



(12)发明专利

(10)授权公告号 CN 105164667 B

(45)授权公告日 2018.09.28

(21)申请号 201380076218.9

(22)申请日 2013.06.26

(65)同一申请的已公布的文献号

申请公布号 CN 105164667 A

(43)申请公布日 2015.12.16

(85)PCT国际申请进入国家阶段日

2015.10.29

(86)PCT国际申请的申请数据

PCT/US2013/047765 2013.06.26

(87)PCT国际申请的公布数据

W02014/209292 EN 2014.12.31

(73)专利权人 安提特软件有限责任公司

地址 美国加利福尼亚州

(72)发明人 A·西米特西斯 W·K·威尔金森

(74)专利代理机构 北京市铸成律师事务所

11313

代理人 张臻贤 屈小春

(51)Int.Cl.

G06F 17/00(2006.01)

G06F 17/30(2006.01)

(56)对比文件

US 2013097592 A1,2013.04.18,

US 2007214111 A1,2007.09.13,

WO 2012033497 A1,2012.03.15,

US 8160999 B2,2012.04.17,

US 2011239202 A1,2011.09.29,

CN 102681878 A,2012.09.19,

CN 101034390 A,2007.09.12,

CN 101727513 A,2010.06.09,

审查员 党英霞

权利要求书2页 说明书7页 附图5页

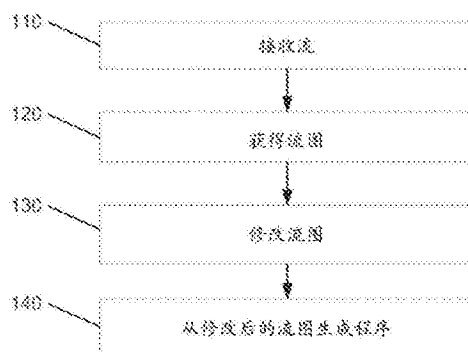
(54)发明名称

修改分析流

(57)摘要

本文公开了用于修改分析流的技术。流可以与执行引擎关联。可以获得表示流的流图。该流图可以用逻辑语言修改。例如,可以生成用逻辑语言表达的新流图。可以从修改后的流图生成程序。

100



1. 一种用于修改分析流的方法,包括由处理系统:
接收与第一执行引擎关联的流;
获得表示所述流的流图;
使用逻辑语言修改所述流图;以及
从修改后的流图生成程序以在执行引擎上执行,
其中所述流图是所述第一执行引擎响应于对所述流的执行计划的请求而输出的执行计划,
其中修改所述流图包括:
解析所述流图;
将解析后的流图转换为采用所述逻辑语言的第二流图;以及
修改所述第二流图以获得所述修改后的流图。
2. 根据权利要求1所述的方法,其中修改所述流图进一步包括优化所述第二流图。
3. 根据权利要求1所述的方法,
其中修改所述流图进一步包括:将所述第二流图分解成子流,并且
其中从修改后的流图生成程序包括:基于所述子流中的一个生成至少第一程序以在所述第一执行引擎上执行,以及基于所述子流中的另一个生成第二程序以在第二执行引擎上执行。
4. 根据权利要求1所述的方法,其中修改所述流图进一步包括:将所述第二流图与另一个流关联的至少一个其它流图组合。
5. 根据权利要求1所述的方法,进一步包括:
在解析所述流图之前,确定所述流的嵌套度;并且
其中修改所述流图进一步包括:基于所述嵌套度将所述第二流图分解成子流。
6. 根据权利要求5所述的方法,其中所述嵌套度基于所述流、所述流的执行窗、所述第一执行引擎、以及包括所述第一执行引擎的系统的状态信息来确定。
7. 根据权利要求1所述的方法,其中所述流用与所述第一执行引擎关联的第一高级语言来表达,并且所述程序用与所述执行引擎关联的第二高级语言来表达。
8. 一种用于修改分析流的系统,包括:
流图模块,用于获得表示与执行引擎关联的流的流图;
解析器,用于将所述流图解析成多个要素;
逻辑流生成器,用于基于所述多个要素生成用逻辑语言表达的逻辑流图;以及
代码生成器,用于从所述逻辑流图生成程序,
其中所述流图模块被配置为通过从所述执行引擎请求所述流的执行计划而获得所述流图。
9. 根据权利要求8所述的系统,进一步包括逻辑流处理器,用于优化所述逻辑流图、分解所述逻辑流图、或者呈现所述逻辑流图的图形视图中的至少一个。
10. 根据权利要求9所述的系统,其中所述逻辑流处理器被配置为将所述逻辑流图与第二逻辑流图组合,以生成单个逻辑流图。
11. 一种非暂时性计算机可读存储介质,所述介质存储由计算机执行以修改分析流的指令,该指令在执行时,使所述计算机:

获得表示与第一执行引擎关联的流的流图；
从所述流图生成用逻辑语言表达的逻辑流图；
将所述逻辑流图分解成多个子流；以及
生成与所述子流对应的多个程序，以在多个执行引擎上执行，
其中所述流图是所述第一执行引擎响应于对所述流的执行计划的请求而输出的执行计划。

修改分析流

背景技术

[0001] 存在很多用于处理分析流的执行引擎。这些引擎可能只接受用高级编程语言,如特定的脚本语言(例如,PigLatin、结构化查询语言(SQL))或者某个流设计工具(例如,Pentaho数据集成(PDI)平台)的语言所表示的输入流。而且,即使是支持相同的编程语言或者流设计工具的执行引擎也可能会提供分析操作的不同实现等等。因此,一个引擎的输入流可能与另一个引擎的输入流不同,即使这两个流是要达到相同的结果。出于这些考虑,修改分析流可能是具有挑战性的并且耗时。而且,若要用一种通用的解决方案在各种各样的经常包括各种执行引擎的分析环境中修改分析流同样是困难的。

附图说明

[0002] 下面的详细描述参考附图,其中:

[0003] 图1图示出根据示例的修改分析流的方法。

[0004] 图2图示出根据示例的修改流图的方法。

[0005] 图3图示出根据示例的示例流。

[0006] 图4图示出根据示例的与示例流对应的、具有解析记号的示例执行计划。

[0007] 图5图示出根据示例的用于修改分析流的计算系统。

[0008] 图6图示出根据示例的用于修改分析流的计算机可读介质。

[0009] 图7图示出根据示例的使用所公开的技术获得的实验结果。

具体实施方式

[0010] 如本文所描述的,本申请涉及对一个或多个数据集应用一系列操作的分析数据处理引擎。此一系列操作在本文中被称为“流”,因为可以将分析计算建模成有向图,在有向图中,节点表示对数据集的操作,并且弧线表示操作之间的数据流。流通常用人们容易编写、阅读和理解的高级语言来说明。给定流的高级语言表示在本文中被称为“程序”。例如,高级语言可以是特定的脚本语言(例如,PigLatin、结构化查询语言(SQL))或者某个流设计工具(例如,Pentaho数据集成(PDI)平台)的语言。在一些情况下,分析引擎是个黑盒子,即,其内部的处理是隐藏的。为了修改要输入到黑盒子执行引擎中的程序,通常要编写一个附属处理引擎,该附属处理引擎是在执行引擎和应用程序中间用于创建程序的一个独立的软件模块中间体。然后,这个附属引擎可以用于从源程序创建一个新的、修改后的程序,其中新程序具有附加的特征。为此,附属引擎通常需要理解程序的语义。编写这样的附属引擎可能会是困难的,因为在各种各样的分析环境中,有大量不同的执行引擎,这些执行引擎支持各种语言,并且其中许多执行引擎具有特定于唯一引擎的操作实现。而且,经常可以用各种方式表达一个程序,以达到相同的结果。此外,翻译程序会需要元数据,而从黑盒子执行引擎的外部看不见元数据,因此需要进行推断,这通常容易出错。

[0011] 许多分析引擎支持“解释计划”命令,在给定源程序的情况下,“解释计划”命令会返回该程序的流图。这个流图可以被称作“执行计划”或“解释计划”(在下文中,称作“执行

计划”)。所公开的系统和方法通过解析执行计划而非解析特定于用户的高级语言程序来均衡执行计划。这样,可能任务更简单,并且信息更丰富,因为分析引擎优化器所做的一些实体性的选择可能在执行计划中是有用的,而在原始的源程序(例如,实现算法、成本估计、资源利用)中会是不可用的。接着,附属引擎可以修改流图,以增加功能。然后,附属引擎可以从修改后的流图生成高级语言的新程序,以在黑盒子执行引擎(或者某些其它引擎)中执行。并且,可以利用优化和分解,以使用更为有效的方式执行流。

[0012] 根据示例,实现本文所描述的的原理的技术可以包括:接收与第一执行引擎关联的流。可以获得表示该流的流图。例如,可以从第一执行引擎请求执行计划。流图可以用逻辑语言修改。例如,可以生成用逻辑语言表达的逻辑流图。可以从修改后的流图生成程序,以在执行引擎上执行。执行引擎可以是第一执行引擎,或者可以是不同的执行引擎。而且,执行引擎可以是一个以上执行引擎,使得多个程序被生成。以下参考附图描述附加示例、优点、特征、修改等等。

[0013] 图1图示出根据示例的修改分析流的方法。方法100可以通过计算设备、系统或者计算机,例如计算系统500或者计算机600来执行。用于实现方法100的计算机可读指令可以存储在计算机可读存储介质上。在本文中,这些存储在介质上的指令被称作“模块”,并且可以由计算机执行。

[0014] 方法100可以从110开始,在110处,可以接收与第一执行引擎关联的流。该流可以包括例如实现类型、资源、存储路径等等的实现细节,并且这些实现细节是特定于第一执行引擎的。例如,取决于第一执行引擎的类型,该流可以用高级编程语言来表达,比如用特定的编程语言(例如,SQL、PigLatin)或者用如提取-变换-加载(ELT)流设计工具PDI这样的特定流设计工具的语言来表达。

[0015] 可以有一个以上的流。例如,可以接收混合流,该混合流可以包括针对不同执行引擎的多个部分(即,子流)。例如,第一流可以用SQL编写,而第二部分可以用PigLatin编写。另外,在支持相同编程语言的执行引擎之间也可能有差异。例如,第一SQL执行引擎(例如,HP Vertica SQL引擎)的脚本可能与第二SQL执行引擎(例如,Oracle SQL引擎)不兼容(例如,可能不正确地运行)。

[0016] 在120处,可以获得表示该流的流图。该流图可以是第一执行引擎获得的执行计划。例如,可以用解释计划命令请求执行计划。如果有多个流,可以从每个流的相应执行引擎获得该流的单独执行计划。如果流用流设计工具的语言表达,则可以从关联的执行引擎请求流说明(例如,用XML表达的)。可以基于从引擎接收的流说明而生成流图。

[0017] 在130处,可以使用逻辑语言修改流图。图2图示出根据示例的修改流图的方法200。

[0018] 在210处,可以将流图解析为多个要素。例如,解析器可以分析流图,并针对流图中每个操作符或者数据存储获取特定于引擎的信息。解析器可以输出构成流图的节点(此处称为“要素”)。因为解析器是特定于引擎的,所以每个被支持的引擎可以有一个单独的解析器。这样的解析器可以作为插件被添加到系统中。

[0019] 在220处,可以将解析后的流图转换为采用逻辑语言的第二流图。该第二流图在本文中被称作“逻辑流图”。该逻辑流图可以通过将多个要素转换为用逻辑语言表示的逻辑要素而生成。这里,示例的逻辑语言是xLM,该语言是由惠普公司的惠普实验室为分析流而研

发的。但是,也可以用其他的逻辑语言。另外,可以使用字典来进行这个转换。字典可以包括逻辑语言和与第一实体流的至少一个执行引擎相关的编程语言之间的映射。因此,字典224使得将特定于引擎的多个要素翻译成引擎不可知的逻辑要素,逻辑要素构成逻辑流。在2013年6月24日提交的PCT/US2013/047252中,进一步详细描述了字典和相关转换,其内容通过引用包含于此。

[0020] 在230处,可以修改逻辑流程图。例如,可以以自动的方式或者通过在GUI中的手动操作的方式对逻辑流程图进行各种优化。因为各种原因,比如因为流是混合流、因为流包括不能被流的执行引擎所优化的用户定义的功能等等原因,在只处理流时,可能不能进行这样的优化。与此相关,可以收集逻辑流程图的统计数据。另外,可以在图形用户界面(GUI)上以图形的方式显示逻辑流程图。这样可以让用户更好地理解流(与其源对应物相比),特别是在流是混合流的情况下。

[0021] 而且,可以将逻辑流程图分解为子流,以利用特定的执行环境。例如,执行环境可以有各种各样的执行引擎,可以均衡这些执行引擎来协作,以更有效的方式执行整个流。在这方面,可以采用流执行调度器。类似地,该逻辑流程图可以和与另一个流关联的另一个逻辑流程图组合。该另一个流可以针对不同的执行引擎,并且可能与第一执行引擎不兼容。但是,表示在逻辑流程图中的是,这两个流现在可以用连接器组合。

[0022] 返回到图1,在140处,可以从修改后的流图(即,逻辑流程图)生成程序。该程序可以被生成以在执行引擎上执行。执行引擎可以是第一执行引擎,或者可以是不同的执行引擎。另外,在将逻辑流程图分解为子流的情况下,可以有多个执行引擎。这样,可以将程序用适于每个它所用的执行引擎的高级语言来表达。

[0023] 这个转换可以包括生成特定于引擎的逻辑流程图的中间版本,然后由该中间版本生成程序代码。当逻辑流程图描述主要流结构时,在初始转换为逻辑语言(例如,xLM)的过程中,可能不包括许多特定于引擎的细节。这些细节包括到脚本中的数据存储的路径,或者坐标,或者流设计中的其他设计元数据。当生成特定于引擎的xLM时,可以获取这样的细节。另外,其他xLM结构,例如操作符类型或者用于表示操作符参数的表达式的规范表达式形式,应当被转换为特定于引擎的格式。这些转换可以由xLM解析器进行。另外,一些引擎需要一些额外的流元数据(例如,流设计工具可能需要流结构的形状、颜色、大小和位置)以处理和利用流。字典可以包括用于在不同的引擎中表示操作符的具有默认元数据信息的模板。

[0024] 最终可以通过从特定于引擎的第二逻辑表示(特定于引擎的xLM)生成代码而生成程序。代码可以在一个或多个执行引擎上执行。这个到可执行代码的转换可以使用代码模板来完成。通过解析特定于引擎的xLM中的每个xLM要素,来解析特定于引擎的xLM,确保重视每个要素可能具有的依附关系。具体地,可以搜索每个要素的代码模板,来找到xML要素所表示的与特定的操作、实现和引擎对应的模板。

[0025] 对于包括多个部分的流(例如,混合流),逻辑流可以将该多个部分表示为经由连接器操作符连接。为了生成执行代码,根据所选择的执行引擎和存储库,可以将连接器操作符实例化为适当的格式(例如,映射-归纳(map-reduce)连接器的数据库、将数据从存储库A转移到存储库B的脚本)。然后,可以将程序输出并分发到适当的引擎以执行。

[0026] 现在描述包括流和执行计划的示例性示例。图3图示出被表述为SQL查询的示例性流300。所示的流300分成三个主逻辑部分。这些划分线是用于添加将此单个流分解成多个

部分(或者“子流”)的分割点的候选。

[0027] 图4图示出执行引擎响应于解释计划命令而可能生成的流300的示例执行计划400。所示的执行计划400也被分成相同的三个逻辑部分,与流300对应。执行计划400如下被解析。为了在将流操作符从执行计划400中读取时,添加流操作符,可以维护一个队列Q(此处是后入先出(LIFO)队列)。解析可以从计划400的根部(由“+”表示)开始,其接着是操作符名称(“SELECT”)。将SELECT加到Q中。该计划有不同的级别,这些级别由符号“|”表示。接着继续对该计划全部进行解析,将每个新操作符加到Q中。在每个级别,第一个遇到的操作符优先。在图4中,用符号“|+>”表示新操作符。如果一个操作符是二元的,那么对其子成员分别进行表示(例如,在JOIN操作符中,通过内部关系分开外部)。在这种情况下,可以用一个特殊的符号对其进行表示(例如,此处,“||||+--Inner->”表示深度为4的内部关系)。当解析完这个计划时,将所有的要素反向从队列Q中出列。每个要素是流图中的一个流操作符。

[0028] 如前文所描述那样,附属的处理引擎可以通过进行流分解而修改流。流分解在使执行更快或者减少资源竞争方面是有用的。可能的划分流的候选位置是在不同的级别,当嵌套有选择类型的操作符时,在开销大的操作之后,等等。这样的点还可以作为恢复点,使得加强后的程序的容错性得到改善。

[0029] 为了辅助分解,可以基于执行要求和服务级别目标(其可以用目标函数表示)来确定流的嵌套度 λ 。一个旨在减少资源竞争的示例性目标函数可以作为给定流的参数(argument)、流可接受的执行窗的阈值、运行流的关联执行引擎、以及系统状态(例如,系统利用率、未决工作量)。

[0030] 嵌套度 λ 可以是具体的值(例如,数字或者百分比),或者是更抽象的值(例如,在[‘低-未嵌套’、‘中’、‘高-嵌套’]的范围内)。使用 λ 可以估计要产生多少个流片段 k (即,应当将输入的流分解成多少个子流)。可以根据流大小与 λ 的比值(例如: $\# \text{节点数} / \lambda$)计算出一个示例的估计值。 λ 的值大(高嵌套),流片段的数目 k 低,随着 $\lambda \rightarrow \infty, k \rightarrow 0$ 。相反, λ 的值较小,可以更积极地分解流。因此,另一个极端是,随着 $\lambda \rightarrow 0, k \rightarrow \infty$,这实质上意味着,应当在每个操作符之后分解流(每个操作符包括一个流片段/子流)。

[0031] 作为示例,如果流是用SQL实现的,那么可以将流看作是查询。在这种情况下,随着 $\lambda \rightarrow \infty$,尽可能地嵌套查询。例如,对于由创建一个表和一个视图(例如,该视图从表中读取数据)的两个SQL语句构成的流,该流包含的流片段不能少于两个。但是,对于流300而言,嵌套版本如图4所示。另一方面,随着 $\lambda \rightarrow 0$,将查询分解成和其操作符一样多的片段,这些片段彼此之间通过中间表连接。例如,流300可以被分解成最多三个片段,每个片段与三个主逻辑部分中的一个相对应。

[0032] 随后,当嵌套度可用时,可以使用 λ 解析执行计划。例如,可以将执行解析的解析函数看作是嵌套度的可选参数。然后,在每个新操作符处,可以评估成本函数,以检查在该点处增加分割点是否有意义。基于 λ 值,可以在当前正在解析的操作符之后增加分割点。这样, λ 值可以被认为是确定成本函数应该更为保守或更不保守(或者,积极)的要点。

[0033] 图5图示出根据示例的用于修改分析流的计算系统。计算系统500可以包括一个或多个计算机和/或由一个或多个计算机实现。例如,计算机可以是服务器计算机、工作站计算机、台式计算机、笔记本电脑、移动设备等等,并且可以是分布式系统的一部分。计算机可

以包括一个或多个控制器和一个或多个机器可读存储介质。

[0034] 控制器可以包括处理器和存储器,用于实现机器可读指令。处理器可以包括至少一个中央处理单元(CPU)、至少一个基于半导体的微处理器、至少一个如数字图像处理单元的数字信号处理器(DSP)、其他适于获取并执行存储在存储器上的指令的硬件设备或者处理元件、或者其组合。处理器可以包括单芯片单核或多核、跨多芯片多核、跨多设备多核、或者其组合。处理器可以从存储器获取指令,并对其进行解码、执行,以完成各种功能。作为获取并执行指令的一种替换方式,或者除此之外,处理器可以包括至少一个集成电路(IC)、其他控制逻辑、其他电子电路、或者其组合,其包括多个用于完成各种任务或者功能的电子组件。

[0035] 控制器可以包括存储器,比如机器可读存储介质。机器可读存储介质可以是任何包含或者存储可执行指令的电子、磁性、光学或者其他物理的存储设备。因此,机器可读存储介质可以包括,例如各种随机存取存储器(RAM)、只读存储器(ROM)、闪存及其组合。例如,机器可读介质可以包括非易失性随机存取存储器(NVRAM)、电可擦除可编程只读存储器(EEPROM)、存储驱动、NAND闪存等等。此外,机器可读存储介质可以是计算机可读和非暂时性的。另外,系统500可以包括与一个或多个控制器分离的一个或多个机器可读存储介质。

[0036] 计算系统500可以包括存储器510、流图模块520、解析器530、逻辑流生成器540、逻辑流处理器550、以及代码生成器560,并可以构成附属处理引擎,或者是其一部分。这些组件中的每一个组件都可以通过单个计算机或者多个计算机来实现。这些组件可以包括软件、用于存储软件的一个或多个机器可读介质、以及用于执行软件的一个或多个处理器。软件可以是包括机器可执行指令的计算机程序。

[0037] 另外,计算系统500的用户可以通过一个或多个其他计算机与计算系统500交互,该一个或多个其他计算机可以被认为或不认为是计算系统500的一部分。作为示例,用户可以通过驻留在系统500或者另一个计算机(比如,台式电脑、工作站电脑、平板电脑等等)上的计算机应用与系统500交互。计算机应用可以包括用户界面(例如,触摸界面、鼠标、键盘、手势输入设备)。

[0038] 计算机系统500可以执行方法100和200及其变型,并且组件520至560可以被配置为执行方法100和200的各个部分及其变型。另外,组件520至560实现的功能可以是较大的软件平台、系统、应用等等的部分。例如,这些组件可以是数据分析系统的部分。

[0039] 在示例中,存储器510可以被配置为存储与执行引擎关联的流512。该流可以用高级编程语言来表达。流图模块520可以被配置为获得表示流512的流图。流图模块520可以被配置为通过从执行引擎请求流512的执行计划来获得流图。解析器530可以被配置为将流图解析为多个要素。逻辑流生成器340可以被配置为基于多个要素生成用逻辑语言(例如,xLM)表达的逻辑流图。逻辑流处理器550可以被配置为将逻辑流图与第二逻辑流图进行组合,以生成一个逻辑流图。逻辑流处理器550还可以被配置为优化逻辑流图、将逻辑流图分解成子流、或者表示逻辑流图的图形视图。代码生成器560可以被配置为从逻辑流图生成程序。该程序可以用高级编程语言表达,以在一个或多个执行引擎上执行。

[0040] 图6图示出根据示例的用于修改分析流的计算机可读介质。计算机600可以是多种计算设备或者系统中的一种,例如像系统500那样。

[0041] 计算机600可以访问数据库630。数据库630可以包括一个或多个计算机,并可以包

括一个或多个控制器和机器可读存储介质,如此处描述的。计算机600可以经由网络连接到数据库630。网络可以是任意类型的通信网络,包括但不限于:基于导线的网络(例如,有线)、无线网络(例如,蜂窝、卫星)、蜂窝通信网络、基于IP的电信网络(例如,互联网协议传输声音(Voice over IP)网络)。网络还可以包括传统的固网或者公共交换电话网(PSTN)或者前述网络的组合。

[0042] 处理器610可以是至少一个中央处理单元(CPU)、至少一个基于半导体的微处理器、其他适于获得并执行存储在机器可读存储介质620上的指令的硬件设备或处理元件、或其组合。处理器610可以包括单芯片单核或多核、跨多芯片多核、跨多设备多核、或者其组合。处理器610可以获取指令622-628,并对其进行解码、执行,以实现各种处理。作为获取和执行指令的替换方式,或者除此之外,处理器610可以包括至少一个集成电路(IC)、其他控制逻辑、其他电子电路、或者其组合,其包括用于执行指令622-628的功能的多个电子组件。相应地,处理器610可以跨多个处理单元来实现,并且指令622-628可以通过计算机600的不同区域中的不同处理单元来实现。

[0043] 机器可读存储介质620可以是任何包含或存储可执行指令的电子、磁性、光学或其他物理的存储设备。因此,机器可读存储介质可以包括,例如各种随机存取存储器(RAM)、只读存储器(ROM)、闪存及其组合。例如,机器可读介质可以包括非易失性随机存取存储器(NVRAM)、电可擦除可编程只读存储器(EEPROM)、存储驱动、NAND闪存等等。此外,机器可读存储介质620可以是机器可读和非暂时性的。机器可读存储介质620可以被编码有一系列可执行指令,用于管理处理元件。

[0044] 指令622-628在由处理器610(例如,经由处理器的一个处理元件或多个处理元件)执行时,能够使处理器610执行处理,例如方法100、200及其变型。而且,计算机600可以与系统500类似,并且可以具有类似的功能,且使用方式与上文描述的方式相同。

[0045] 例如,获取指令622可以使处理器610获取表示流632的流图。流632可以与第一执行引擎相关联,并可以存储在数据库630中。LFG生成指令624可以使处理器610从流图生成用逻辑语言(例如,xLM)表达的逻辑流图。分解指令626可以使处理器610将逻辑流图分解为多个子流。程序生成指令628可以使处理器610生成与子流相对应的多个程序,以在多个执行引擎上执行。

[0046] 图7(a)至图7(b)图示出根据示例的使用所公开的技术获得的实验结果。具体地,通过这些结果来说明使用本文公开的技术分解流的益处。该实验包括运行由930个混合的分析流构成的工作量。这些流是在并行数据库上运行的TPC-DS(事务处理性能委员会-决策支持)查询。总共有93个TPC-DS查询的十个实例用MPL8(多编程级别8)按随机的顺序运行。这些流实例标示在x轴上,而对应的执行时间标示在y轴上。图7(a)图示出不分解任何流的工作量执行。图7(b)图示出使用所公开的技术进行分解的有益效果。具体地,对一些长的运行流进行分解,这创建了额外的一些流,产生了1100个流(而非930个流)的工作量。尽管在流的纯粹的数目上工作量增加,但显然,执行时间显著改善,尤其是对于图7(a)中较长的运行流而言。额外的好处是,改善了系统的资源竞争,因为不再有任何流比其它流在相当长的一段时间内独占资源。

[0047] 虽然可以针对每个特定于引擎的编程语言手动进行分解或者通过写分析器进行分解,所公开的技术可以通过均衡执行引擎将它们的程序在数据集和操作方面表达为执行

计划(解释计划)的能力而避免这个工作。为用这种形式表达的计算编写解析器会更简单,因而所公开的技术使得支持如图7(a)至图7(b)中示出的技术(并获得结果)的附属处理引擎成为可能。

[0048] 在前面的描述中,陈述了许多细节以提供对本文公开的主题的理解。但是,即使没有这些细节中的一些或者全部,也可以实现实施方式。其他实施方式可以包括对上文讨论的细节的修改和变型。所附权利要求旨在覆盖这些修改和变型。

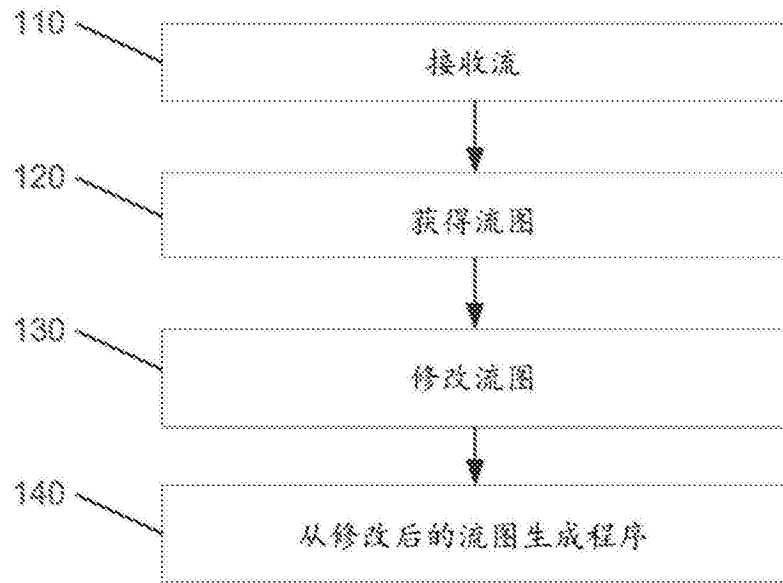
100

图1

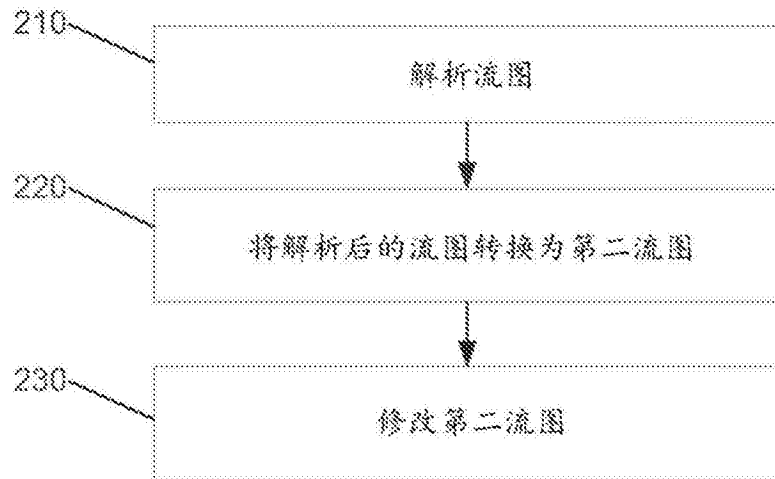
200

图2

300

查询	Select s_store_name,sum(ss_net_profit) from store_sales,date_dim,store, (select X1.ca_zip from (SELECT substr(ca_zip,1,5) ca_zip FROM customer_address WHERE substr(ca_zip,1,5) IN (<<long list of values>>>)) X1, (select ca_zip from (SELECT substr(ca_zip,1,5) ca_zip,count(*) cnt FROM customer_address, customer WHERE ca_address_sk = c_current_addr_sk and c_preferred_cust_flag='Y' group by ca_zip having count(*) > 10)A1)X2 where X1.ca_zip = X2.ca_zip) V1 where ss_store_sk = s_store_sk and ss_sold_date_sk = d_date_sk and d_qoy = 2 and d_year = 1998 and (substr(s_zip,1,2) = substr(V1.ca_zip,1,2)) group by s_store_name order by s_store_name limit 100;			
部分 1				
部分 2				
部分 3				
部分 1 (继续)				

图3

400

访问路径:	
部分1	<pre> --SELECT COUNT(*) (Cost: 25K, Rows: 100) (PATH ID: 0) -- Output Only: 100 names -- Execute on: Query Initiator -->>> GROUPBY HASH (SORT OUTPUT) (SEGMENTS: 08/03/05) (Cost: 25K, Rows: 100) (PATH ID: 1) -- Aggregates: sum(store_sales.ss_net_profit) -- Group By: store_sales.ss_net_profit -- Execute on: All Nodes -->>> JOIN HASH (Cost: 25K, Rows: 216K) (5K BLEN) (PATH ID: 2) -- Join Cond: (substr(customer_address.ca_zip, 1, 5) = a1.ca_zip) -- Materialize at Input: customer_address.ca_zip -- Execute on: All Nodes -->>> Outer -> JOIN HASH (Cost: 21K, Rows: 216K) (5K BLEN) (PATH ID: 3) -- Join Cond: (substr(store_sales.ss_sold_date_sk = substr(substr(customer_address.ca_zip, 1, 5), 1, 2)) -- Materialize at Input: store_sales.ss_sold_date_sk, store_sales.ss_sold_date_sk, store_sales.ss_net_profit -- Execute on: All Nodes </pre>
部分2	<pre> -->>> Outer -> STORAGE ACCESS for customer_address (Cost: 19, Rows: 216K) (2K BLEN) (PATH ID: 5) -- Projection: tocols:customer_address_080_10_rep_010_010_node001 -- Materialize: customer_address.ca_zip -- Filter: (substr(customer_address.ca_zip, 1, 5) = a1.ca_zip) <<< using last of values >>> -- Execute on: All Nodes -- Runtime Filters: (SP2)(HashJoin): substr(substr(customer_address.ca_zip, 1, 5), 1, 2), (SP2)(HashJoin): substr(customer_address.ca_zip, 1, 5) </pre>
部分1 (继续)	<pre> -->>> Inner -> JOIN HASH (Cost: 21K, Rows: 108K) (16 BLEN) (PATH ID: 6) -- Join Cond: (store_sales.ss_store_sk = store_sales.ss_store_sk) -- Materialize at Input: store_sales.ss_store_sk -- Execute on: All Nodes -->>> Outer -> JOIN HASH (Cost: 40, Rows: 108K) (16 BLEN) (PATH ID: 7) -- Join Cond: (store_sales.ss_sold_date_sk = date_dim.d_date_sk) -- Execute on: All Nodes -->>> Outer -> STORAGE ACCESS for store_sales (Cost: 17, Rows: 89M) (12K BLEN) (PATH ID: 8) -- Projection: tocols:store_sales_080_16_rep_010_010 -- Materialize: store_sales.ss_sold_date_sk -- Execute on: All Nodes -- Runtime Filters: (SP3)(HashJoin): store_sales.ss_sold_date_sk, (SP4)(HashJoin): store_sales.ss_store_sk -->>> Inner -> STORAGE ACCESS for date_dim (Cost: 12, Rows: 92) (PATH ID: 9) -- Projection: tocols:date_dim_010_3_rep_010_010_node001 -- Materialize: date_dim.d_date_sk -- Filter: (date_dim.d_year = 1999) -- Filter: (date_dim.d_moy = 2) -- Execute on: All Nodes -->>> Inner -> STORAGE ACCESS for store (Cost: 93, Rows: 79) (PATH ID: 10) -- Projection: tocols:store_080_5_rep_010_010_node001 -- Materialize: store.s_store_sk, store.s_store_name, store.s_zip -- Execute on: All Nodes -->>> Inner -> SELECT (Cost: 4K, Rows: 5K) (PATH ID: 11) -- Execute on: All Nodes -->>> GROUPBY PIPELINED (Cost: 4K, Rows: 5K) (PATH ID: 12) -- Aggregates: count(*) -- Group By: customer_address.ca_zip -- Filter: (cvar > 10) -- Execute on: All Nodes -->>> JOIN HASH (Cost: 4K, Rows: 136K) (PATH ID: 13) -- Join Cond: (customer_address.ca_address_sk = customer.c_current_addr_sk) -- Materialize at Output: customer_address.ca_zip -- Execute on: All Nodes -->>> Outer -> STORAGE ACCESS for customer_address (Cost: 517, Rows: 216K) (PATH ID: 14) -- Projection: tocols:customer_address_1080_14_rep_010_010_node001 -- Materialize: customer_address.ca_address_sk -- Execute on: All Nodes -- Runtime Filter: (SP1)(HashJoin): customer_address.ca_address_sk -->>> Inner -> STORAGE ACCESS for customer (Cost: 453, Rows: 136K) (PATH ID: 15) -- Projection: tocols:customer_080_26_rep_010_010_node001 -- Materialize: customer.c_current_addr_sk -- Filter: (customer.c_current_addr_sk = 1) -- Execute on: All Nodes </pre>
部分3	<pre> -->>> Inner -> STORAGE ACCESS for customer (Cost: 453, Rows: 136K) (PATH ID: 15) -- Projection: tocols:customer_080_26_rep_010_010_node001 -- Materialize: customer.c_current_addr_sk -- Filter: (customer.c_current_addr_sk = 1) -- Execute on: All Nodes </pre>

图4

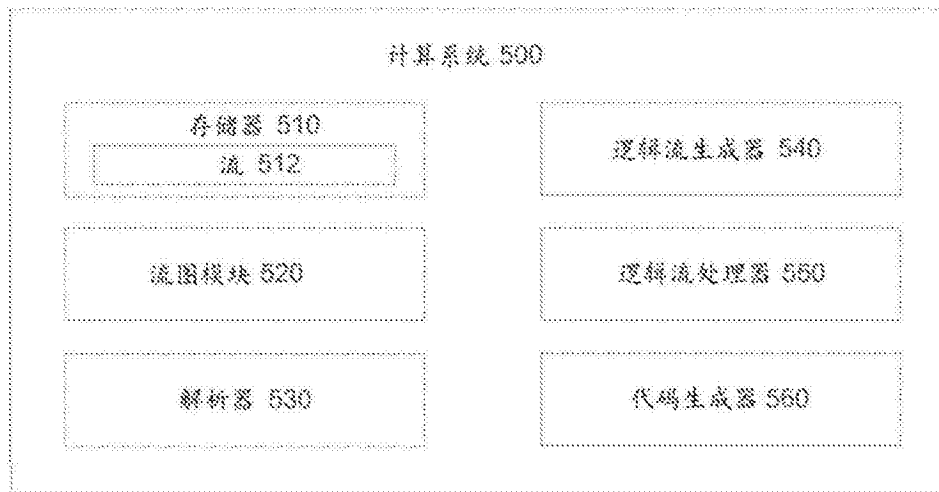


图5

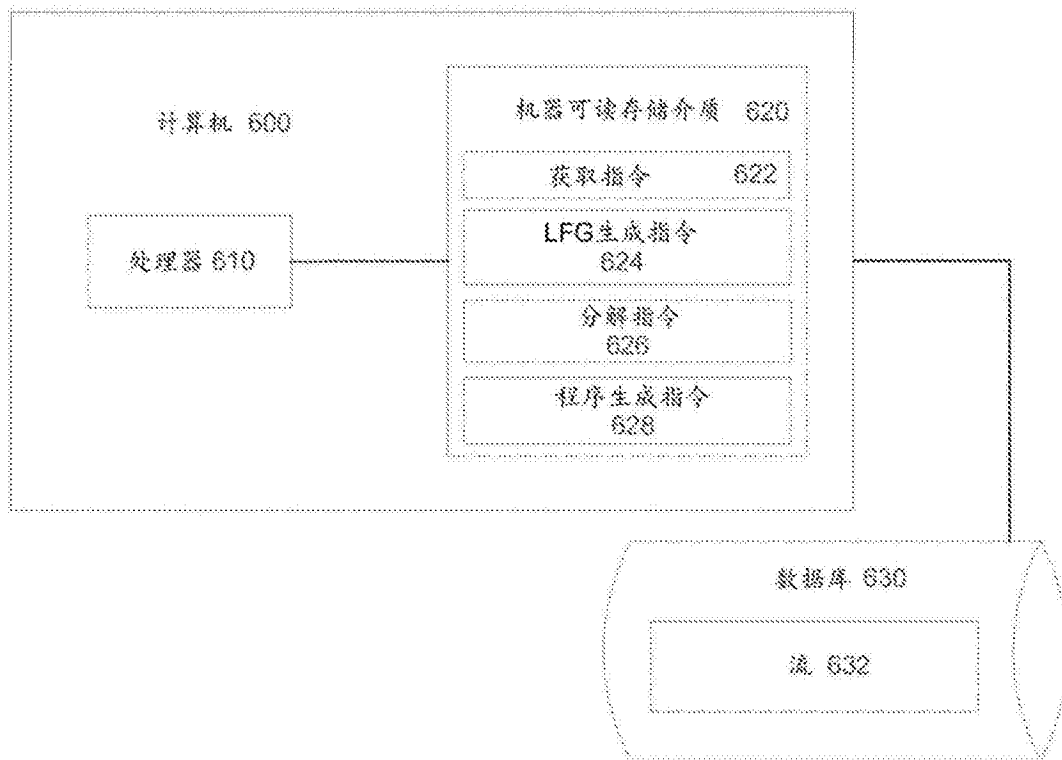


图6

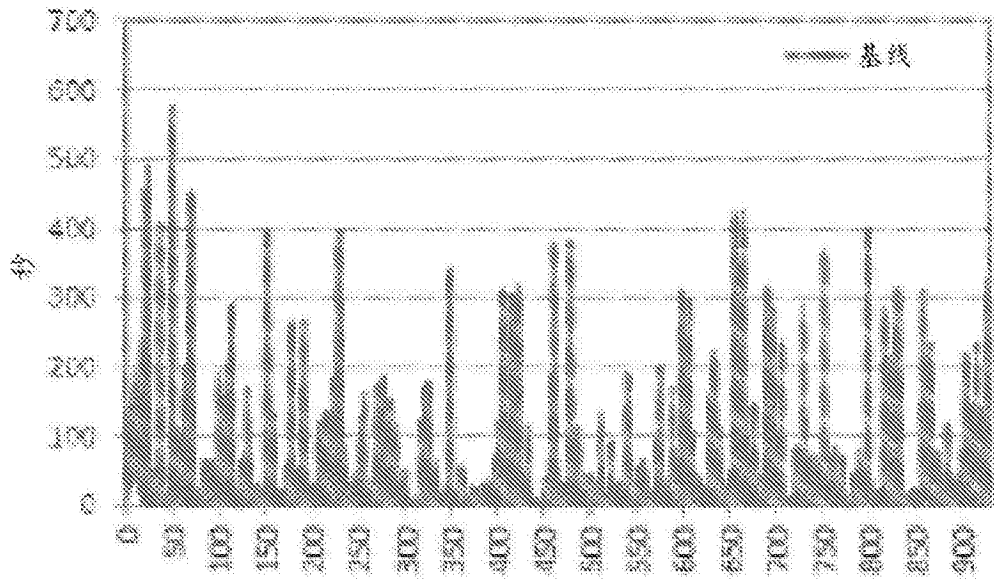


图7 (a)

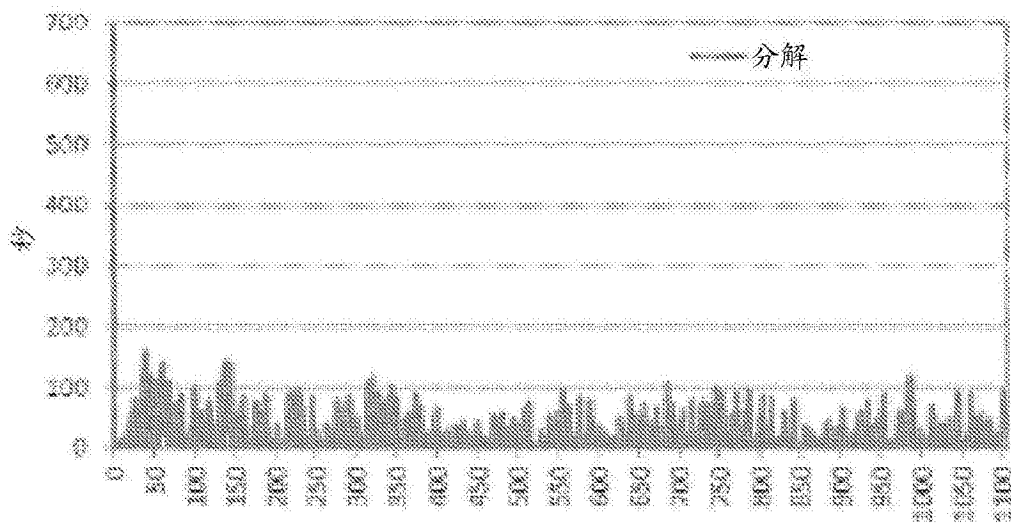


图7 (b)