Table
1148
针对映射中的每个值类型，调用Create_Schema(图9)来将值类型添加到 New_Table
1152
Schema_Element 是数组？
1138
否
是
创建用于数组的 New_Table
1156
将Join_Key添加到 Current_Table
1164
将Join_Key添加到 New_Table
1168
将整数索引字段添加到New_Table
1172
针对数组中的每个项类型，调用Create_Schema(图9)来将项类型添加到New_Table
1176
(Schema_Element 是基元)
Current_Table 中的字段是用于基元的名称？
1160
否
是
将名称字段添加到 Current_Table
1184
重新命名现有名称字段来附加其类型
1180
基于基元来将名称<类型>字段添加到 Current_Table
1188
返回
1136

图9