软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2016,27(8):2068−2085 [doi: 10.13328/j.cnki.jos.005052]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

分布式大数据不一致性检测*

李卫榜, 李战怀, 陈 群, 杨婧颖, 姜 涛

(西北工业大学 计算机科学学院,陕西 西安 710072)

通讯作者: 李卫榜, E-mail: wbli2003@163.com

摘 要: 关系数据库中可能存在数据不一致性现象,关系数据库数据质量的一个主要问题是存在违反函数依赖情况.为找出不一致数据,需要进行函数依赖冲突检测.集中式数据库中可以通过 SQL 技术检测不一致情况,尽管检测效率不高;而分布式环境下不一致性检测更富有挑战性,不仅需要考虑数据的迁移,检测任务如何分配也是一个难题.在大数据背景下,上述问题更加突出.提出了一种分布式环境单函数依赖不一致性检测方法,给出了不一致性检测响应时间代价模型.为减少数据迁移量和响应时间,基于等价类对待检测数据进行预处理.由于分布式环境不一致性检测问题为 NP-hard 问题,多项式时间内难以得到最优解,给出了代价模型的多项式时间 3/2-近似最优解.提出了一种分布式环境多函数依赖不一致性检测方法,基于最小集合覆盖理论,通过一次数据遍历,对多个函数依赖进行并行批检测,同时考虑检测过程中的负载均衡等问题.在真实和人工数据集上的实验表明:相对于传统的检测方法以及基于 Hadoop 的 Naïve 方法,所提出的检测方法检测效率有明显的提升,且扩展性能良好.

关键词: 函数依赖;不一致性;冲突检测;分布式数据;大数据

中图法分类号: TP311

中文引用格式: 李卫榜,李战怀,陈群,杨婧颖,姜涛.分布式大数据不一致性检测.软件学报,2016,27(8):2068−2085. http://www.jos.org.cn/1000-9825/5052.htm

英文引用格式: Li WB, Li ZH, Chen Q, Yang JY, Jiang T. Inconsistency detection in distributed big data. Ruan Jian Xue Bao/Journal of Software, 2016,27(8):2068−2085 (in Chinese). http://www.jos.org.cn/1000-9825/5052.htm

Inconsistency Detection in Distributed Big Data

LI Wei-Bang, LI Zhan-Huai, CHEN Qun, YANG Jing-Ying, JIANG Tao

(College of Computer Science, Northwestern Polytechnical University, Xi'an 710072, China)

Abstract: Data inconsistency may exist in relational database. One major problem of data quality in relational database is functional dependency violation. To find out inconsistent data in a relational database, people need to detect the functional dependency violations. It is easy to detect dependency violations in centralized databases via SQL-based techniques, although the detection efficiency is not high. However, it is far more challenging to check inconsistencies in distributed databases, not only data shipment needs to be considered, but also the distribution of detecting tasks is a conundrum. These problems are more prominent with big data. This paper proposes a novel single functional dependency inconsistency detection approach in distributed big data, and provides a cost model of inconsistency detection. To reduce data shipment and response time, distributed data are pretreated based on equivalence class. Considering that the inconsistency detection problem is NP-hard, that is impossible to find an optimal solution in polynomial time, this work provides a 3/2-approximate optimal solution. A multiple functional dependencies detection approach is developed for distributed big data based on

* 基金项目: 国家重点基础研究发展计划(973)(2012CB316203); 国家自然科学基金(61472321, 61332006, 61502390); 国家高技术研究发展计划(863)(2015AA015307); 西北工业大学基础研究基金(3102014JSJ0005, 3102014JSJ0013)

Foundation item: National Program on Key Basic Research Project of China (973) (2012CB316203); National Natural Science Foundation of China (61472321, 61332006, 61502390); National High-Tech R&D Program of China (863) (2015AA015307); Basic Research Fund of Northwestern Polytechnical University of China (3102014JSJ0005, 3102014JSJ0013)

收稿时间: 2015-08-07; 修改时间: 2015-11-25; 采用时间: 2016-02-23; jos 在线出版时间: 2016-03-16

CNKI 网络优先出版: 2016-03-17 09:57:17, http://www.cnki.net/kcms/detail/11.2560.TP.20160317.0957.003.html

the minimal set cover theory. This approach allows detecting multiple functional dependencies violations in parallel after one-time data traversal，and it also incorporates load balancing in the detecting process. Experiments on real-world and generated datasets demonstrate that compared with previous detection methods and Naïve method based on Hadoop framework, the presented approach is more efficient and is scalable on big data.

Key words: functional dependency; inconsistency; violation detection; distributed data; big data

数据质量是数据管理中最重要的问题之一[1].数据质量的一个重要内容是不一致性检测.不一致数据普遍存在却并非显而易见的,使用不一致数据可能导致错误的决策,甚至会付出昂贵的代价.

在实际应用中,为提高数据质量,通常需要找出不一致数据,也就是对数据进行不一致性检测.对一个集中式数据库来说,不一致性检测较为容易,如对集中式数据库进行函数依赖冲突检测,可以使用一种基于 SQL 技术的检测方法[2].然而在现实中,一个关系表的数据可能被切分,并分布在不同的站点上[2].

例 1:考虑如下一个关系表 STUDENT(id,SNo,SName,Dept,CNo,CName,CGrade,Instr,Office),每一个学生元组包含了学号、姓名、院系、课程编号、课程名称、课程成绩、任课教师、办公室编号等内容.这里,ID 是关系表 STUDENT 的主键,STUDENT 的一个实例 D_0 见表 1.

Table 1 STUDENT relation D_0

表 1 STUDENT 实例 D_0

ID	SNo	SName	Dept	CNo	CName	CGrade	Instr	Office
1	5283	Susan	LifeScience	101	Economics	B	J. Fred	A10
2	3235	Dave	Automation	202	Ecology	A	P. Smith	A12
3	7689	Ross	LifeScience	102	Zoology	D	L. James	B05
4	3067	Sam	ComputerScience	203	Algebra	C	J. Allen	D01
5	2312	Joe	Automation	301	Geometry	B	L. Muller	B06
6	7684	Bob	LifeScience	203	Sociology	A	C. Brown	C09
7	5036	Smith	LifeScience	302	Mathematics	C	J. White	D13
8	5283	Steven	Automation	101	Economics	A	C. Owen	A10
9	4356	Philip	ComputerScience	301	Philosophy	A	B. Paul	A09
10	5628	Jack	ComputerScience	202	Ecology	C	P. Smith	B10

为了检测不一致性,在关系表 STUDENT 上定义了如下几个 FD 作为数据质量规则:

- φ_1:SNo→SName
- φ_2:CNo→CName
- φ_3:CNo,SNo→Instr
- φ_4:Instr→Office

这里,φ_1 声明了学号唯一决定学生姓名,φ_2 声明了课程编号唯一决定课程名称,φ_3 表示课程编号和学号唯一确定任课教师,φ_4 表示任课教师唯一确定办公室编号.

假定想要找到 D_0 中不一致的数据,即 D_0 中至少违反一个函数依赖的元组.这里,用 t_i 表示 D_0 中 id 为 i 的元组,则在 D_0 中存在冲突的元组有 $t_1,t_2,t_4,t_5,t_6,t_8,t_9,t_{10}$.具体来说,元组 t_4 和 t_6 违反函数依赖 φ_2:元组 t_4 和 t_6 课程编号相同,课程名称不同.与此类似,元组 t_5 和 t_9 违反函数依赖 φ_2.

在集中式数据库中,要检测 D_0 中存在的冲突元组,可以使用基于 SQL 的技术.对于每一个函数依赖,编写一套相应的 SQL 语句,在数据库中执行该语句,即可检测违反该函数依赖的元组[2].而如果 D_0 被水平或者垂直切分,各数据块分布在不同的站点.这种检测方法无法直接使用,通常需要进行数据迁移.

考虑前面提到的表 1,其数据被水平切分成 3 部分,分别见表 2~表 4,其中,分片 D_{H_1},D_{H_2},D_{H_3} 分别分布在站点 S_1,S_2 和 S_3 上.为检测元组违反函数依赖 φ_2 的情况,需要进行数据迁移.数据迁移的原则是,函数依赖左端属性值相同的元组迁移到相同的节点.迁移的方案有多种:(a) 从站点 S_3 迁移元组 t_9 和 t_{10} 到 S_1,从站点 S_3 迁移元组 t_4 到 S_2,从站点 S_1 迁移元组 t_8 到 S_2;(b) 从站点 S_1 迁移元组 t_2 和 t_5 到 S_3,从站点 S_3 迁移元组 t_4 到 S_2,从站点 S_1 迁移元组 t_8 到 S_2;等等.可以看出:与集中式环境不同,在分布式环境下,不一致检测通常需要数据迁移,因此,集中式环境下的检测方法不适用于分布式环境下的不一致性检测.

Table 2 D_{H_1}, a fragment of horizontally partitioned D_0

表 2 D_0的一个水平划分的片段 D_{H_1}

ID	SNo	SName	Dept	CNo	CName	CGrade	Instr	Office
2	3235	Dave	Automation	202	Ecology	A	P. Smith	A12
5	2312	Joe	Automation	301	Geometry	B	L. Muller	B06
8	2043	Steven	Automation	101	Economics	A	C. Owen	A10

Table 3 D_{H_2}, a fragment of horizontally partitioned D_0

表 3 D_0的一个水平划分的片段 D_{H_2}

ID	SNo	SName	Dept	CNo	CName	CGrade	Instr	Office
1	5283	Susan	LifeScience	101	Economics	B	J. Fred	A10
3	7689	Ross	LifeScience	102	Zoology	D	L. James	B05
6	7684	Bob	LifeScience	203	Sociology	A	C. Brown	C09
7	5036	Smith	LifeScience	302	Mathematics	C	J. White	D13

Table 4 D_{H_3}, a fragment of horizontally partitioned D_0

表 4 D_0的一个水平划分的片段 D_{H_3}

ID	SNo	SName	Dept	CNo	CName	CGrade	Instr	Office
4	3067	Sam	ComputerScience	203	Algebra	C	J. Allen	D01
9	4356	Philip	ComputerScience	301	Philosophy	A	B. Paul	A09
10	5628	Jack	ComputerScience	202	Ecology	C	P. Smith	B10

已有的关于分布式环境下完整性约束的研究主要关注条件函数依赖(conditional functional dependency,简称 CFD)的冲突检测,如 Fan 提出了一种分布式环境下检测条件函数依赖的方法[2],该检测方法主要是基于模式表(pattern tableau).与条件函数依赖不同,函数依赖由于缺乏可以利用的模式表,无法利用该技术进行不一致性检测,因此,该技术不适合分布式环境下函数依赖的不一致性检测.其他关于分布式环境下完整性约束的研究主要集中于定义在系统状态上的约束[3]或者是约束可以在本地进行验证的情况[4,5].

本文的主要贡献如下:

(1) 给出了分布式环境下单个函数依赖不一致性检测响应时间代价模型,基于该模型,将最小化响应时间问题划归为整数规划问题.由于分布式环境不一致性检测任务分配问题为 NP-难问题,无法在多项式时间得到最优解,本文基于该模型给出了 3/2-近似最优检测任务分配算法.

(2) 研究了多个函数依赖进行批量检测的问题,总结了可以进行批量检测的多个函数依赖的特征,并给出了分布式环境下多个函数依赖批量并行检测的算法.

(3) 基于真实和人工数据集,通过实验验证了本文提出的单个函数依赖不一致性检测算法和多个函数依赖批量检测算法,并进行了对比分析.

本文使用真实及人工大数据集,基于 Hadoop 平台对提出的检测方法进行了性能检测,并与传统的集中式方法和基于 Hadoop 的 Naïve 方法进行了比较.实验结果表明:本文的方法在数据规模和分片数量方面扩展性良好,在减少响应时间和数据迁移方面效果明显.

1 预备知识

1.1 函数依赖

函数依赖可以看作是定义在关系上的完整性约束.假定 R 是一个关系模式,在其上定义了一个函数依赖集合,$Attrs(R)=\{A_1,A_2,\ldots,A_m\}$定义了 R 上的属性集合,R 上的每一个属性 $A\in Attrs(R)$的域用 $Dom(A)$表示.R 的一个实例 I是一个元组的集合,其中每一个元组属于 $Dom(A_1)\times\ldots\times Dom(A_m)$.这里,用 $t[A]$表示元组 t 的属性 A 的值,用 $t[L]$表示 $Attrs(R)$中一组属性 L 在 t 上的投影.

定义 1. 函数依赖(简称 FD)是定义在关系 R 上的形如 $X\rightarrow Y$ 的表达式,这里,X, Y 是 $Attrs(R)$上的属性集合.函数依赖 $X\rightarrow Y$ 在关系 R 上成立,当且仅当对 R 的每一个实例,如果 R 中的任意两个元组有着相同的 X 属性值,

则必然有着相同的 Y 属性值.一个函数依赖 $X\rightarrow Y$ 是平凡的,如果 Y 是 X 的一个子集.平凡函数依赖对所有的关系实例都成立.如果一个函数依赖满足 $Y\cap X=\varnothing$,则这个函数依赖是非平凡的.

对于函数依赖 $\varphi:X\rightarrow Y$,X 是 φ 的左侧(left hand side,简称 LHS),而 Y 是 φ 的右侧(right hand side,简称 RHS).由于平凡函数依赖对所有关系实例都成立,本文中仅考虑非平凡函数依赖的不一致性检测问题.

1.2 函数依赖冲突

函数依赖作为完整性约束的一种,其冲突意味着违反了完整性约束.函数依赖冲突在现实中普遍存在,特别是在数据集成、数据融合以及 Web 数据抽取等应用中更为常见.

给定关系 R 的一个实例 D,函数依赖 $\varphi:A\rightarrow B$,将 D 上违反 φ 的冲突表示为 $Viot(\varphi,D)$.对于 D 中的每一个元组 t,如果存在一个元组 t' 满足 $t(A)=t'(A)$ 且 $t(B)\neq t'(B)$,说明元组 t 和 t' 违反了函数依赖 φ,因此有 $t\in Viot(\varphi,D)$.假定 Σ 为定义在 D 上的函数依赖的集合,则 $Viot(\Sigma,D)$ 表示 D 上的所有违反 Σ 中函数依赖的冲突.用符号 $Viot^{\Pi}(\varphi,D)$ 表示 $\Pi_A Viot(\varphi,D)$,这里,$\Pi_A Viot(\varphi,D)$ 是 $Viot(\varphi,D)$ 在属性 A 上的投影,R 中其他元素值用空值 $null$ 表示.与 $Viot(\varphi,D)$ 相比,$Viot^{\Pi}(\varphi,D)$ 包含数据更少.

一个关系 R 的任一水平分片与 R 有着相同的模式,因此,如果函数依赖 φ 是定义在 R 的实例 D 上面的,则 φ 同样也是定义在 D 的各个分片上的.这里,将 D 的任一水平分片 D_i 上违反 φ 的冲突定义为 $Viot(\varphi,D_i)$.

1.3 函数依赖冲突检测

给定关系 R 的实例 D,D 水平切分为 $(D_1,\ldots,D_n)$.假定 D 的每一个切分分布在一个单独的节点上,对于 $i\in[1,n]$,D_i 分布在节点 S_i 上.函数依赖 φ 是定义在 R 上的一个函数依赖,函数依赖 φ 的冲突检测问题是查找 $Viot^{\Pi}(\varphi,D)$.给定函数依赖 φ,φ 可以在本地检测当且仅当满足 $Viot^{\Pi}(\varphi,D)=\bigcup_{i\in[1,n]}Viot^{\Pi}(\varphi,D_i)$.

具有最小响应时间的函数依赖不一致性检测问题,其输入是一个函数依赖的集合 Σ 和水平切分的关系 R 的实例 D,响应时间满足以下条件:(a) 集合 Σ 中的函数依赖在数据迁移 U 之后可以本地检测;(b) 响应时间 RT 是最小的.然而找到一个最小响应时间的检测算法是不现实的,在水平切分情况下,具有最小响应时间的函数依赖不一致性检测问题是一个 NP-难问题[4].

2 分布式大数据不一致性检测

这一节探讨分布式环境下水平切分的关系数据不一致性检测问题.与集中式数据相比,分布式环境下函数依赖不一致性检测更具挑战性,通常需要进行数据的迁移.在进行不一致检测时,需要确定哪些元组需要迁移以及迁移到哪个节点.对于单个函数依赖来说,该问题是已经是非平凡的.根据前面的介绍,找出一个具有最小响应时间的检测算法是 NP-难的.

2.1 单个函数依赖冲突检测

- 算法 CetDetect.

首先,选择一个节点 S_j 作为函数依赖 φ 的执行节点,其他节点上的元组都迁移到节点 S_j;最后,函数依赖 φ 的冲突检测可以在节点 S_j 进行.这里选择元组数量分布最多的节点作为执行节点,可以减少数据迁移量及通信代价.算法 CetDetect 中,每个元组最多迁移一次.由于该算法将所有数据迁移到一个节点,在该节点进行不一致性检测,在数据规模特别大时,大量数据迁移到一个节点,检测任务由一个节点承担,导致负载严重不均衡,该节点很容易成为检测的瓶颈.

- 算法 $\text{SingleFdDet}_{\text{RT}}$.

算法 $\text{SingleFdDet}_{\text{RT}}$ 进行分布式环境大数据不一致检测时,分为以下几个步骤:(a) 数据预处理;(b) 任务分配;(c) 数据迁移;(d) 不一致性检测.

给定关系 R 的实例 D,D 水平切分为 $(D_1,\ldots,D_n)$.假定 D 的每一个切分分布在一个单独的节点上,对于 $\forall i\in[1,n]$,D_i 分布在节点 S_i 上.对任一节点 S_i,$i\in[1,n]$,算法首先对其上的数据 D_i 进行预处理,假定其数据预处理时

间为 $predeal(D_i)$.为保证不一致检测的准确性,预处理后的数据需要进行数据的迁移,也就是不一致性检测任务的分配.假定任一节点 S_i 迁入的数据为 $\Delta D_{i\text{-}in}$,迁出的数据为 $\Delta D_{i\text{-}out}$,则总的数据迁移量为

$$\Delta D = \sum_{i\in[1,n]} \Delta D_{i\text{-}in} = \sum_{i\in[1,n]} \Delta D_{i\text{-}out} .$$

也就是说,在数据迁移过程中,各节点总的迁入数据量与总的迁出数据量相同.假定对任一节点 $S_i, i\in[1,n]$,完成数据迁移后,其数据规模为 D_i',则有 $D_i' = D_i + \Delta D_{i\text{-}in} + \Delta D_{i\text{-}out}$.假定函数依赖 φ 不一致性检测总的响应时间为 $cost_{RT}(\varphi)$,这里定义不一致性检测响应时间代价模型如下:

$$cost_{RT}(\varphi) = \max_{i\in[1,n]} predeal(D_i) + \frac{1}{bw}\sum_{i\in[1,n]} \Delta D_{i\text{-}out} + \max_{i\in[1,n]} check(D_i') \tag{1}$$

其中,$predeal(D_i)$ 为各节点数据预处理耗时,n 为节点个数,bw 为网络带宽,φ 为待检测的函数依赖.

依据公式(1),函数依赖不一致性检测的响应时间取决于 3 部分:各节点数据预处理的最大耗时 $\max_{i\in[1,n]} predeal(D_i)$、数据迁移耗时 $\frac{1}{bw}\sum_{i\in[1,n]} \Delta D_{i\text{-}out}$ 以及数据重分布后各节点最大检测耗时 $\max_{i\in[1,n]} check(D_i')$.要想减少总的响应时间 $cost_{RT}(\varphi)$,就要考虑对以上几个阶段的响应时间进行优化.

不一致性检测的第 1 步是数据的预处理.数据预处理的主要目的是消除与待检测函数依赖属性值相关的冗余数据,其好处有两个方面:一是可以减少数据迁移量,特别是在冗余数据比较多的情况;二是可以提高检测效率.预处理一方面消除了冗余数据,另一方面,预处理后的数据更有利于不一致性检测.

定义 2. 关系 r 上基于函数依赖 $X\to Y$ 的属性集 $X\cup Y$ 将 r 中所有元组划分成不同的等价类,称为 r 在 $X\cup Y$ 上的一个划分,用 $\Pi_{X\cup Y}$ 表示.等价类划分的原则是同一个等价类内有着相同的 X 属性值,且去除冗余.

在数据预处理阶段,首先在各个节点并行扫描一遍本地数据,依据定义 2,将本地元组进行等价类的划分.进一步将划分好的等价类表示成键值对〈*key*,*value*〉的形式,用 $\Pi_{\langle X,Y\rangle}$ 表示,其中,以每个等价类中 X 属性值为 *key*,Y 的不同属性值集合为 *value*.这种键值对的表示形式对 X 和 Y 属性值是相同的,仅保留一组值,因此可以有效去除各个节点的冗余数据.经过预处理后的数据由于去除了冗余数据,因此可以减少数据迁移量和检测的工作量.以表 2 为例,对函数依赖 φ_2,根据定义 2,可以得到关系 r 在数据片段 D_{H_2} 上的属性集合 $CNo\cup CName$ 上的一个划分为

$$\Pi_{CNo\cup CName}=\{\{(202,Ecology)\},\{(301,Geometry)\},\{(101,Economics)\}\}.$$

用键值对表示为

$$\Pi_{\langle CNo,CName\rangle}=\{\langle 202,[Ecology]\rangle,\langle 301,[Geometry]\rangle,\langle 101,[Economics]\rangle\}.$$

对于各个节点得到的局部数据等价类,为并行检测函数依赖不一致性,需要进行数据迁移.这里使用哈希函数计算每个等价类的哈希值,哈希函数的输入为函数依赖 LHS 属性值.根据定义 2,同一等价类内的函数依赖 LHS 属性值相同,因此其哈希值也相同.也就是说,每一个等价类有唯一的哈希值.然后,将具有相同哈希值的等价类迁移到同一个执行节点,并进行等价类的合并.等价类合并过程中消除了全局的冗余数据,因此可以减少不一致性检测要处理的数据规模,提高检测效率.各个执行节点得到合并后的等价类后,根据同一等价类中包含的元组个数进行不一致性检测:如果等价类中元组个数为 1,说明不存在违反函数依赖的情况;如果等价类中元组个数大于 1,说明存在违反函数依赖的情况.

完成以上的数据等价类合并工作,得到初步预处理数据,下一步考虑在各个节点对得到的初步预处理数据进行分片.数据分片的目的是为了便于接下来的并行检测.数据分片的原则是,存在潜在冲突的元组数据分配到相同的分片内,这样可以保证不一致检测结果的准确性.

本文考虑利用散列函数对各节点初步预处理过的数据进行分片,假定每个节点上的预处理数据根据散列函数 μ 散列为 m 片,将预处理得到的键值对的 *key* 作为散列函数 μ 的输入,可以将 *key* 转换为 0 到 m–1 的整数值.根据各个键值对散列值的不同,散列函数将 D 的任一切分 D_i 分成 m 片,分别为 $H_i^1,\ldots,H_i^m$.对 $\forall k\in[1,m]$, H_i^k 内的键值对有着相同的散列值.这样可以将 D 分成 m 片,片内的键值对有着相同的散列值,如图 1 所示.

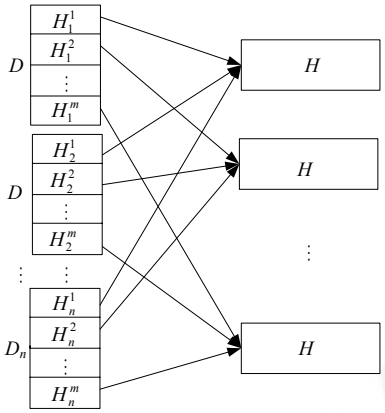


Fig.1 Partition of $D_1,\ldots,D_n$ by hash function

图 1 $D_1,\ldots,D_n$ 被散列函数切分

图 1 中,H^1 为散列值为 0 的键值对组成的块,H^2 为散列值为 1 的键值对组成的块,以此类推.对$\forall j\in[1,m]$,有:

$$H^j=H_1^j\cup\ldots\cup H_n^j.$$

这里考虑分片大小 m 的取值问题.m 取值的大小会影响数据分配时的负载均衡,m 取值越小,则分片的粒度越大;m 取值越大,则分片的粒度越小.如果 $m<n$,说明分片数量<节点数量,在进行任务分配时,会出现有些节点没有负载,而有些节点的负载较大的情况.如果分片过多,理论上有利于负载分配更加均衡,但是进行负载分配也需要耗时.为提高不一致性检测的并行度,使得每个节点都不会因为没有分配检测任务而空闲,本文中散列分片数量不少于节点个数,即 $m\geqslant n$.

在得到散列及合并后的数据分片 $H^1,\ldots,H^m$ 后,为对数据分片执行不一致性检测,需要将数据片分配到执行节点,接下来考虑 m 个数据片的任务分配情况,该问题为多处理机调度问题.

问题 1(多处理机调度问题). 任给有穷的任务集 A,每一个任务 $a\in A$ 的响应时间为 $t(a)\in\mathbf{Z}^+$,处理机数目为 $n\in Z^+$,任务截止时间为 $D\in\mathbf{Z}^+$,其中,$\mathbf{Z}^+$为正整数集合.判定:

是否能将任务集 A 划分成 n 个不相交的集合 $A=A_1\cup\ldots\cup A_n$,使得: $\max\left\{\sum_{a\in A_i}t(a):1\leqslant i\leqslant n\right\}\leqslant D$.

定理 1. 多处理机调度问题为 NP-完全问题[6].

证明:利用限制法证明.考虑该问题的一种特例,当 $m=2$ 且 $D=\left\lceil\frac{1}{2}\sum_{a\in A}t(a)\right\rceil$ 时,该问题可以限制成划分问题.由文献[6]知,划分问题为 NP-完全问题.因此,多处理机调度问题为 NP-完全问题. □

本文中需要检测的数据块为 m 个,节点个数为 n 个,对每个 $i=1,2,\ldots,m$ 和每个 $j=1,2,\ldots,n$,第 i 个数据块在第 j 个节点上检测耗时为 t_{ij}.依据不一致性检测响应时间代价模型 $cost_{RT}(\varphi)$,要想减少总的响应时间,应当最小化数据迁移时间 $\frac{1}{bw}\sum_{j\in[1,n]}\Delta D_{j\text{-}out}$ 和不一致性检测时间 $\max_{j\in[1,n]}check(D_j')$.也就是在对所有待检测数据块进行任务分配后,数据迁移耗时与最后检测完成的节点耗时之和最小.这里,用 x_{ij} 表示在第 j 个节点上处理第 i 个数据块,则当第 i 个数据块分配给第 j 个节点时,$x_{ij}=1$;否则,$x_{ij}=0$.该问题可以转化为整数规划问题:

- 目标函数:

$$\min\ t+\frac{1}{bw}\sum_{j=1}^{n}\Delta D_{j\text{-}out}.$$

- 约束条件:

$$\sum_{j=1}^{n} x_{ij} = 1,\ 1 \leqslant i \leqslant m;$$

$$\forall j \in [1,n],\ t \geqslant \sum_{i=1}^{m} x_{ij} t_{ij};$$

$$\forall j \in [1,n],\ \Delta D_{j\text{-}out} = D_j - \sum_{i=1}^{m} x_{ij} H_j^i,\ 1 \leqslant i \leqslant m;$$

$$x_{ij} \in \{0,1\},\ 1 \leqslant i \leqslant m,\ 1 \leqslant j \leqslant n.$$

可以将上述整数规划问题关于 x_{ij} 的约束进行松弛,松弛后的约束条件为 $0 \leqslant x_{ij} \leqslant 1, 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n$.这样,可以将上述整数规划问题松弛为线性规划问题.

由于该问题为 NP-完全问题,在多项式时间内难以得到问题的精确解,本文给出一种近似最优的综合考虑数据迁移量与负载均衡的算法.

算法 1. Sorted-LoadBalance.

Input:机器集合 $M=\{M_i\}, i\in[1,n]$;任务集合 $A=\{A_1,\ldots,A_m\}$.

Output: $A(1),A(2),\ldots,A(n)$.

```
1.  for each i∈[1,n] do
2.     T_i=0, A_i={};
3.  end for;
4.  Desc(A);   //依据 A 中任务耗时 t_j 的大小对 A 中任务降序排列
5.  假定 t_1≥t_2≥…≥t_m;
6.  for each j∈[1,m] do
7.     if 机器 M_i 任务耗时达到最小值
8.           if |H_i^k|≥|H_l^k|, i,l∈[1,n]且 i≠l,任务 j 对应的数据块为 H^k
9.              将任务 j 分配给机器 M_i;
10.          end if;
11.       A(i)←A(i)∪j;
12.       T_i←T_i+t_j;
13.       end if;
14.  end for.
```

引理 1. 给定包含 m 个任务的集合 $A=\{A_1,\ldots,A_m\}$,n 个机器节点,T^*为响应时间最优值,t_j 为任务 A_j 的响应时间,$j\in[1,m]$,则有:$T^* \geqslant \frac{1}{n}\sum_{j=1}^{m} t_j$.

证明:m 个任务的总的处理时间为 $\sum_{j=1}^{m} t_j$,由于 n 个机器节点中至少有一个机器节点处理的任务量不少于总任务量的 $\frac{1}{n}$,因此,该机器节点的响应时间 $\geqslant \frac{1}{n}\sum_{j=1}^{m} t_j$;而 T^* 为所有机器节点响应时间的最大值,因此有 $T^* \geqslant \frac{1}{n}\sum_{j=1}^{m} t_j$. □

引理 2. 给定包含 m 个任务的集合 $A=\{A_1,\ldots,A_m\}$,n 个机器节点,T^*为响应时间最优值,t_j 为任务 A_j 的响应时间,$j\in[1,m]$,如果 $m>n$,则有 $T^* \geqslant 2t_{n+1}$.

证明:对于根据执行时间大小降序排序后的任务序列,其前 $n+1$ 个任务中,每一个需要的响应时间至少为 t_{n+1}.任务个数为 $n+1$ 个,而机器节点个数为 n 个,根据鸽笼原理,必然有一个机器节点分得的任务个数为 2 个,因此,该节点的响应时间 $\geqslant 2t_{n+1}$;而 T^* 为所有机器节点响应时间的最大值,因此有 $T^* \geqslant 2t_{n+1}$. □

定理 2. 当 $m>n$ 时,算法 Sorted-LoadBalance 的响应时间 $T\leqslant\frac{3}{2}T^*$[7].

证明:并行执行多个任务时,如果负载最大的机器 M_i 的负载为一个任务,则调度是最优的.假定节点 S_i 负载至少为 2 个任务,设 t_j 为分配给 S_i 的最后一个任务的响应时间,这里,$j\geqslant n+1$.因为前 n 个任务分配给不同的节点,由引理 2,

$$t_j\leqslant t_{n+1}\leqslant\frac{1}{2}T^* \tag{2}$$

在将响应时间为 t_j 的任务分配给节点 S_i 之前,S_i 的负载是所有节点里最小的,此时,S_i 的负载响应时间为 T_i-t_j,其中,T_i 为在将响应时间为 t_j 的任务分配给节点 S_i 之后,S_i 的所有负载的响应时间.因此,所有节点负载的响应时间均$\geqslant T_i-t_j$.所有节点的负载响应时间相加得到 $\sum_{k=1}^{n}T_k\geqslant n(T_i-t_j)$,进一步得到 $T_i-t_j\leqslant\frac{1}{n}\sum_{k=1}^{n}T_k$.

由于所有节点负载的任务响应时间之和与全体任务的响应时间之和相等,因此有: $\sum_{k=1}^{n}T_k=\sum_{j=1}^{m}t_j$. 由引理 1,

$$T_i+t_j\leqslant T^* \tag{3}$$

公式(2)和公式(3)相加,可得:$T_j\leqslant\frac{3}{2}T^*$. □

引理 3. 给定函数依赖φ,哈希函数μ, $D_i=\bigcup_{j\in[1,n]}H_i^j$,则有 $Viot^{\Pi}(\varphi,D)=\bigcup_{j\in[1,n]}Viot^{\Pi}(\varphi,\bigcup_{i\in[1,n]}H_i^j)$.

根据该引理,为计算 $Viot^{\Pi}(\varphi,D)$,可以通过为任一数据块 $H^j,j\in[1,n]$,选择一个执行节点 S_j,在该节点上计算 $Viot^{\Pi}(\varphi,\bigcup_{i\in[1,n]}H_i^j)$,即可得到函数依赖φ的所有冲突元组.算法 $\text{SingleFdDet}_{\text{RT}}$ 就是基于这种思想.

算法 $\text{SingleFdDet}_{\text{RT}}$ 如算法 2 所示.

算法 2. $\text{SingleFdDet}_{\text{RT}}$.

Input: Functional Dependency $\varphi:X\rightarrow Y$; $D=(D_1,\ldots,D_n)$.

Output: $Viot^{\Pi}(\varphi,D)$.

/* 在任一节点 S_i,并行执行以下程序: */

1. $D_{i\text{-}pd}\leftarrow predeal(D_i)$; //数据预处理
2. **for each** $k\in[1,m]$ **do**
3. **for each** $\langle key,[value]\rangle\in D_{i\text{-}pd}$ **do**
4. **if** $\mu(key)==k-1$ //$\mu(key)$为散列函数
5. $H_i^k\leftarrow H_i^k\cup\langle key,[value]\rangle$;
6. **end for**;
7. $H^k\leftarrow H^k\cup H_i^k$;
8. **end for**;
9. $H\leftarrow\{H^1,\ldots,H^m\}$;
10. $H'\leftarrow Desc(H)$; //依据数据规模大小降序排列
11. 假定 $H'=\{H^{1\prime},\ldots,H^{m\prime}\}$,其中,$|H^{1\prime}|\geqslant\ldots\geqslant|H^{m\prime}|$
12. **for each** $j\in[1,m]$ **do**
13. **if** $H'\neq\{\}$
14. **if** 节点 S_i 分配任务最少
15. $A(i)\leftarrow A(i)\cup H^{j\prime}$; //$A(i)$为 S_i 负载
16. $H'\leftarrow H'\backslash H^{j\prime}$;
17. **end if**;
18. **end if**;

19. **end for**;

20. **return** $Viot^{\Pi}((X \to Y), \bigcup_{i\in[1,n]} A(i))$.

根据函数依赖的定义,对函数依赖 $X \to Y$,当出现多个元组 X 属性值相同、Y 属性值不同的情况,表明函数依赖冲突存在.在算法 2 中,经过散列和数据迁移后,任一执行节点上的数据块内的元组有着相同的散列值.由于散列函数本身是稳定的,同一散列函数,相同的输入,其输出也是相同的,因此对于有着相同 X 属性值的元组来说,其散列后的值是相同的.因此在算法 2 中,有着相同 X 属性值的元组会被迁移到相同的节点,这保证了算法结果的正确性.

2.2 多个函数依赖的检测算法

下面给出分布式环境下检测水平切分数据多个函数依赖冲突的算法.

- 算法 SuccDetect

该算法检测多个函数依赖的冲突时,顺序对多个函数依赖中的每一个函数依赖进行检测,对单个函数依赖进行检测的时候,使用前面提到的算法 $\text{SingleFdDet}_{\text{RT}}$.算法 SuccDetect 对多个函数依赖不一致性进行检测的时候,采用流水线处理方式,在检测完一个函数依赖的不一致性之后,接着检测下一个函数依赖的不一致性,直到所有函数依赖全部检测完为止.

算法 SuccDetect 在对多个函数依赖不一致性进行检测的时候,会产生过多的网络负载:检测每一个函数依赖不一致性的时候,通常都要进行数据迁移,同一个元组,在检测不同的函数依赖不一致性的时候可能会被多次迁移,这样就产生了较多的负载.由于对每个函数依赖进行检测的时候都要对全部数据扫描一遍,而数据的扫描是十分耗时的,特别是在数据量比较大的情况下,因此算法 SuccDetect 的响应时间比较长.

函数依赖冲突检测是一个耗时的工作,特别是大数据的背景下,扫描数据的时间开销比较大,如果能够一次扫描数据库,实现对多个函数依赖不一致性的检测,则可以有效提高检测效率.算法 $\text{MultiFdDet}_{\text{RT}}$ 可以对多个函数依赖冲突进行批量检测,检测时只需扫描一次数据库.

- 算法 $\text{MultiFdDet}_{\text{RT}}$

该算法利用输入的多个函数依赖的 LHS 部分的结构特点,通过对数据的一次扫描,实现对多个函数依赖不一致性的批量检测,可以有效减少函数依赖不一致性检测的响应时间.

定义 2. 关系 r 上基于 $LHS(\varphi_1)\cap\ldots\cap LHS(\varphi_n)$不为空的函数依赖集合$\Phi=\{\varphi_1,\varphi_2,\ldots,\varphi_n\}$的属性集 $Attr(\Phi)=LHS(\varphi_1)\cup\ldots\cup LHS(\varphi_n)\cup RHS(\varphi_1)\cup\ldots\cup RHS(\varphi_n)$将 r 中所有元组分成不同的等价类,称为 r 在 $Attr(\Phi)$上的一个划分,用$\Pi_{Attr(\Phi)}$表示.等价类划分的原则是同一个等价类内有着相同的 $LHS(\varphi_1)\cap\ldots\cap LHS(\varphi_n)$属性值,且去除冗余.

算法 $\text{MultiFdDet}_{\text{RT}}$ 首先依据定义 2 并行地对各个节点的数据进行等价类划分,进一步将划分好的等价类转换成$\langle key,value\rangle$键值对的形式,key 值为 $LHS(\varphi_1)\cap\ldots\cap LHS(\varphi_n)$属性值,$value$ 为 $Attr(\Phi)\backslash LHS(\varphi_1)\cap\ldots\cap LHS(\varphi_n)$去重后的属性值.进行数据重分布时,以键值对的 key 值为散列函数的输入,每一个键值对对应一个散列值,然后,将有着相同散列值的键值对迁移到同一个执行节点.由于有着相同散列值的键值对包含了所有的潜在违反多个 FD 的情况,因此可以在各个节点并行进行多个函数依赖的不一致性检测.

与直接散列相比,先进行等价类的划分,再进行数据迁移和不一致性检测,其优势在于:

1) 有效消除冗余数据,减少数据迁移量;
2) 冗余数据的消除可以减少数据迁移时间,同时将潜在冲突的元组划分到相同的等价类,可以减少不一致性检测的响应时间;
3) 冗余数据的消除可以减少不一致检测时的数据规模,提高检测效率;
4) 一个等价类内的元组仅需散列一次,无需每个元组单独散列一次,可以减少总的散列时间.

数据预处理后,在各个节点以键值对的 key 值为散列函数的输入计算其散列值,将 key 值转换为 $0\sim\tilde{m}1$ 的整数值.与算法 $\text{SingleFdDet}_{\text{RT}}$ 类似,根据各个键值对的 key 值对应散列值的不同,散列函数将 D 的任一切分 D_i 分成 m 块,分别为 $H_i^1,\ldots,H_i^m$.对$\forall k\in[1,m]$, H_i^k 内的元组有着相同的散列值,这样可以将 D 分成 m 块,每一块内

的元组有着相同的散列值.算法 MultiFdDet$_{RT}$ 如算法 3 所示.

算法 3. MultiFdDet$_{RT}$.

Input:函数依赖集合$\Sigma=\{\varphi_1,\ldots,\varphi_n\}$;$D=(D_1,\ldots,D_n)$.

Output:$Viot^{\Pi}(\varphi_1,D),Viot^{\Pi}(\varphi_2,D),\ldots,Viot^{\Pi}(\varphi_n,D)$.

1. Compute $\sigma(\Sigma)$; //计算Σ中函数依赖 LHS 部分交集

/* 在任一节点 S_i,并行执行以下程序: */

2. $D_{i\text{-}pd}\leftarrow predeal(D_i)$; //数据预处理
3. **for each** $k\in[1,m]$ **do**
4. **for each** $\langle key,[value]\rangle\in D_{i\text{-}pd}$ **do**
5. **if** $\mu(key)==\bar{k}1$ //$\mu(key)$为散列函数
6. $H_i^k\leftarrow H_i^k\cup\langle key,[value]\rangle$;
7. **end for**;
8. $H^k\leftarrow H^k\cup H_i^k$;
9. **end for**;
10. $H\leftarrow\{H^1,\ldots,H^m\}$;
11. $H'\leftarrow Desc(H)$; //依据数据规模大小降序排列
12. 假定 $H'=\{H^{1\prime},\ldots,H^{m\prime}\}$,其中,$|H^{1\prime}|\geqslant\ldots\geqslant|H^{m\prime}|$
13. **for each** $j\in[1,m]$ **do**
14. **if** $H'\neq\{\}$
15. **if** 节点 S_i 分配任务最少
16. $A(i)\leftarrow A(i)\cup H^{j\prime}$; //$A(i)$为 S_i 负载
17. $H'\leftarrow H'\backslash H^{j\prime}$;
18. **end if**;
19. **end if**;
20. **end for**;
21. **return** $Viot^{\Pi}(\varphi_1,\bigcup_{i\in[1,n]}A(i)),Viot^{\Pi}(\varphi_2,\bigcup_{i\in[1,n]}A(i)),\ldots,Viot^{\Pi}(\varphi_n,\bigcup_{i\in[1,n]}A(i))$.

显然,$LHS(\varphi_1)\cap\ldots\cap LHS(\varphi_n)$为 $LHS(\varphi_1),\ldots,LHS(\varphi_n)$的子集.根据散列函数的定义,有着相同$\sigma(\Sigma)$属性值的元组被散列到相同的数据块.对于$\forall i\in[1,n]$,$D$ 中有着相同 $LHS(\varphi_i)$属性值的元组是有着相同$\sigma(\Sigma)$属性值的元组集合的子集合.因此经过散列函数$\mu()$散列后,对集合Σ中所有的函数依赖,其存在潜在冲突的元组都被散列到相同的数据块中,这保证了算法 MultiFdDet$_{RT}$ 不一致性检测结果的准确性.接下来在各个节点统计各散列子数据块的元组个数,节点之间交换统计数据,汇总后根据前面算法 SingleFdDet$_{RT}$ 中提到的执行节点分配策略,将数据块分配到相应的执行节点,各节点并行进行数据的迁移.数据迁移完成后,函数依赖集合Σ中所有函数依赖存在潜在冲突的元组都被分配到相同的节点,根据引理 2,对$\forall i\in[1,n]$,为计算 $Viot^{\Pi}(\varphi_i,D)$,可以通过在任一执行节点 S_j 计算 $Viot^{\Pi}\left(\varphi,\bigcup_{i\in[1,n]}A(i)\right)$ 最后将各执行节点的检测结果汇总得到.这样,算法 MultiFdDet$_{RT}$ 通过一次数据遍历和一次数据迁移,实现了对分布式环境多个函数依赖的不一致性检测.

引理 4. 给定关系 R 的一个实例 D,假定存在函数依赖 $X\rightarrow Y,X'\rightarrow Y,X\subset X'$,$\forall t_i,t_j\in D$,如果 $t_i[X']=t_j[X']$且 $t_i[Y]\neq t_j[Y],i\neq j$,则必然有 $t_i[X]=t_j[X]$且 $t_i[Y]\neq t_j[Y]$.

证明:已知 $t_i[X']=t_j[X']$且 $t_i[Y]\neq t_j[Y],i\neq j$.又 $X\subset X'$,即,属性集合 X 为 X'的真子集.对于元组 t_i,t_j 来说,其属性集合 X'的属性值相同,去掉属性集合 $X'\backslash X$,剩余属性集合为 X.假设 $t_i[X]\neq t_j[X]$,由 $X\subset X'$可知,$t_i[X']=t_i[X]\cup t_i[X'\backslash X]$,$t_j[X']=t_j[X]\cup t_j[X'\backslash X]$.由 $t_i[X]\neq t_j[X]$可知,$t_i[X]\cup t_i[X'\backslash X]\neq t_j[X]\cup t_j[X'\backslash X]$,即 $t_i[X']\neq t_j[X']$,与已知 $t_i[X']=t_j[X']$矛盾.故假设不成立,$t_i[X]=t_j[X]$成立. □

算法 $MultiFdDet_{RT}$ 在检测多个函数依赖的不一致性时,根据多个函数依赖的 LHS 部分以及 RHS 部分所包含的属性情况的不同,实现了中间计算结果的共享,可以有效减少检测的计算量,提高检测效率.前面提到,多个函数依赖不一致性检测中,散列函数的 *key* 为 $LHS(\varphi_1)\cap\ldots\cap LHS(\varphi_n)$属性值,*value* 为 $Attr(\Phi)\backslash LHS(\varphi_1)\cap\ldots\cap LHS(\varphi_n)$去重后的属性值.在数据迁移之后,可能存在冲突的元组都被散列到相同的节点.根据待检测多个函数依赖的 LHS 部分及 RHS 部分所包含的属性情况,检测过程中可以实现如下的中间计算结果共享:

(1) 待检测的多个函数依赖 LHS 部分与 $LHS(\varphi_1)\cap\ldots\cap LHS(\varphi_n)$相同,则在不一致性检测过程中,所有节点在处理各元组数据时,这些函数依赖的 LHS 部分属性值为⟨*key*,*value*⟩键值对的 *key* 值,因此无需计算.只需根据其 RHS 所包含属性情况,对元组所对应的键值对的 *value* 部分进行切分,进而得到相关的属性值,合并去重后可以得到各函数依赖的冲突元组.

(2) 待检测的多个函数依赖 LHS 部分相同且与 $LHS(\varphi_1)\cap\ldots\cap LHS(\varphi_n)$不同,检测时,多个函数依赖的 LHS 部分属性值的计算结果可以共享,即,只需计算一遍,无需每个 FD 都单独计算一遍.每个待检测 FD 的 RHS 部分可由⟨*key*,*value*⟩键值对的 *value* 部分切分得到.

(3) 待检测的多个函数依赖 RHS 部分相同,则根据⟨*key*,*value*⟩键值对进行计算时,RHS 部分的属性值计算中间结果可以共享,无需每个 FD 都单独计算一遍,因此可以有效提高计算效率.

(4) 如果待检测的多个函数依赖存在 LHS 包含的情况,假定 LHS 存在包含情况的多个待检测函数依赖中 LHS 包含属性最多的函数依赖为φ,由引理 4,违反该函数依赖的元组必然违反 LHS 为其子集合的其他函数依赖,故在不一致性检测时,违反φ的元组必然违反 LHS 部分为 $LHS(\varphi)$子集合的其他函数依赖,因此对于这些函数依赖来说,违反φ的元组无需重复验证.

算法 $MultiFdDet_{RT}$ 的前提是给定的多个函数依赖 LHS 部分的交集不为空.然而通常情况下,给定一个函数依赖集合Σ,其 LHS 部分不一定有公共属性,这种情况下不能直接使用算法 $MultiFdDet_{RT}$.为了将算法 $MultiFdDet_{RT}$ 应用到多个函数依赖的不一致性检测,考虑将函数依赖集合Σ中的多个函数依赖进行分组.分组的依据是组内的函数依赖 LHS 部分有公共属性,同一分组内的函数依赖可使用前面提到的批量并行检测算法 $MultiFdDet_{RT}$ 进行不一致性检测,从而提高分布式环境下函数依赖不一致性检测的效率,同时减少网络负载.

例 2:以表 1 中实例 D_0 上定义的函数依赖集合$\Sigma=\{\varphi_1,\varphi_2,\varphi_3\}$为例,函数依赖$\varphi_2$和$\varphi_3$的 LHS 部分存在公共属性 *CNo*,而$\varphi_1$与$\varphi_2$和$\varphi_3$在 LHS 部分没有公共属性,因此可以将$\Sigma=\{\varphi_1,\varphi_2,\varphi_3\}$分成两组:$\{\varphi_1\}$和$\{\varphi_2,\varphi_3\}$,组内的函数依赖可以使用批量并行检测算法 $MultiFdDet_{RT}$ 进行不一致性检测,有效提高检测效率.

由于对函数依赖进行不一致性检测的时候需要扫描全部数据,在大数据背景下,扫描全部数据是一个十分耗时的工作.而前面提到,算法 $MultiFdDet_{RT}$ 在对多个函数依赖进行不一致性检测的时候只需要对全部数据进行一次扫描,因此在对函数依赖集合Σ中的多个函数依赖进行分组的时候,要满足组内的函数依赖 LHS 部分有公共元素,而且分组的个数尽可能地少,这个问题可以划归为最小集合覆盖(minimal set cover,简称 MSC)问题.根据文献[6],最小集合覆盖问题是一个 NP-完全问题,不存在一个多项式时间的解,除非 *P*=*NP*,但是可以在多项式时间内得到该问题的近似解.算法 4 是一个最小集合覆盖问题的贪心算法.

算法 4. GreedyMSC.

Input:集合 $X=\{1,2,\ldots,n\}$;X 的子集合构成的集合 $F=(S_1,S_2,\ldots,S_n)$.

Output:X 的具有最小势的集合覆盖 C.

```
1.  Initialize: C←{},U←X;
2.    while U≠{} do
3.      Select S where S∈F and S is a set such that |S∩U| is maximized;   //S 覆盖 U 中最多的元素
4.        C←C∪{S};   //将 S 并入覆盖 C
5.        U←U\S;   //从 U 中移除被 S 覆盖的所有元素
6.  return C.
```

算法 4 初始化时 $U\leftarrow X$,然后逐步从 U 中减去新增到覆盖 C 中的集合中的元素,直到 C 包含 X 中全部元素

为止.算法的时间复杂度主要取决于 F 的情况.然而,贪心算法并不总是能得到最优解.

例 3:假定 X={1,2,3,4,5,6,7,8,9,10},F=(S_1,S_2,…,S_5),S_1={1,2,5,7,9,10},S_2={1,2,3,4,5},S_3={3,4,8},S_4={4,6,8},S_5={6,7,8,9,10}.贪心算法在执行的时候首先选择包含最多元素的集合 S_1,剩余的集合在去掉 S_1 中元素后为{3,4},{3,4,8},{4,6,8},{6,8}.要想最终得到 X 的一个覆盖,至少还要选取 S_2 和 S_3 中的至少一个以及 S_4 和 S_5 中的至少一个.因此,最终选取的集合 X 的覆盖至少包含 F 中的 3 个元素.而该问题的最优解可以通过选择 S_2 和 S_5 共 2 个子集合得到.

为得到该问题的最优解,可以考虑将该问题表述为如下的(0-1)-规划问题:

任给集合 S,S 的子集合 S_1,S_2,…,S_n,求解以下(0-1)-规划:

求最小值 $c(x)=x_1+x_2+\ldots+x_n$,

满足条件 $S_1\times x_1\cup S_2\times x_2\cup\ldots\cup S_n\times x_n=S;x_1,x_2,\ldots,x_n\in\{0,1\}$.

令 opt 为目标函数最优值,则 $opt\leqslant n$.对于子集合 $I\subseteq\{1,2,\ldots,n\}$,令 S_I 表示编号在 I 中的 S 的子集合的并集,$S_I=\bigcup_{k\in I}S_k$.对 $\{i,j\}$,$1\leqslant i\leqslant n$,$0\leqslant j\leqslant n$,如果存在集合 $I=\{1,2,\ldots,n\}$,使得 $\sum_{k\in I}1=j$ 且 $S_I=S$,则令 $c(i,j)$为使得 $S_I=S$ 的最小子集合 I;否则认为 $c(i,j)$无定义,记为 $c(i,j)=nil$.根据以上定义,目标函数最优值可以表示为 $opt=\min\{j|c(i,j)\neq nil\}$.要找出该(0-1)-规划问题的最优解,只需计算出所有的 $c(i,j)$,具体的计算过程见算法 5.

算法 5. OptimalMSC.

Input:集合 S={1,2,…,n};S_1,S_2,…,$S_n\subset S$.

Output:S 的具有最小势的集合覆盖 C.

```
1.  Initialize: C←{};
2.    for (int i=1; i≤n; i++)
3.        for (int j=0; j≤i; j++)
4.           if (c(i−1,j)=nil && S_{c(i−1,j)}⊂S)
5.              c(i,j)=c(i−1,j)+{i};
6.           else if (c(i,j)≠nil);
7.              c(i,j)=c(i−1,j);
8.        end for;
9.  end for;
10. return C=min{j|c(n,j)≠nil}.
```

对于任意一个子集合 $I\subseteq\{1,2,\ldots,n\}$,计算 S_I 需要时间 $O(n^2)$,因此,算法 5 的时间复杂度为 $O(n^4)$.该算法为一个伪多项式时间算法,如果问题的最大输入值比较大,则算法的效率会比较低.

通常情况下,得到的一个函数依赖集合的最小集合覆盖不是一个最小精确覆盖(minimum exact cover),意味着所得到的最小集合覆盖中通常存在冗余元素.假定函数依赖集合Σ的元素个数为 n,所得到的最小集合覆盖为 C=(C_1,C_2,…,C_k),则出现 C 中任意 2 个子集合有交集或者满足$|C_1|+|C_2|+\ldots+|C_k|>n$,意味着 C 中存在Σ的冗余元素.冗余元素的存在,使得对多个函数依赖进行不一致性批量检测的时候出现重复检测的问题.

为解决所得到的最小集合覆盖存在冗余元素的问题,需要对最小集合覆盖进行预处理.算法 6 用来去除最小集合覆盖中冗余元素.在算法 6 中,多个函数依赖的最小集合覆盖用包含 0,1 元素的矩阵 M 表示,矩阵的行为 X中的所有元素,列为最小集合覆盖中的元素.如果矩阵的任一列值为 1 的元素个数大于 1,则意味着在最小集合覆盖中该列所代表的元素出现了冗余.

算法 6. MSCRedundantEliminate.

Input:集合 X={1,2,…,n};X 的最小集合覆盖 C=(C_1,C_2,…,C_k).

Output:每一列元素值为 1 的元素个数不超过 1 个的矩阵 M.

1. If the matrix M is empty, terminate;
2. Otherwise:

```
3.    for (int i=0; i<|X|, i++)
4.      for (int j=0, counter=0; j<|C|, j++)
5.          if (M_{j,i}==1 && counter<1)
6.              counter++;
7.          else if (M_{j,i}==1 && counter==1)
8.              M_{j,i}=0;   /*修改当前值为 0*/
9.      end for;
10.   end for;
11.  return M.
```

例 4:给定 X={1,2,3,4,5,6,7},C={A,B,C}是 X 的一个最小集合覆盖,其中,A={1,2,4},B={2,3,5},C={4,6,7},用矩阵表示的 X 的最小覆盖如图 2(a)所示.

算法 6 对图 2(a)的矩阵 M 进行扫描,发现第 2 列和第 4 列分别有 2 个值为 1 的元素,因此算法分别修改 M[B][2]以及 M[C][4]的值为 0,修改后的矩阵如图 2(b)所示.根据图 2(b)的矩阵可以得到 X 的一个最小精确覆盖,C'={{1,2,4},{3,5},{6,7}}.

	1	2	3	4	5	6	7
A	1	1	0	1	0	0	0
B	0	1	1	0	1	0	0
C	0	0	0	1	0	1	1

(a) 修改前的矩阵

	1	2	3	4	5	6	7
A	1	1	0	1	0	0	0
B	0	0	1	0	1	0	0
C	0	0	0	0	0	1	1

(b) 修改后的矩阵

Fig.2 Matrix modification of the minimum set cover

图 2 最小集合覆盖矩阵修改

通过算法 6 求解最小精确覆盖的方法,可以得到一个函数依赖集合Σ的最小精确覆盖.基于该最小精确覆盖对Σ中的多个 FD 进行分组,实现任意情况的多个 FD 在分布式环境下不一致性的并行检测.

3 实 验

为了验证文中提出的分布式环境下函数依赖不一致性检测算法的有效性,本文使用真实数据集和合成数据集对算法进行了实验验证,测试了算法在不同节点个数、不同数据集规模以及不同函数依赖个数等情况下的响应时间以及数据迁移量情况.

- 实验设置

本实验中使用了由 10 台服务器通过局域网连接构成的一个集群,每一台机器配备了主频为 1.87G 的 Intel Xeon 2 处理器和 16G 内存,操作系统为 Ubuntu10.4.所有算法均由 Java 实现,算法运行平台为分布式系统架构 Hadoop 平台与基于 BSP(bulk synchronous parallel)并行计算框架的 Hama 平台.

(a) 数据集.本文中使用了 2 种不同类型的数据集:一种是 TranStats data library 提供的 Airline On-Time Statistics 统计数据[8],这是一个真实的数据集;另外一种是人工生成的数据集.第一种数据集简称“OTPF”,包含了 64 个属性,如 airline ID,flight number 等.数据集的规模为 35GB,包含了 15 亿条元组.本文使用该数据集生成一个包含 4 000 万条元组的数据库实例 $otpf_4$,一个包含 8 000 万条元组的数据库实例 $otpf_8$,一个包含 1.2 亿条元组的数据库实例 $otpf_{12}$.另外一种数据集是一个人工生成的前文提到的 *STUDENT* 表的数据集,简称“STUD”,包含了 2 亿条元组.利用该数据集生成一个包含 4 000 万条元组的数据库实例 $stud_4$,一个包含 8 000 万条元组的数据库实例 $stud_8$,一个包含 1.2 亿条元组的数据库实例 $stud_{12}$.

(b) 函数依赖.对于数据集 OTPF,找出一组反映真实约束关系的函数依赖,函数依赖个数为 8 个,每个包含了 2~5 个属性.对于 STUD,定义了 10 个函数依赖,为了验证实验效果,数据集中人为增加了噪声.

- 实验结果

本文设计了 5 组实验,分别对单个函数依赖的集中式算法 CetDetect、基于 Hadoop 的 Naïve 算法 $\text{SingleFdDet}_{\text{Hadoop}}$ 以及本文提出的 $\text{SingleFdDet}_{\text{RT}}$、基于多个函数依赖的顺序检测算法 SuccDetect、基于 Hadoop 的多函数依赖不一致性检测 Naïve 算法 $\text{MultiFdDet}_{\text{Hadoop}}$ 以及本文提出的 $\text{MultiFdDet}_{\text{RT}}$ 进行测试,并对近似最优算法与真正最优方案的执行时间进行了比较.在实验中,分别改变节点的个数(|*S*|)、数据的规模(|*D*|),每一组实验均在相同条件下运行 3 次,取实验结果的平均值为最终结果值.

首先评估基于单个函数依赖的检测算法,对每一个数据集选取一个函数依赖进行检测.

- 实验 1:改变数据的分片个数

为了评价算法在不同分块数(节点数)情况下的扩展性,本文在数据规模固定的情况下增加节点数|*S*|从 2 到 8,分别基于数据集 $otpf_8$ 和 $stud_8$,对算法的执行时间进行测试.图 3 和图 4 反映了算法 CetDetect,$\text{SingleFdDet}_{\text{Hadoop}}$, $\text{SingleFdDet}_{\text{RT}}$ 在不同节点下响应时间情况.跟预想的情况类似,算法 $\text{SingleFdDet}_{\text{Hadoop}}$ 与 $\text{SingleFdDet}_{\text{RT}}$ 的响应时间随着节点的增加呈现减少的趋势.由于算法 CetDetect 随着节点的增加,数据迁移量相应增加,相应数据传输时间增加,而集中检测的检测时间在每种情况下基本相同,因此算法总的响应时间有增加的趋势.由于算法 $\text{SingleFdDet}_{\text{RT}}$ 可以进行并行检测,因此响应时间明显比算法 CetDetect 要小.算法 $\text{SingleFdDet}_{\text{Hadoop}}$ 与 $\text{SingleFdDet}_{\text{RT}}$ 相比,由于本身 Hadoop 任务启动存在一定的耗时,且数据未经过去重等预处理,数据迁移量及数据规模相对较大,因此响应时间比 $\text{SingleFdDet}_{\text{RT}}$ 要长.

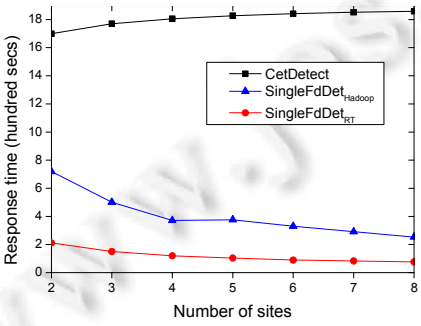


Fig.3 Scalability with |*S*| ($otpf_8$)

图 3 |*S*|的扩展性($otpf_8$)

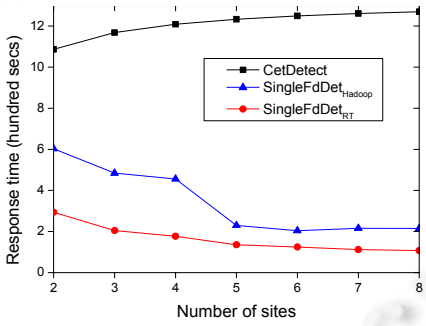


Fig.4 Scalability with |*S*| ($stud_8$)

图 4 |*S*|的扩展性($stud_8$)

- 实验 2:改变数据的规模

为评价算法在不同数据规模情况下的扩展性,本文在节点个数固定(4 个节点)的情况下,增加数据规模|*D*|从 2 000 万条元组到 1.2 亿条元组,分别基于数据集 $otpf_{12}$ 和 $stud_{12}$,对算法的响应时间进行测试.图 5 和图 6 反映了算法 CetDetect,$\text{SingleFdDet}_{\text{Hadoop}}$,$\text{SingleFdDet}_{\text{RT}}$ 在不同数据规模下响应时间情况.

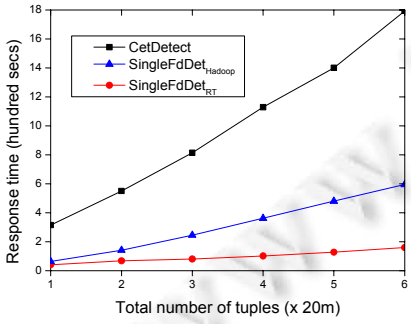


Fig.5 Scalability with |*D*| ($otpf_{12}$)

图 5 |*D*|的扩展性($otpf_{12}$)

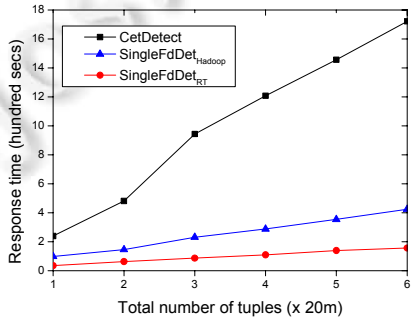


Fig.6 Scalability with |*D*| ($stud_{12}$)

图 6 |*D*|的扩展性($stud_{12}$)

从图中可以看出,3 种算法的响应时间随着数据规模的增加而增大,数据规模的增大直接影响到检测的时间消耗及数据传输时间,因此数据规模与算法响应时间成正相关.从图 5 和图 6 不难看出,算法 SingleFdDet$_{Hadoop}$, SingleFdDet$_{RT}$ 比算法 CetDetect 的响应时间少很多,说明分布式并行算法 SingleFdDet$_{Hadoop}$,SingleFdDet$_{RT}$ 的检测效率明显高于集中式算法 CetDetect.而且从图 5 和图 6 可以看出,随着数据规模的不断增大,算法 SingleFdDet$_{Hadoop}$,SingleFdDet$_{RT}$ 与 CetDetect 相比,在响应时间方面优势明显;而算法 SingleFdDet$_{RT}$ 与 SingleFdDet$_{Hadoop}$ 相比,随着数据规模的增加,减少的响应时间呈增加的趋势.

其次评估基于多个函数依赖的检测算法,对数据集 OTPF 和 STUD,分别选取 LHS 包含公共属性的 3 个 FD 进行检测.

- 实验 3:改变数据的分片数

为评价多个 FD 的检测算法在不同分块数情况下的扩展性,本文在数据规模固定的情况下增加节点数|S|从 2 到 8,分别基于数据集 $otpf_8$ 和 $stud_8$,对算法的执行时间进行测试,实验结果如图 7 和图 8 所示.从实验结果可以看出:随着节点数的不断增加,算法的响应时间呈减少的趋势.与顺序检测的算法 SuccDetect 相比,批量检测算法 MultiFdDet$_{Hadoop}$,MultiFdDet$_{RT}$ 的响应时间还不到其一半,说明批量检测算法在检测多个 FD 不一致性时优势明显.随着节点数的增加,MultiFdDet$_{RT}$ 响应时间均明显少于 MultiFdDet$_{Hadoop}$,说明本文提出的检测算法与基于 Hadoop 的 Naïve 方法相比在响应时间方面优势明显.

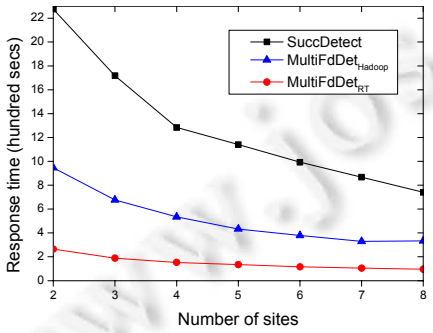


Fig.7 Scalability with |S|, multiple FDs ($otpf_8$)

图 7 多 FDs 时,|S|的扩展性($otpf_8$)

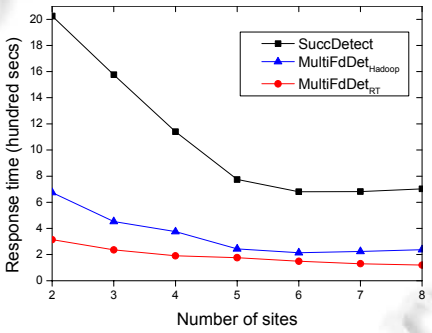


Fig.8 Scalability with |S|, multiple FDs ($stud_8$)

图 8 多 FDs 时,|S|的扩展性($stud_8$)

- 实验 4:改变数据的规模

为评价多个 FD 的检测算法在不同数据规模下的扩展性,本文在节点个数固定(4 个节点)的情况下增加数据规模|D|从 2 000 万条元组到 1.2 亿条元组,对算法的响应时间进行测试.图 9 和图 10 反映了算法 SuccDetect, MultiFdDet$_{Hadoop}$,MultiFdDet$_{RT}$ 在不同数据规模下响应时间情况.

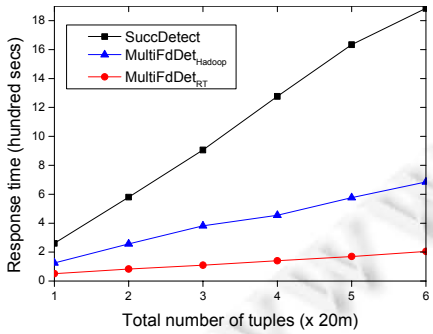


Fig.9 Scalability with |D|, multiple FDs ($otpf_{12}$)

图 9 多 FDs 时,|D|的扩展性($otpf_{12}$)

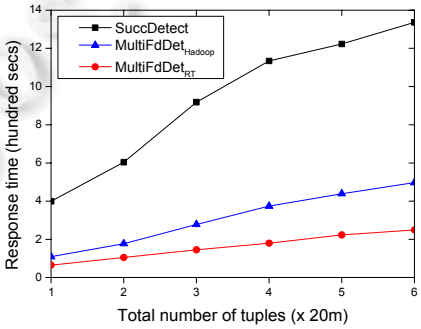


Fig.10 Scalability with |D|, multiple FDs ($stud_{12}$)

图 10 多 FDs 时,|D|的扩展性($stud_{12}$)

不难看出,随着数据规模的不断增加,3 种算法的响应时间也不断增加.与分布式并行算法 $\text{MultiFdDet}_{\text{Hadoop}}$, $\text{MultiFdDet}_{\text{RT}}$ 相比,算法 SuccDetect 的响应时间更长,说明 $\text{MultiFdDet}_{\text{Hadoop}}$,$\text{MultiFdDet}_{\text{RT}}$ 的检测效率更高.而就分布式并行算法来看,本文提出的算法 $\text{MultiFdDet}_{\text{RT}}$ 与 Naïve 算法 $\text{MultiFdDet}_{\text{Hadoop}}$ 相比,在响应时间方面优势明显.随着数据规模的不断增加,算法 $\text{MultiFdDet}_{\text{RT}}$ 的效率优势呈现出不断扩大的趋势.

- 实验 5:近似算法与最优算法比较

为了评价本文提出的近似最优的分布式环境不一致性检测算法与真正最优方案的响应时间差距情况,增加节点数|*S*|,基于数据集 $otpf_8$ 和 $stud_8$,分别对单个函数依赖以及多个函数依赖情况下近似最优算法与真正最优方案的执行时间进行测试.图 11 和图 12 是单个函数依赖情况下近似最优检测算法 $\text{SingleFdDet}_{\text{RT}}$ 和真正最优检测算法 $\text{SingleFdDet}_{\text{OPT}}$ 在不同节点下响应时间情况.不难看出,在节点个数为 2 个时,近似最优算法与真正最优方案的执行时间基本相同;随着节点各种的增加,与近似最优算法相比,真正最优方案的执行时间相对较少,但是总体来说相差不大.图 13 和图 14 是多个函数依赖情况下近似最优检测算法 $\text{MultiFdDet}_{\text{RT}}$ 和真正最优检测算法 $\text{MultiFdDet}_{\text{OPT}}$ 在不同节点下响应时间情况.可以看出,与单个函数依赖情况类似,在节点个数较少的情况下,近似最优算法与真正最优方案的耗时基本相同;随着节点数的增加,真正最优方案与近似最优算法在执行时间方面相比优势有增加的趋势,但是并不十分明显.

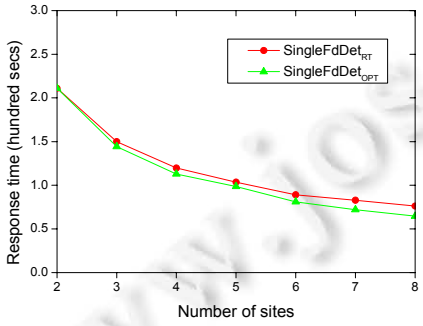


Fig.11 Scalability of single FD with |*S*| ($otpf_8$)

图 11 单 FD |*S*|扩展性($otpf_8$)

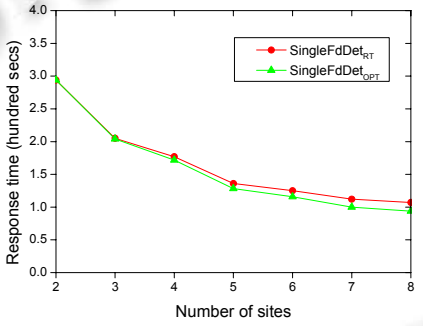


Fig.12 Scalability of single FD with |*S*| ($stud_8$)

图 12 单 FD |*S*|扩展性($stud_8$)

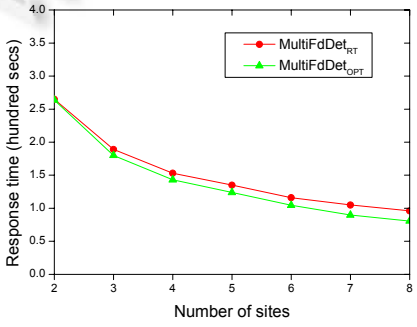


Fig.13 Scalability of multiple FD with |*S*| ($otpf_8$)

图 13 多 FD |*S*|扩展性($otpf_8$)

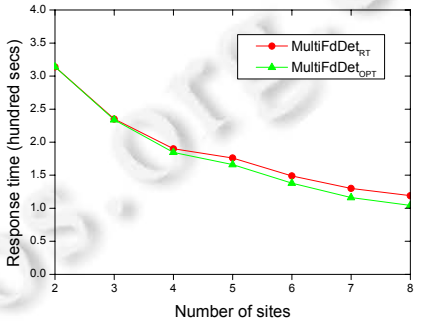


Fig.14 Scalability of multiple FD with |*S*| ($stud_8$)

图 14 多 FD |*S*|扩展性($stud_8$)

4 相关工作

函数依赖的概念最早由 Armstrong 提出[1].Fan 提出了一种基于 SQL 技术检测集中式环境中条件函数依赖冲突的方法[2].然而在分布式环境下对函数依赖冲突进行检测,文献[2]中的方法并不适用.Fan 等人对分布式环境下条件函数依赖冲突检测进行了研究,利用条件函数依赖的结构减少数据迁移或响应时间[4].文献[9]研究了

分布式环境下条件函数依赖不一致性的增量检测.与条件函数依赖不同,函数依赖没有条件函数依赖特有的模式元组,因此,文献[2,9]中提到的算法不适合分布式环境下对函数依赖进行检测.

文献[5,10]研究了分布式数据库中完整性约束的检测问题.文献[10]中给出了检测的本地条件,满足这些条件,不用进行数据的迁移,减少了通信代价.文献[5]研究了在不访问全部基本关系的情况下对基本关系更新时全局不一致性检测问题.在某些条件下,条件函数依赖可以在不进行数据迁移的情况下进行不一致性检测[4].与文献[5,10]不同,在分布式环境下检测函数依赖的不一致性的时候,通常需要数据迁移.

文献[11,12]对分布式数据源的异常检测问题进行了研究.文献[11]提出了一种分布式环境中检测流量异常的框架,避免了全局通信和集中决策.文献[12]提出了一种针对包含混合属性数据集的快速分布式异常检测策略.这些方法主要针对的是异常检测,而函数依赖冲突检测主要是检测函数依赖约束的违反情况.

文献[13]对数据融合中数据冲突的解析进行了研究,主要针对的是集中式环境下的冲突问题.文献[14]中对约束的冲突情况进行了研究,主要研究了分布式系统中使用本地约束检测全局约束违反的情况.

文献[15−17]对函数依赖冲突的修复进行了相关研究.为了对存在违反函数依赖情况的不一致数据进行修复,通常采用对属性值进行修改的方法.与本文工作不同,这些文献主要关注的是函数依赖冲突的修复问题.文献[18,19]研究了函数依赖发现的问题,主要是研究给定一个模式,如何发现隐含的约束关系.文献[20]研究了分布式环境函数依赖发现问题.

5 结论和下一步工作

本文研究了分布式背景下函数依赖冲突检测的问题,给出了分布式环境下单个函数依赖和多个函数依赖不一致性并行检测的算法,并利用函数依赖的结构特征对多个函数依赖实现批量并行检测.为实现对多个函数依赖的批量检测,考虑对多个函数依赖进行分组,组内的可以进行批量不一致性检测.将多个函数依赖的分组问题划归为最小集合覆盖问题,给出了得到问题近似解的贪心算法.为了消除最小集合覆盖中存在的冗余元素导致的重复检测问题,给出了消除最小集合覆盖中冗余元素得到一个最小精确覆盖的算法.基于真实数据集和人工数据集对本文提出的检测算法进行了验证,实验结果表明,算法在数据规模和节点个数方面扩展性良好,而且在减少响应时间方面优势明显.限于篇幅,本文仅对分布式环境下水平切分的数据不一致性检测进行了研究,下一步考虑对分布式环境下垂直切分的数据不一致性检测问题进行研究.

References:

[1] Armstrong WW. Dependency structures of data base relationships. Processings of IFIP Congress 74, 1974,74:580−583.

[2] Fan W, Geerts F, Jia X, Kementsietsidis A. Conditional functional dependencies for capturing data inconsistencies. ACM Trans. on Database Systems, 2008,33(2):1−48. [doi: 10.1145/1366102.1366103]

[3] Beskales G, Ilyas I, Golab L. Sampling the repairs of functional dependency violations under hard constraints. Proc. of the VLDB Endowment, 2010,3(1-2):197−207. [doi: 10.14778/1920841.1920870]

[4] Fan W, Geerts F, Ma S, Muller H. Detecting inconsistencies in distributed data. In: Proc. of the IEEE ICDE. Long Beach, 2010. [doi: 10.1109/ICDE.2010.5447855]

[5] Huyn N. Maintaining global integrity constraints in distributed databases. Constraints, 1997,2(3-4):377−399. [doi: 10.1023/A:1009703814570]

[6] Garey M, Johnson D. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, 1979. 32−38.

[7] Kleinberg J, Tardos É. Algorithm Design. New York: Pearson Education, 2006. 600−622.

[8] http://apps.bts.gov/xml/ontimesummarystatistics/src/index.xml

[9] Fan W, Li J, Tang N, Yu W. Incremental detection of inconsistencies in distributed data. In: Proc. of the IEEE ICDE. Washington, 2012. 318−329. [doi: 10.1109/ICDE.2012.82]

[10] Gupta A, Widom J. Local verification of global integrity constraints in distributed databases. In: Proc. of the 1993 ACM SIGMOD Int'l Conf. on Management of Data. Washington, 1993. [doi: 10.1145/170035.170048]

[11] Chhabra P, Scott C, Kolaczyk ED, Crovella M. Distributed spatial anomaly detection. In: Proc. of the INFOCOM 2008. 2008. 1705−1713.

[12] Koufakou A, Georgiopoulos M. A fast outlier detection strategy for distributed high-dimensional data sets with mixed attributes. Data Mining and Knowledge Discovery, 2010,20(2):259−289. [doi: 10.1007/s10618-009-0148-z]

[13] Dong X, Naumann F. Data fusion—Resolving data conflicts for integration. Proc. of the VLDB Endowment, 2009,2(2):1654−1655. [doi: 10.14778/1687553.1687620]

[14] Agrawal S, Deb S, Naidu KVM, Rastogi R. Efficient detection of distributed constraint violations. In: Proc. of the Int'l Conf. on Management of Data (COMAD 2006). Delhi, 2006. [doi: 10.1109/ICDE.2007.369002]

[15] Kolahi S, Lakshmanan L. On approximating optimum repairs for functional dependency violations. In: Proc. of the 4th Int'l Conf. on Digital Telecommunications. Colmar, 2009. 53−62. [doi: 10.1145/1514894.1514901]

[16] Bohannon P, Fan W, Flaster M. A cost-based model and effective heuristic for repairing constraints by value modification. In: Proc. of the 2005 ACM SIGMOD Int'l Conf. on Management of Data. Baltimore, 2005. [doi: 10.1145/1066157.1066175]

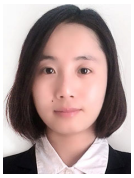
[17] Lopatenko A, Bravo L. Efficient approximation algorithms for repairing inconsistent databases. In: Proc. of the IEEE ICDE. Turkey, 2007. [doi: 10.1109/ICDE.2007.367867]

[18] Huhtala Y, Karkkainen J, Porkka P, Toivonen H. Tane: An efficient algorithm for discovering functional and approximate dependencies. Computer Journal, 1999,42(2):100−111.

[19] Novelli N, Cicchetti R. Fun: An efficient algorithm for mining functional and embedded dependencies. In: Proc. of the 8th Int'l Conf. on Digital Telecommunications. 2001. 189−203. [doi: 10.1007/3-540-44503-X_13]

[20] Li W, Li Z, Chen Q, Jiang T, Liu H. Discovering functional dependencies in vertically distributed big data. In: Proc. of the 16th Int'l Conf. on Web Information System Engineering. 2015. [doi: 10.1007/978-3-319-26187-4_15]

李卫榜(1979－),男,河南周口人,博士生,主要研究领域为数据质量,大数据管理.



杨婧颖(1990－),女,硕士,主要研究领域为数据质量,云数据管理.

李战怀(1961－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为数据库理论与技术.

姜涛(1983－),男,博士生,CCF 学生会员,主要研究领域为生物信息挖掘,数据管理.

陈群(1976－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为云计算,RFID 数据管理, XML 数据管理.