

毕业设计说明书

|  |  |  |  |
| --- | --- | --- | --- |
| **作 者:** | 周子聪 | **学 号：** | 914106840745 |
| **学 院:** | 计算机科学与技术 | | |
| **专业(方向):** | 软件工程 | | |
| **题 目:** | 基于Spark平台的分布式 | | |
|  | 函数依赖发现算法 | | |

**指导者： 蔡志成 导师**

(姓 名) (专业技术职务)

(姓 名) (专业技术职务)

**评阅者：**

(姓 名) (专业技术职务)

二零一八 年 五 月

声 明

我声明，本毕业设计说明书及其研究工作和所取得的成果是本人在导师的指导下独立完成的。研究过程中利用的所有资料均已在参考文献中列出，其他人员或机构对本毕业设计工作做出的贡献也已在致谢部分说明。

本毕业设计说明书不涉及任何秘密，南京理工大学有权保存其电子和纸质文档，可以借阅或网上公布其部分或全部内容，可以向有关部门或机构送交并授权保存、借阅或网上公布其部分或全部内容。

学生签名：

年 月 日

指导教师签名：

年 月 日

毕业设计说明书中文摘要

|  |
| --- |
| 函数依赖发现，是一种十分重要的数据分析技术，它普遍适用于采用关系型数据库的项目中。现如今，在大型企业内部，由于用户数据的急速增长，使得关系型数据库普遍采用分布式存储。倘若继续采用传统的函数依赖发现算法，就会导致单节点的数据不完整，使得计算结果不准确。本论文在当前分布式大数据的环境背景下，提出并实现了一种高效而又准确的分布式非平凡最小函数依赖发现算法DBFD。DBFD算法遵循了Spark平台提倡的中心思想：移动数据不如移动计算，充分发挥内存计算的优势。为达到以上目的，对经典的TANE算法的搜索策略和剪枝策略进行了多处改造适配，并对改造后的策略进行了充分严谨的论证。  关键词 大数据 数据挖掘 分布式计算 函数依赖发现 Spark |

毕业设计说明书外文摘要

|  |
| --- |
| **Title**  Distributed functional dependency discovery algorithm  based on Spark platform  **Abstract**  Functional dependency discovery is a very important data analysis technique that is universally applicable to any project that uses relational databases. Nowadays, in large enterprises, due to the rapid growth of user data, distributed storage is generally adopted in relational databases. If we continue to use the traditional functional dependency discovery algorithm, the data of single node will be incomplete, making the calculation result inaccurate. In the context of current distributed big data, this paper proposes and implements a efficient and accurate distributed functional dependency discovery algorithm DBFD. The DBFD algorithm follows the central idea advocated by the Spark platform: move data is not as good as move computing, giving full play to the advantages of memory computing. In order to achieve the above objectives, the search strategy and pruning strategy of the classic TANE algorithm are modified and adapted, and the modified strategy is fully and rigorously demonstrated.  Keywords Big data Data mining Distributed computing Functional dependency discovery FD discovery Spark |

目录

[1 绪论 1](#_Toc515217336)

[1.1 背景 1](#_Toc515217337)

[1.2 国内外研究现状 1](#_Toc515217338)

[1.2.1 经典函数依赖发现算法 1](#_Toc515217339)

[1.2.2 分布式函数依赖发现算法现状 2](#_Toc515217340)

[1.3 本课题主要研究内容 3](#_Toc515217341)

[1.4 论文章节安排 3](#_Toc515217342)

[2 预备知识 5](#_Toc515217343)

[2.1 函数依赖相关术语[12] 5](#_Toc515217344)

[2.1.1 数据依赖 5](#_Toc515217345)

[2.1.2 函数依赖(Functional Dependency) 5](#_Toc515217346)

[2.1.3 平凡函数依赖 6](#_Toc515217347)

[2.1.4 非平凡函数依赖 6](#_Toc515217348)

[2.1.5 最小函数依赖(完全函数依赖) 6](#_Toc515217349)

[2.1.6 寻找(挖掘)最小非平凡函数依赖过程中的剪枝策略 6](#_Toc515217350)

[2.1.7 等价类(Equivalent Class) 7](#_Toc515217351)

[2.1.8 划分(Partition) 7](#_Toc515217352)

[2.1.9 精简划分(Stripped Partition) 7](#_Toc515217353)

[2.2 分布式计算相关术语[13] 7](#_Toc515217354)

[2.2.1 数据迁移 7](#_Toc515217355)

[2.2.2 负载均衡 7](#_Toc515217356)

[2.3 Spark相关概念 7](#_Toc515217357)

[2.3.1 HDFS[14] 8](#_Toc515217358)

[2.3.2 内存计算 8](#_Toc515217359)

[2.3.3 Shuffle[15] 8](#_Toc515217360)

[2.3.4 主节点与工作节点 8](#_Toc515217361)

[3 问题描述 9](#_Toc515217362)

[3.1 分布式大数据环境下的典型问题 9](#_Toc515217363)

[3.2 目前算法仍待解决的问题 9](#_Toc515217364)

[4 DBFD算法思想 12](#_Toc515217365)

[4.1 耗时操作分析 12](#_Toc515217366)

[4.1.1 一次数据重分布的时间计算 12](#_Toc515217367)

[4.1.2 一次划分求阶的时间估算 13](#_Toc515217368)

[4.2 搜索策略 13](#_Toc515217369)

[4.2.1 搜索前的准备工作 14](#_Toc515217370)

[4.2.2 整体搜索策略 14](#_Toc515217371)

[4.2.3 局部搜索策略 15](#_Toc515217372)

[4.2.4 属性集内部搜索策略 15](#_Toc515217373)

[4.2.5 复杂度分析 15](#_Toc515217374)

[4.3 剪枝策略 16](#_Toc515217375)

[4.3.1 剪枝概念介绍 16](#_Toc515217376)

[4.3.2 右方集剪枝策略 18](#_Toc515217377)

[4.3.3 增强右方集剪枝策略 19](#_Toc515217378)

[4.3.4 键剪枝 21](#_Toc515217379)

[4.4 函数依赖判定 22](#_Toc515217380)

[4.4.1 FD的判定方法一 22](#_Toc515217381)

[4.4.2 FD的判定方法二 22](#_Toc515217382)

[4.5 数据迁移策略 24](#_Toc515217383)

[4.6 算法整体流程图 27](#_Toc515217384)

[5 算法实现 28](#_Toc515217385)

[5.1 数据结构 28](#_Toc515217386)

[5.1.1 存储关系数据集r 28](#_Toc515217387)

[5.1.2 表示属性及其映射 30](#_Toc515217388)

[5.1.3 记录已发现的FDs 31](#_Toc515217389)

[5.1.4 存储当前层级的属性集 31](#_Toc515217390)

[5.1.5 存储属性集对应的增强右方集 31](#_Toc515217391)

[5.1.6 记录已计算的划分的阶 31](#_Toc515217392)

[5.1.7 最大内存占用 31](#_Toc515217393)

[5.2 伪代码 32](#_Toc515217394)

[5.2.1 搜索前的准备 32](#_Toc515217395)

[5.2.2 数据重分布 33](#_Toc515217396)

[5.2.3 并行计算划分的阶 33](#_Toc515217397)

[5.2.4 FD判定 34](#_Toc515217398)

[5.2.5 增强右方集剪枝 34](#_Toc515217399)

[5.2.6 键剪枝 34](#_Toc515217400)

[5.2.7 计算遗留的候选FD 34](#_Toc515217401)

[5.2.8 生成下一层级 35](#_Toc515217402)

[5.2.9 筛选最小FD 35](#_Toc515217403)

[6 实验结果分析 37](#_Toc515217404)

[6.1 实验准备 37](#_Toc515217405)

[6.1.1 实验环境 37](#_Toc515217406)

[6.1.2 实验数据 37](#_Toc515217407)

[6.1.3 实验设计 37](#_Toc515217408)

[6.2 实验结果 38](#_Toc515217409)

[6.3 理论分析 39](#_Toc515217410)

[6.3.1 分析节点扩充对算法的影响 39](#_Toc515217411)

[6.3.2 分析数据扩展对算法的影响 40](#_Toc515217412)

[6.3.3 分析数据顺序变更对算法的影响 40](#_Toc515217413)

[6.3.4 DBFD的性能优势 40](#_Toc515217414)

# 1 绪论

本章主要介绍分布式函数依赖的背景、研究现状以及本文的主要内容。

## 1.1 背景

在大数据时代，从数量巨大但价值密度较低的关系型数据库中挖掘有价值的信息已成为了技术热点话题。然而在大型分布式关系型数据库中进行函数依赖发现并没有想象中的那么热门，一是因为它不像关联规则发现[1]那样，具有很高的商业价值；二是因为它要做到分布式并行计算代价十分昂贵。

尽管函数依赖发现在数据挖掘、数据库设计以及数据质量评估等众多领域有着广泛的应用，但这些领域一般对数据规模的要求并不高，对发现结果的精确性要求并不高，加上分布式大数据函数依赖发现算法的研究资料匮乏，自行探索该算法的成本较高，往往会选择采取经典的函数依赖发现算法。相反的，关联规则发现在商品推荐领域产生了巨大经济效益，纸尿布与啤酒的销售搭配成了佳话。由此可见，函数依赖发现技术的缓慢发展一方面是由于需求不够旺盛，另一方面，函数依赖发现不同与关联规则发现，要进行分布式并行计算，它必须进行各节点之间的数据迁移才能够保证结果的正确性。

本文正是在分布式函数依赖发现算法十分匮乏的背景下，本着技术驱动社会发展的原则，选择迎难而上，填补分布式大数据函数依赖发现算法的缺口。

## 1.2 国内外研究现状

本小节将分别介绍经典函数依赖发现算法、分布式函数依赖发现算法的研究现状以及优缺点。

### 1.2.1 经典函数依赖发现算法

函数依赖在关系型数据库设计基础中是一种重要的概念。函数依赖还在数据库管理中扮演重要角色，以保证数据库的质量，并在反向工程中捕获关系数据源中的数据语义。因而，在关系型数据库流行时期，诞生了许多函数依赖发现算法，主要分为以下三大类[2]：

**（1）生成并测试候选集(TANE**[3]**、FUN**[4]**)**

总体策略同Apriori算法，具体处理某一结点的做法是基于属性集的具体取值而将元组集划分等价类。该算法具体步骤如下:首先,扫描函数依赖的左边;然后利用剪枝规则减少候选集,从而减少搜索空间。TANE和FUN只是在修剪规则上略有不同。

**（2）最小覆盖法(FDEP**[5]**、FastFDs**[6]**、Dep Miner**[7]**)**

该类方法中,最典型的三个算法为FDEP、FastFDs和Dep Miner。方法的基本步骤是:从最原始关系中提炼出剥离分区数据库,计算划分并产生最大的划分集,根据这些最大的划分,产生最小的FD覆盖。

**（3）规范概念分析(FD Mine**[8]**)**

考虑关系数据库理论与形式概念分析的关系,使用预定义的形式概念分析闭包操作提取数据库中的函数依赖。FD Mine用的是generate-and-test方法。

以上FD算法都是为集中式关系数据集设计的，当这些算法被应用于分布式关系数据集时，不同节点之间数据通信的时间消耗将会使得算法不再高效。

若将以上提出的集中式函数依赖发现算法直接移植到分布式环境下，可能会产生如下问题：a.负载不均衡，b.准确性低，c.数据迁移量大

三种简单的解决办法及其优缺点：

1）将数据全部迁移至一个节点,全部在这个节点计算,闲置其他计算资源,十分费时。

当节点增加时，不仅浪费的资源更多，还使得网络传输时间增加。

2）不考虑分布式问题（正确性问题），虽然十分高效但存在很大的计算错误风险。

3）不停地进行数据重新分布，大量数据迁移使得计算资源被闲置，耗时严重。

因此需要对集中式算法进行深入的改造，一次性解决以上问题以适用于分布式大数据环境。

### 1.2.2 分布式函数依赖发现算法现状

在关系数据库中,函数依赖发现是一种十分重要的数据库分析技术,在知识发现、数据库语义分析、数据质量评估以及数据库设计等领域有着广泛的应用。现有的函数依赖发现算法主要针对集中式数据,通常仅适用于数据规模比较小的情况。在大数据背景下,分布式环境函数依赖发现更富有挑战性。

分布式函数依赖发现算法按照切分方式分为两类：

**（1）水平切分：**将关系数据集按照元组进行切分。其基本思想是首先在各个节点利用本地数据并行进行函数依赖发现,必要时进行数据迁移，每次函数依赖计算完成后将数据回传主节点，基于已计算的结果对函数依赖候选集进行剪枝。

**（2）垂直切分：**将关系数据集按照属性进行切分。基本思想是遵循广度优先遍历策略并对属性集进行了高效修剪，并且可以随着集群节点的数量和数据集的大小进行扩展。

由于本文研究的是对水平切分的数据集进行函数依赖发现，因而不对垂直切分方式进行深入探讨，若感兴趣可阅读DFDD算法[9]。目前针对水平切分方式设计的算法也十分稀少，参考文献十分有限，仅找到一篇相关论文。接下来将对该论文中提出的FDPar\_Discover算法[10]进行大体介绍与分析。

FDPar\_Discover算法其基本思想是：首先在各个节点利用本地数据并行进行函数依赖发现,基于以上发现的结果对函数依赖候选集进行剪枝,然后进一步利用函数依赖的LHS的特征,对函数依赖候选集进行分组,针对每一组候选函数依赖并行执行分布式环境发现算法,最终得到所有函数依赖.对不同分组情况下所能检测的候选函数依赖数量进行分析，在算法的执行过程中,综合考虑数据迁移量和负载均衡的问题。

主要步骤如下:

1）通过利用候选函数依赖的结构特征对候选函数依赖进行分组，组内的候选函数依赖可以通过一次数据重分配进行平行发现;

2）利用各个节点并行发现函数依赖,有效提高函数依赖的发现效率,大大减少函数依赖发现所耗费的时间。

FDPar\_Discover算法虽然避免了上述分布式环境下可能出现的三种问题，但仍旧存在诸多问题。具体详情将在后续章节进行讨论并予以解决。

## 1.3 本课题主要研究内容

提出并实现一种基于Apache Spark[11]平台的分布式大数据最小非平凡函数依赖发现算法。

总体目标：找出给定关系r的所有最小非平凡函数依赖：X→A.其中，|X|大于等于1 ,|A|=1。

其依据如下：所谓X→A是“最小”的，是指若X→A成立，则X的任何真子集Y，都有Y→A不成立。所谓X→A是“非平凡”的，是指若X→A成立，则A不属于X。只挖掘|A|=1的原因是根据阿姆斯特朗公理中的合并律，任何大于1的右方属性集，都可以由多个X→A形式的FD合并而成。

本文将通过设计巧妙的搜索策略、剪枝策略以及数据迁移策略，来解决分布式大数据环境下所面临的诸多问题。

本课题将在较适合大数据计算的TANE算法中的搜索策略与剪枝策略之上进行改造升级以适用于分布式大数据环境下的FD发现。

## 1.4 论文章节安排

接下来，将在第二章中预先介绍本文涉及的术语；第三章中将会介绍本课题所面临的主要问题；第四章重点介绍了DBFD算法思想，并对算法中采用的各种策略进行了严密的论证；第五章则对DBFD算法的数据结构进行了详尽的解释说明，以及给出了对算法的整体代码实现。最后一章对本算法的实验结果进行了分析。

# 2 预备知识

本章主要介绍课题相关术语以及须预先掌握的知识概念。其中包括函数依赖相关术语、分布式计算相关术语以及Spark相关概念。

## 2.1 函数依赖相关术语[12]

本小结介绍了课题中频繁提及的函数依赖相关术语。

### 2.1.1 数据依赖

在计算机科学中，数据依赖是指一种状态，当程序结构导致数据引用之前处理过的数据时的状态。其中最重要的是函数依赖和多值依赖。

### 2.1.2 函数依赖(Functional Dependency)

表2.1 关系r

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | A | B | C | D |
| 1 | a1 | b1 | c1 | d1 |
| 2 | a2 | b1 | c2 | d1 |
| 3 | a2 | b2 | c2 | d1 |
| 4 | a3 | b2 | c3 | d1 |

设X,Y是关系R的两个属性集合，当任何时刻R中的任意两个元组中的X属性值相同时，则它们的Y属性值也相同，则称X函数决定Y，或Y函数依赖于X，记为X→Y。X称为LHS（left hand side），Y称为RHS（right hand side）。

例如表2.1中：

A→C,A→D,B→D,C→A,C→D,AB→A,AB→B,AB→C,AB→D,AC→A,AC→C,AC→D,AD→A,AD→C,AD→D, BC→A,BC→B,BC→C, BC→D,BD→B, BD→D, CD→A, CD→C, CD→D等等均为关系集R的函数依赖

注：

a．A→C简写自{A}→{C};

b．AB→C简写自{A,B}→{C};

c．A→C,A→D等价于A→CD

本文重要约定：

* RHS均只有一个属性
* 函数依赖简称为FD
* R代表当前关系集所有属性构成的集合
* 用字母X或Y代表属性集合，A或B或C代表单个属性
* 诸如此类{A,B}→{C}均简写为AB→C
* X\A代表属性集X除去A之后的属性集
* XA代表属性集X加上A之后的属性集

### 2.1.3 平凡函数依赖

当关系中属性集合Y是属性集合X的子集时(Y⊆X)，存在函数依赖X→Y，即一组属性函数决定它的所有子集，这种函数依赖称为平凡函数依赖。

例如表2.1中：

AB→A, AB→B, AC→A, AC→C等等均为关系集R的平凡函数依赖

### 2.1.4 非平凡函数依赖

当关系中属性集合Y不是属性集合X的子集时，存在函数依赖X→Y，则称这种函数依赖为非平凡函数依赖。

### 2.1.5 最小函数依赖(完全函数依赖)

如果X→A（在r上）成立，且不存在X的真子集(如果有)函数依赖A，那么X→A称为最小函数依赖。

如：

若存在A→C和AB→C，则AB→C不是最小的。

注意，最小函数依赖可以是平凡的。例如，若函数依赖XA→A成立，且不存在X的子集函数依赖A，那么XA→A可以称为最小平凡函数依赖。

### 2.1.6 寻找(挖掘)最小非平凡函数依赖过程中的剪枝策略

当发现或排除一个函数依赖则可以对候选函数依赖集进行相应的剪枝。

如：

若存在AB→C，则可以将ABD→C等非最小的候选函数依赖（即含有AB的集合 →C）进行剔除；

若不存在AB→C，则可以将A→C, B→C等较小的候选函数依赖（即AB的子集→C）进行剔除

### 2.1.7 等价类(Equivalent Class)

属性集X的某一等价类是指在给定关系实例中，所有与元组t在X上取值相等的元组的集合。

如：

[2]{A}=[3]{A}说明有在属性集{A}上的等价类{2,3}

### 2.1.8 划分(Partition)

属性集X上的划分的含义是在给定关系实例r中，X的所有等价类的集合。

例如表2.1中：

={{1},{2,3},{4}}

### 2.1.9 精简划分(Stripped Partition)

X的精简划分是指在划分的基础上去除所有阶为1的等价类后的划分。

例如表2.1中：

={{2,3}}

注：集合的阶(Rank)：指集合中所含元素的个数，集合S的阶记为|S|。

## 2.2 分布式计算相关术语[13]

本小结中将对数据迁移和负载均衡两个专有名词进行简要解释。

### 2.2.1 数据迁移

由于完整的数据资源分布在不同计算机中，当某台计算机进行计算需要访问其他计算机之上的数据资源时，此时需要进行数据迁移——将其他计算机上的数据通过网络迁移到本机。

### 2.2.2 负载均衡

多台计算机进行并行计算时，难免出现不同计算机上承载的计算任务负担高低不同，采取一定的措施后，能够在一定程度上保证不产生负载高低悬殊过大的情况发生，我们称该计算集群达到了负载均衡。

## 2.3 Spark相关概念

### 2.3.1 HDFS[14]

全称为Hadoop分布式文件系统，运行于使用网络连接的Hadoop集群之上。将整个数据集上传至HDFS中，使得各节点可以自动读取文件的不同部分，否则需要人工切分数据集并人工拷贝到各节点。

### 2.3.2 内存计算

将原始数据全部读入内存，并将中间计算结果也存储在内存中，相比于反复从磁盘存取，可大大提高计算效率。但对计算机集群内存总大小要求较高。

### 2.3.3 Shuffle[15]

当某种操作无法在多台计算机上进行并行计算，而需要依靠彼此的数据时，需要将各自的数据相互传输，这种需要进行数据迁移才能进行计算的操作称为Shuffle操作，对应的Spark方法叫做Shuffle算子。Shuffle是十分费时的操作，换言之，本论文的主要目的是：减少Shuffle的同时保证计算的正确性。

### 2.3.4 主节点与工作节点

简单来说，主节点的主要工作是：首先，产生计算任务并将计算任务分配给各个工作节点，然后，接收各个工作节点返回的计算结果数据并对这些数据进行综合计算，从而得出本次计算任务的最终结果。所有工作节点接收到主节点分配的任务之后，将在各个节点对此任务进行并行计算，然后各自把计算结果回传给主节点。主节点与工作节点又称主从节点，也称Master与Slaves。

# 3 问题描述

本小结将介绍课题所面临的主要问题，其中包括分布式大数据环境下的典型问题和FDPar\_Discover算法中存在的问题。

## 3.1 分布式大数据环境下的典型问题

在分布式大数据环境下进行函数依赖发现，具有两大难点：

**分布式问题（正确性问题）**

当关系集r（表2.1）拆分为了两个关系集（表3.1.1）和（表3.1.2）时，分别对各自表中进行函数依赖发现，得到的结果并不准确。

如:

函数依赖A→B分别在和上成立，但在r上不成立。

表3.1.1 关系

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | C | D |
| a1 | b1 | c1 | d1 |
| a2 | b1 | c2 | d1 |

表3.1.2 关系

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | C | D |
| a2 | b2 | c2 | d1 |
| a3 | b2 | c3 | d1 |

**大数据问题（高效性问题）**

关系数据集原始数据或中间计算结果可能会超出计算机的内存，因而对数据采取的处理方法与存储方式显得十分关键。

## 3.2 目前算法仍待解决的问题

FDPar\_Discover算法解决了集中式环境下FD发现算法存在的问题（负载不均衡、准确性低和数据迁移量大）。解决办法如下：

对关系（表1-1）和（表1-2）上的数据进行重新分布，即进行数据迁移。若要判断函数依赖A→B, A→C, A→D, AB→C, AB→D, AC→B, AC→D, AD→B, AD→C,ABC→D, ABD→C,ACD→B（即左侧含有A属性的FD）是否成立，则只需要重新分布一次，将属性A相同的元组放在一个关系集中，可借助适当的哈希算法[16]完成。

但仍旧存在较大的问题：

（1）存在不必要的计算。它寻找的仅仅是非平凡FD，也就是说它将会计算非最小的FD，我们知道非最小的FD没有多大价值，将其计算并输出不仅影响结果的观察，而且十分浪费计算资源。一般地，若计算出A→B仍会计算AC→B、AD→B、AE→B，而这些FDs是一定成立的且没有多大价值的，因此属于不必要的计算开销。

（2）存在负载不均衡的缺陷。它的数据重分布策略存在明显的缺陷。特别地，当所有的元组对应某一属性的值均相同时，则会将所有数据均迁移到一个工作节点之上，造成严重的负载不均衡。

表1.1 A属性值均相同的关系集

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ID | A | B | C | D | E |
| 1 | a1 | • | • | • | • |
| 2 | a1 | • | • | • | • |
| ••• | a1 | ••• | ••• | ••• | ••• |
| N | a1 | • | • | • | • |

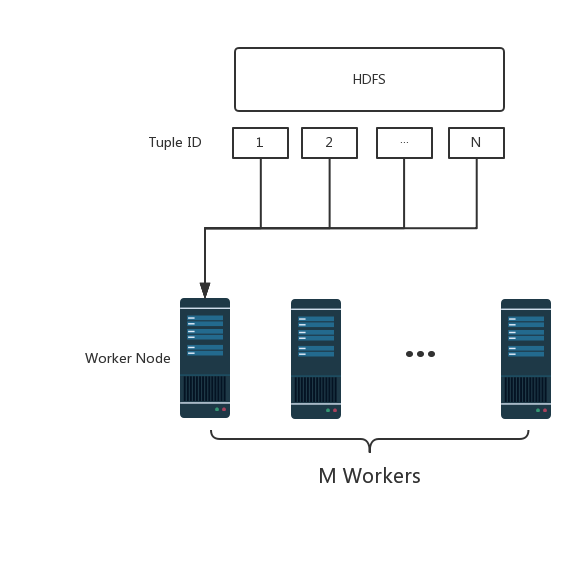


图1.1 FDPar\_Discover中可能出现的负载不均衡现象

更一般地，当存在任意一个属性的所有元组的值的Set集合的阶小于工作节点数，则势必造成负载不均衡。换言之，当存在一个属性A，使得所有元组的A属性的值总共有X种，若工作节点数目Y>X，则会造成比较严重的负载不均衡，即存在(Y-X)个节点“无事可做”。

本章节提及的所有问题将会在后续章节中逐一解决。

# 4 DBFD算法思想

本章将提出一种较为高效的分布式最小FD发现算法——DBFD，并将对其进行较为严密的论证。

## 4.1 耗时操作分析

对算法中存在的耗时操作有充分的认识是设计出高效可行算法的前提。分布式大数据FD发现算法的耗时操作主要为数据迁移（网络传输）和FD的判定计算。接下来提到的关系数据集大小均为对每一个原始属性值取其32位哈希值后的大小。至于对原始属性值取哈希值的原因将会在第五章进行解释说明。

### 4.1.1 一次数据重分布的时间计算

假定Spark集群各节点采用千兆以太网连接，则N台工作节点每次重分布X GB数据耗时约为：

s

计算过程如下：

由于各个工作节点并行进行数据迁移，因而只需要计算单个工作节点的数据迁移时间

单个节点的数据量为：

GB

由于每个节点的数据约有是需要保留给自己的，所以单个节点需要进行迁移的数据量约为：

GB

千兆以太网的传输速率为：

GB/s

网络传输时间约为：

s

一次数据传输时间还包括本地发送延时、排队延时、路由寻址延时和远程接收延时等，但一般这些延时总和在毫秒级以下，因而暂可忽略不计。

假设采用8个节点的集群，对1GB数据进行重新分布，则大约需要花费1秒。

### 4.1.2 一次划分求阶的时间估算

使用配有1.4 GHz Intel Core i5处理器的计算机进行采样计算，测得每计算1千万条数据对应的单个属性的划分时间消耗约为：0.3s

并测得该时耗与划分的属性个数成正比，与数据量成正比

假定集群均采用1.4 GHz Intel Core i5的处理器进行计算，则N台工作节点对具有M个属性的X GB数据的Y个属性进行划分求阶计算，耗时为：

s

计算过程如下：

由于每条数据有M个属性，且已知每个属性为32位的哈希值，所以每条数据的大小为：

B

具有M个属性的X GB数据条数约为：

千万条

一台工作节点对这26.8X/M千万条数据对应的Y个属性进行划分求阶耗时约为：

s

N台工作节点耗时则为：

s

假设采用8个节点的集群，对具有10个属性的1 GB数据的10个属性进行划分求阶计算，则大约需要花费1秒。

## 4.2 搜索策略

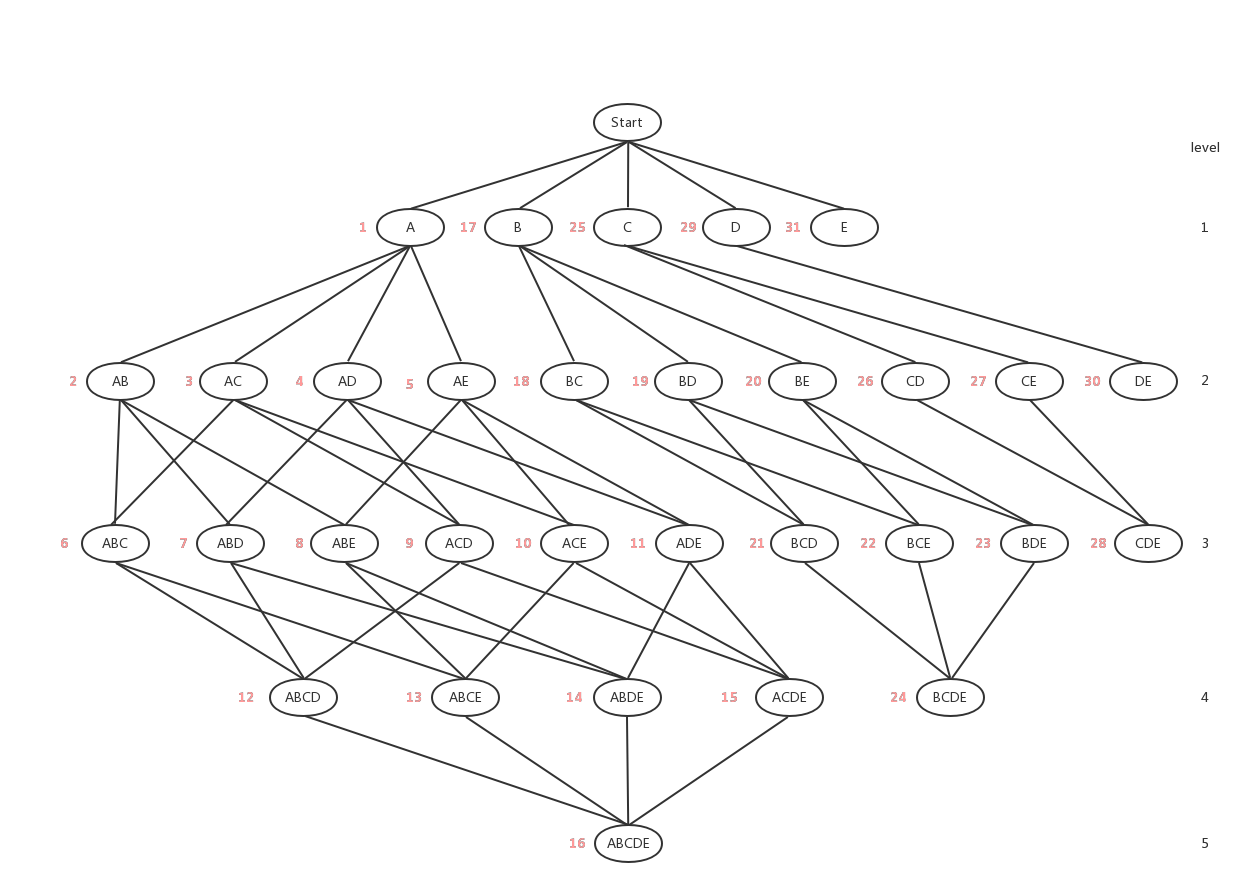


图4.1 DBFD搜索策略

### 4.2.1 搜索前的准备工作

首先，计算出各个属性值的种类数目，按照属性值种类多到少的原则进行排序，也即图4.1的ABCDE这个顺序已经表明这些属性对应的值的种类数目依次递减。

计算各个属性值的种类与数目不是也需要进行数据迁移吗？

是的，但是数据迁移不等同于上面提到的数据重分布，数据重分布指的是所有数据进行迁移。而计算每个属性值的种类只需要进行小规模的数据迁移。过程如下：

1. 各节点分别计算出相应属性值的Set集合（由于Set集合会去除重复数据，所以规模会比原始数据小很多）
2. 依次将两个节点的数据迁移汇合求并集
3. 最终的集合大小即为该属性值的种类数目

因此，求单个属性值的种类的数据迁移耗时可忽略不计。

### 4.2.2 整体搜索策略

整体而言，从属性值种类多到少进行搜索。如图4.1，先搜索A属性节点的所有子树，再搜索B、C、D和E的所有子树。目的是在较大程度上解决图1中存在的这类负载不均衡问题。同时，采取分属性依次搜索的策略，不仅确保了FD发现的准确性，而且使得数据迁移次数达到较低水平。

### 4.2.3 局部搜索策略

局部而言，处理的属性集从小到大。如图4.1，针对A属性节点的所有子树，先搜索属性个数为2的并且包含属性A的子节点，再搜索属性个数为3、4和5的并且包含属性A的子节点，也即对A属性节点的所有子树进行层次遍历（每一层按照字典升序排列）。目的是要能够尽可能保证优先计算的是最小FD（但不能确保一定是最小的，因为可能存在RHS相同但LHS去除A属性的FD），同时能够进行高效地剪枝。

### 4.2.4 属性集内部搜索策略

属性集内部而言RHS从右到左。如图4.1中的ABC属性集（节点6）内部，先计算RHS为C的FD，再计算RHS为B的，（A属性节点的所有孩子节点内部，均不计算RHS为A的FD，因为存在之前提出的正确性问题）也即依次计算AB→C,AC→B。目的仅仅是为了FD发现的顺序直观。需要注意的是，当计算BC属性集（节点18）时，先计算B→C，再计算BC→A，也就是说所有B属性节点的孩子节点均要计算LHS为本节点属性集、RHS为A的FD；C属性节点的孩子节点均要计算LHS为本节点属性集、RHS为A和B的FD；以此类推。

### 4.2.5 复杂度分析

对M个属性构建此类树型结构，则拥有个节点。

证明如下：

第0层节点个数为：

第1层节点个数为：

第2层节点个数为：

第i层节点个数为：

第M层节点个数为：

由二项式定理可计算出总的节点个数为：

因此可见，搜索的时间复杂度与属性个数呈指数关系。这势必导致无法对拥有较多属性的关系集进行FD发现。但幸运的是，FD存在一些特殊的性质，使得我们可以通过一定的剪枝策略来提高可计算属性的个数。

## 4.3 剪枝策略

### 4.3.1 剪枝概念介绍

经过对一些FDs进行计算后，有时能够推断出某些节点的所有子节点均不需要进行计算（由于本课题寻找的是最小非平凡FD，因此非最小FD也无须进行计算），因此可以将所有这些无须计算的节点进行修剪删除，这种操作称为剪枝[17]。

剪枝的重要依据一：

**当函数依赖X→A成立，若，则必有Y→A成立。**

证明如下：

根据阿姆斯特朗公理系统的自反律，若，则有Y→X成立；

再根据阿姆斯特朗公理系统的传递律，若有Y→X成立并且X→A成立，则必有Y→A成立。

证毕。

剪枝的重要依据二：

**当函数依赖X→A成立，，均不是最小的函数依赖。**

证明如下（反证法）：

根据阿姆斯特朗公理系统的传递律，若有X→A成立，且使得成立，则必有X→B成立。

显然不是最小的FD。

证毕。

注：由此可知DBFD搜索策略中BC属性集内部先计算B→C，再计算BC→A的原因——若B→C成立，则BC→A无须进行计算。

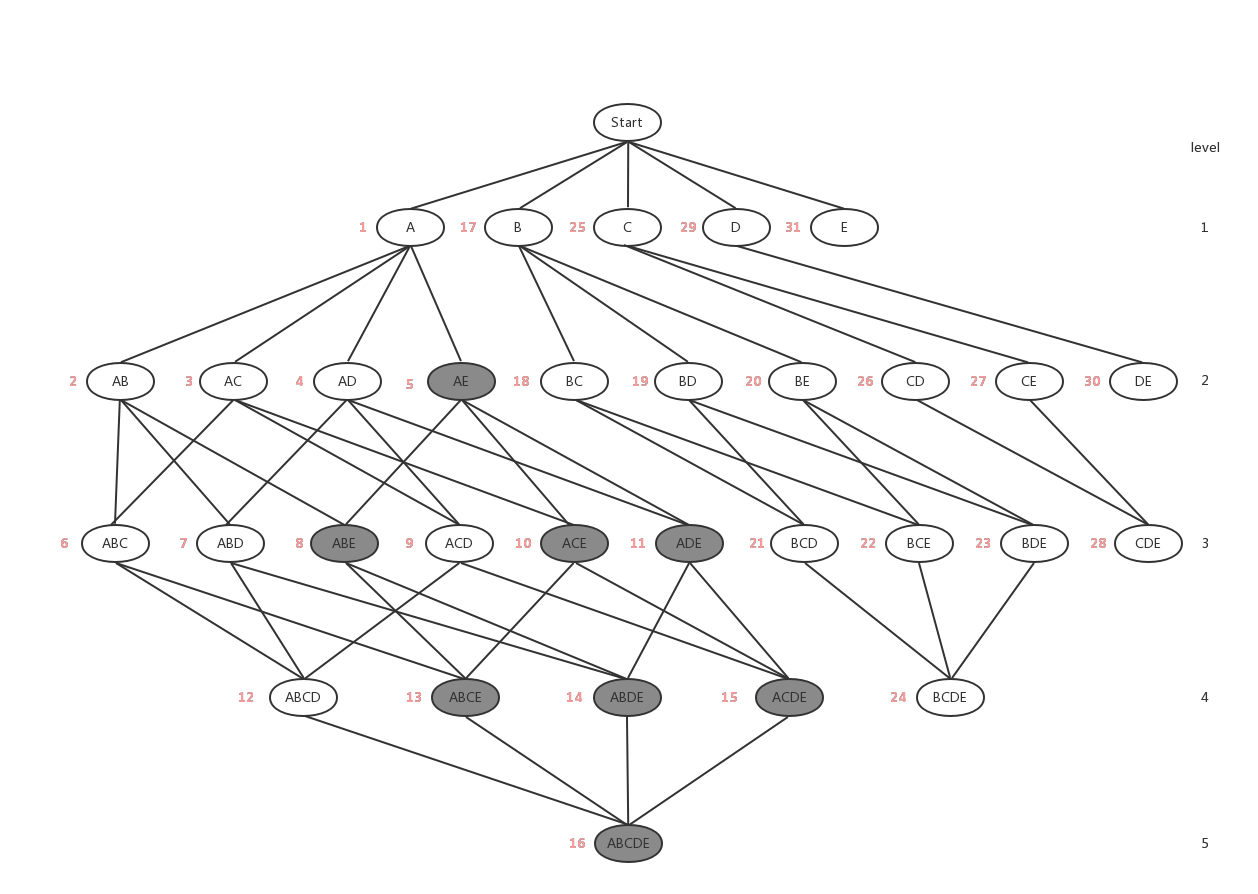


图4.2 DBFD剪枝策略

如图4.2，当计算出5号节点中的函数依赖A→E成立后，可将5号节点AE的所有子节点进行剪枝。

证明如下：

首先，AE的所有子节点等价于包含A、E属性的所有属性集。

要证：函数依赖A→E成立，可对AE节点的所有子节点进行剪枝。

也即证：函数依赖A→E成立，包含A、E属性的属性集内部的候选FD均无须计算。

包含A、E属性的属性集内部的候选FD均无须计算的充要条件是：RHS为A的候选FD无须计算，RHS为E的候选FD无须计算，RHS既不为A也不为E的候选FD无须计算。

所以只需证：

函数依赖A→E成立，RHS为A或为E或既不为A也不为E的候选FD无须计算

即可。

由上述搜索策略可知，5号节点AE的所有子节点均不计算RHS为A的FD。

由于当RHS为E时，LHS势必包含属性A，此时因为A→E成立，所以只要满足，必有X→E成立（由剪枝的重要依据一可知），显然X→E不是最小的函数依赖，所以RHS为E的FD无须计算。

由于当RHS既不为A也不为E时，LHS势必包含属性AE，因为A→E成立，即便使得成立也一定不是最小的FD（由剪枝的重要依据二可知），所以LHS包含AE（也即RHS既不为A也不为E）的候选FD均无须计算。

证毕。

### 4.3.2 右方集剪枝策略

通过以上观察，得出推论（右方集剪枝策略）：

对于任意属性个数大于1的节点，若节点属性集内部除了RHS为根属性的所有候选函数依赖均成立，则可对该节点的所有子节点进行剪枝。

注：节点的根属性即为该节点的根节点的属性，也是该节点属性集的第一个属性。

证明如下：

设内部除了RHS为根属性（O）的所有候选FD均成立的属性集为X（|X|>1）

当|X|=2、O=A时，

内部除了RHS为A的候选FD只有一个：A→X\A

由于，AE节点若满足A→E成立，便可对AE节点进行剪枝（已证）

同理可得，|X|=2、O=A的节点若满足A→X\A成立，则可进行剪枝。

当|X|>2、O=A时，内部除了RHS为根属性的所有候选FD均成立

等价于均有X\B→B成立

所以所有子节点中为RHS的FD均不是最小的（由剪枝的重要依据一可知）

所有子节点中LHS为X的所有候选FD也无须计算（由剪枝的重要依据二可知）

RHS为A的所有候选FD均不在具有属性A的节点计算（先前约定）

综上，|X|>2、O=A的节点若满足均有X\B→B成立，则可进行剪枝。

当|X|=2、O=B时，与O=A不同之处在于，还需要对每个节点判断X→A的候选FD是否成立（若O=C则还需要判断X→A和X→B的候选FD是否成立）。

已知：均有X\B→B成立

，若有X→C成立则一定不是最小的（由剪枝的重要依据二可知）

显然X→A无须计算。

所以，|X|=2、O=B的节点若满足均有X\B→B成立，则可进行剪枝。

同理，当|X|>2、O=B时，若满足均有X\B→B成立，则可进行剪枝。

相似的，当|X|、O=C……

综上可得，若任意属性个数大于1的节点内部除了RHS为根属性的所有候选FD均成立，则可对该节点的所有子节点进行剪枝。

证毕。

上述推论也即右方集剪枝策略。

### 4.3.3 增强右方集剪枝策略

然而DBFD算法采用的是增强右方集（）剪枝策略。

增强右方集剪枝策略是对右方集剪枝策略的进一步优化，具体思想如下：

**对于任意属性个数大于1的节点X（根属性用O表示），若满足,有X\A→A成立或使得X\A\B→B成立，则可对X节点的所有子节点进行剪枝。**

注：O代表泛指并非具体的值。

换言之，对于任意属性个数大于1的节点X（根属性用O表示），若均满足以下条件之一：

1. **X\A→A成立；**
2. **使得X\A\B→B成立；**

即可对X节点的所有子节点进行剪枝。

证明如下：

对于任意属性个数大于1的节点X（根属性用O表示），

（1）若均满足条件a，

此时即为右方集剪枝的充要条件，

自然可对X节点的所有子节点进行剪枝；

（2）若均满足条件b，

因为使得X\A\B→B成立

所以X\A→A即便成立也一定不是最小的（剪枝的重要依据二）

因此X属性集内部的所有候选FD均无须计算

并且当O>A（注意此处的A不同于之前提到的，这里并非泛指而是代指图3中的第一个属性）（表示字典序意义上的大于，即X节点的根属性的字母在字母表中位于A之后）时，X节点还需要计算X→C，C属于字典序小于O的所有属性的集合（例如O=C，X节点还需要计算X→A和X→B）

但由于使得X\A\B→B成立

所以使得X\B→B成立（剪枝的重要依据一）

因此均有X→C不是最小的FD（剪枝的重要依据二），自然也就无须计算

综上，可对X节点的所有子节点进行剪枝。

（3）若均满足条件a或b时，

1. 假设且满足条件a（即X\B→B成立）

则子节点（即属性集包含X的节点）中RHS为B的候选FD均无须计算（剪枝的重要依据一）

并且X→均无须计算（剪枝的重要依据二）；

1. 假设且满足条件b（即使得X\B\C→C成立）

因为使得X\B\C→C成立

那么X\B→B即便成立也一定不是最小的FD（剪枝的重要依据二）

所以子节点（即属性集包含X的节点）中RHS为B的候选FD均无须计算

因为使得X\B\C→C成立

则X\C→C必然成立

所以X→均无须计算（剪枝的重要依据二）；

综合A.B.可得，假设且满足条件a或b，则X的子节点中RHS为B的候选FD和X→均无须计算。

进而可得，若均满足条件a或b时，则X的子节点中除了RHS为O的候选FD和X→均无须计算，因此可对X的所有子节点进行剪枝。

综合（1）（2）（3）可得出结论：

对于任意属性个数大于1的节点X（根属性用O表示），若均满足以下条件之一：

1. **X\A→A成立；**
2. **使得X\A\B→B成立；**

即可对X节点的所有子节点进行剪枝。

证毕。

### 4.3.4 键剪枝

在关系型数据中，普遍存在一种特殊的属性集——超键（super key）。超键是一种能够唯一标志元组的属性集。

通过分析超键的特殊性可得出以下推论（键剪枝策略）：

**若某节点的属性集为该关系模式的超键，则可对该节点的所有子节点进行剪枝。换言之，若X为超键，则对，Y中的候选函数依赖均无须计算。**

证明如下：

由于超键定义可知：

（1）X为超键的充要条件为||的值为元组总数

由子节点与父节点的性质以及超键的性质可知：

（2）任何子节点的属性集Y（X包含于Y）也为超键

提前引用下一小节中的FD判定方法一：

（3）Y\A→A成立的充要条件为

结合（1）（2）（3）可推出：

（4）若存在A使得Y\A→A成立，则代表Y\A也为超键

由（4）可知，若子节点Y中存在A使得Y\A→A成立，那么Y\A必为超键，但由于本算法引入键剪枝策略后，所有Y\A→A（Y\A为超键）将会在Y\A节点计算，因此节点Y上无须对Y\A→A进行计算。

证毕。

注意：

1.键剪枝策略中提到的是对子节点进行剪枝，本节点内部的候选FD还是需要进行判定计算的。

2.若超键X并非候选键，则任意X→A（A∈R\X）一定不是最小的FD。

3.若超键X为候选键，也并不一定代表任意X→A（A∈R\X）是最小FD，因为可能存在B使得X\B→A也成立。

## 4.4 函数依赖判定

分布式函数依赖判定的重要依据[18]：

设、、···、为关系集r的水平切分集，若相同X值的数据不分布于、、···、中（即相同X值的数据均集中于一个切分集），则函数依赖X→A在r上成立的充要条件为X→A在（i=1，2，···，n）上均成立。

由于本算法的数据迁移策略能够保证，每次对当前FD进行判定计算时，此FD的LHS值相同的数据均集中于一个节点。因此只需在各节点执行单机FD判定方法即可。

本小节将直接给出参考文献[3]中的两种FD判定方法，证明过程不再详述。

### 4.4.1 FD的判定方法一

FD的判定方法一：

**函数依赖X→A在关系集r上成立，当且仅当在关系集r上有**

例如表2.1中，

关系r上有，所以函数依赖A→D在r上成立。

### 4.4.2 FD的判定方法二

FD的判定方法二：

**函数依赖X→A在关系集r上成立，当且仅当在关系集r上有细化**

例如表4.1.1中，表示细化了，所以在该关系集上函数依赖X→A成立。

例如表4.1.2中，表示未能细化，所以在该关系集上函数依赖X→A不成立。

图中合并单元格代表这些元组对应的属性值均相同。

表4.1.1 细化

|  |  |  |
| --- | --- | --- |
| Tuple ID | X | A |
| 1 |  |  |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |  |  |
| 7 |
| 8 |  |

表4.1.2 未能细化

|  |  |  |
| --- | --- | --- |
| Tuple ID | X | A |
| 1 |  |  |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |  |
| 7 |  |
| 8 |  |

由于本算法采取的分层搜索策略中，下一层级的FD判定需要使用上一层级的中间计算结果，甚至还需要上一轮整体搜索的中间计算结果，所以本算法选择采用第一种中间计算结果数据规模较小的FD判定方法（对主节点内存要求较小）。

## 4.5 数据迁移策略

在[18]提出的分布式函数依赖计算框架中可以看出，数据迁移是很重要的一个环节。预先进行适当的数据迁移，能够十分有效地减少分布式FD算法中的Shuffle操作。

假设有关系数据集r（如表4.2），以及三个工作节点Node1、Node2和Node3。将该数据集r水平切分后，Node1将读入ID为1，2，3，4的数据。接下来将对Node1中的数据迁移方式进行举例说明。

表4.2 关系数据集r

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ID | A | B | C | D | E |
| 1 | 10 | 1 | 1 | 1 | 1 |
| 2 | 12 | 1 | 2 | 1 | 2 |
| 3 | 12 | 2 | 3 | 1 | 3 |
| 4 | 3 | 2 | 3 | 1 | 4 |
| 5 | 4 | 4 | 3 | 1 | 5 |
| 6 | 1 | 4 | 4 | 1 | 6 |
| 7 | 2 | 3 | 4 | 1 | 7 |
| 8 | 7 | 3 | 2 | 1 | 8 |
| 9 | 9 | 5 | 5 | 1 | 9 |
| 10 | 8 | 5 | 7 | 1 | 10 |
| 11 | 5 | 1 | 6 | 1 | 11 |
| 12 | 6 | 1 | 1 | 1 | 12 |

根据4.2.2整体搜索策略可知，本数据集的搜索顺序为E、A、C、B、D，也即数据重分布顺序。因此先根据E属性值（E属性值取节点数3的余数）进行数据重分布，Node1上的数据迁移过程如图4.3.1。

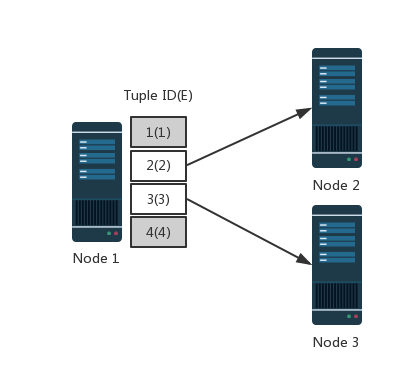
****

图4.3.1 Node1上依据E属性值进行数据迁移

图中灰色数据代表无须迁移。

同上，Node2中将ID为7的数据迁移到Node1，ID为6迁移到Node3；Node3中将ID为10的数据迁移到Node1，ID为11迁移到Node2。

因此经过第一次依据E属性值进行数据重分布后，Node1中新的数据对应的ID为1、4、7和10。

接下来将根据A属性值对Node1进行第二次数据迁移，如图4.3.2:

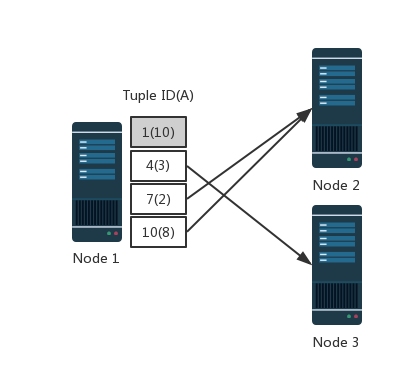


图4.3.2 Node1上依据A属性值进行数据迁移

同理，容易推出第三、四、五次分别依据C、B、D属性值进行数据重分布的过程，因此不再赘述。

## 4.6 算法整体流程图

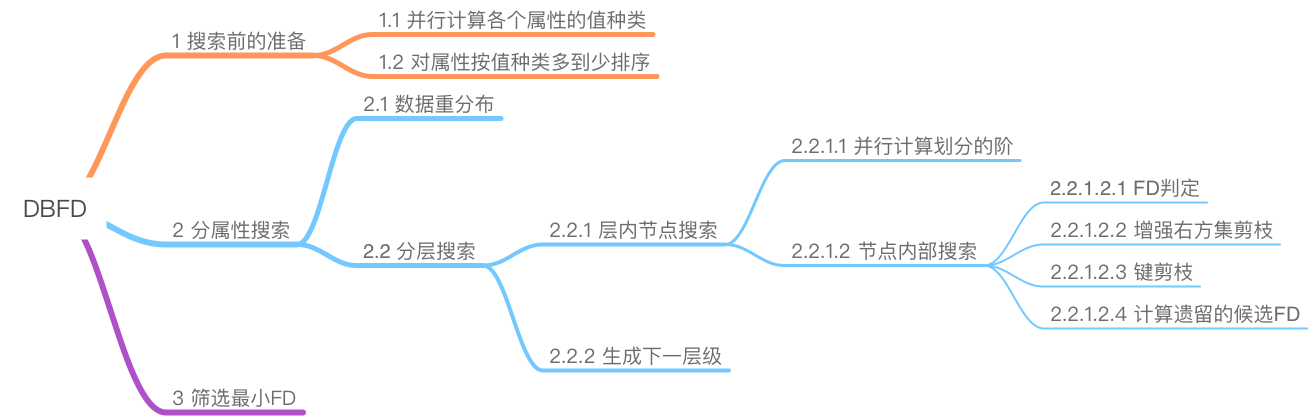


图4.4 DBFD算法整体流程

# 5 算法实现

本章将重点介绍DBFD算法的数据结构以及实现代码。

## 5.1 数据结构

### 5.1.1 存储关系数据集r

r: RDD[Array[Int]]

解释说明：RDD[19]代表需要进行分布式处理计算的数据集。Int代表每个属性的哈希值，之所以对每个原始属性的值进行hash转换有两个原因，如下：

a.减少数据规模（由于原始数据的大多数属性值的字符平均个数大于4，也即大于Int值所占用的4字节空间），进而减少数据迁移的时间，以及减少CPU进行划分（Partition）求阶计算的时间

b.对数据进行重分布时，需要根据相应属性值的哈希值才能尽可能地保证数据的均匀分布，进而保证负载均衡

注意事项：

当某一属性A的取值种类（范围）多达，必然会产生哈希碰撞，进而使得FD判断不准确。其实，即便是属性值只有两种，也存在哈希碰撞的可能性。这里给出取值种类有n(n<)种的属性A产生哈希碰撞的概率：

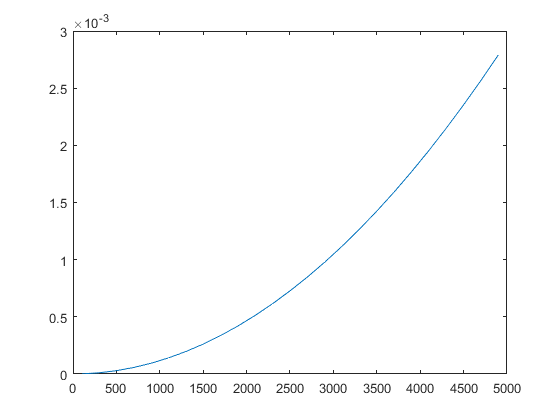


图5.1 哈希碰撞概率

当n=2时约为四十三亿分之一

据计算本文采用的实验数据中属性值种类最大值不超过4000。

由图5.1可知，当n小于4000时，A属性值产生碰撞的概率低于千分之2，因此可忽略不计。

即便产生碰撞，也只可能会影响RHS为A且成立的FD，以及LHS包含A且不成立的FD。换言之碰撞不会影响RHS为A且不成立的FD，以及LHS包含A且成立的FD。

证明如下：

X→A不成立，等价于（FD的判定方法一）：

若属性A产生一次碰撞则可能会导致的值比真实值少1，由于的性质存在，所以：

若A值发生碰撞后仍有，则可推出X→A不成立

若XA→B成立，等价于：

若属性A产生一次碰撞则可能会导致的值比真实值少1，并且当的值确实比真实值少1时才有可能使得的值也比真实值少1，所以当时，发生碰撞后不可能有，所以：

若A值发生碰撞后仍有，则说明碰撞前该等式也一定成立，即 XA→B成立

证毕。

通过观察以上证明过程，我们可以发现即便产生一次碰撞，对FD发现的影响也十分微小。只有在非常不幸的情况下（由于此概率变量太多、结果不够直观因而未对其进行计算）才会导致FD的判断错误。

因此，本算法约定：

属性值均为4字节哈希值，且忽略哈希碰撞。

（声明：并不代表本算法主体思想必须建立在哈希转换的基础上，只是额外提供了一种通过牺牲微小的准确性从而大幅提高效率的手段）

### 5.1.2 表示属性及其映射

R: Array[Int]

解释说明：由于实验数据文件中往往去除了本应该有的第一行属性名称。所以算法中使用Int值代表属性。也即0、1、2···分别代表关系模式中的第0、1、2···个属性。

O: Array[Int]

解释说明：在4.2.1中，提到了按照属性种类多少对属性进行排序，由于我们直接更改r中所有属性值的顺序代价太大，所以使用一个映射关系来代表新的属性序列。这样使得对搜索算法进行设计时完全不受新排序的影响，只需要使用R即可；只有在访问具体属性值时（进行划分求阶计算时）才会使用新的顺序，此时也只需要将R的属性经过一次映射（通过O数组）即可得到排序后的相应位置的属性。

例如：若关系模式中的第0、1、2、3个属性经过排序后为2、1、3、0，即R中的值为0、1、2、3，O中的值为2、1、3、0。

在执行搜索策略时，只需要使用0、1、2、3的顺序即可（可以假设r中的属性位置已经进行了调整），当真正去访问第0个属性具体的值的时候，则使用数组O进行一次映射由0得到2，因此实际访问的是r中的第2个属性。

### 5.1.3 记录已发现的FDs

FDs: Set[String]

解释说明：由于本算法局部对FD的计算采用的是LHS中属性个数从少到多的顺序，所以能够在局部保证发现的FD均为最小的，若已发现的FD在整体也最小的充要条件是LHS比该FD少了第一个属性的FD不成立。

最小 FDs的个数的理论最大值为：(本人有一种巧妙的办法可以证明，但由于过程过于冗长，本文直接附上此结论)

### 5.1.4 存储当前层级的属性集

L: Array[String]

解释说明：代表当前层次的所有属性集。每个属性集用字符串表示，例如第1、2、4列属性组成属性集表示为“1，2，4”。由于查询某属性集的增强右方集以及划分的阶均需要使用字符串作为键去访问它的值。

使用有序数组存储是为了快速生成下一层级，具体原因见[17]中生成下一层级的策略或第六章中生成下一层级伪代码。

### 5.1.5 存储属性集对应的增强右方集

: Map[String, Array[Int]]

解释说明：存放属性集对应的增强右方集。

通过属性集对应的字符串去查询显得一目了然。

### 5.1.6 记录已计算的划分的阶

P: Map[String, Int]

解释说明：存储了所有已计算的划分的阶。

使用键值对的形式存储属性集划分的阶，能够大大提升查询效率。

### 5.1.7 最大内存占用

主节点最大内存占用：

以上数据除了r均存储在主节点内存中，据估计最大内存占用与数据集r的属性个数(即|R|)呈指数关系，因此当属性达到一定数量势必会造成主节点的内存溢出。但由于本文只采用10个属性进行计算，并且经过粗略估算一定不会出现内存溢出的现象，所以本文暂不对主节点最大内存占用进行详细计算。

工作节点内存占用：

由于数据集r均匀分布在各个工作节点，且工作节点计算过程中并不会产生大量新的数据，因此每个工作节点的内存占用大约为：r的大小/节点数。x千万条元组10个属性构成的r大小为： GB。因此n个工作节点，计算x千万条元组，每个工作节点内存占用约为 GB。据估算，若采用1个配备4 G内存的工作节点计算1.1亿条10属性元组则刚好会发生内存溢出。

注意，Spark默认设置每个节点内存使用上限为：当前节点的物理内存

## 5.2 伪代码

本小节将使用伪代码实现DBFD算法主要内容。建议结合图4.4查看。

### 5.2.1 搜索前的准备

**Procedure** PrepareForSearching

1 for each do

2 = = ParallelComputePartition (A)

3 end for

4

5 Sort O by

上述代码是对4.2.1的实现。第1行的R和第4行的O已在5.1.2 中进行了介绍，第2行的P则是在5.1.6中进行了介绍，P集合用来存储所有已计算的划分的阶。第5行则是按照O中元素对应的划分的阶，从大到小对O中元素进行排序。ParallelComputePartition (A)的实现如下：

输入属性A

/\*在n个工作节点并行执行以下操作\*/

//假设当前为第j个工作节点

计算出

将发送给主节点

/\*在主节点执行以下操作\*/

根据，，···，计算出

输出

### 5.2.2 数据重分布

**Procedure** ExchangeDataWith()

输入：第i个属性

/\*在n个工作节点并行执行下面代码\*/

//假设当前为第j个工作节点

1 for each do

2

3 if then

4

5 end for

上述代码是对4.5的实现。第1行代码中代表当前工作节点上的关系数据集，t代表数据集中的一条元组；第2行代码中代表元组t上属性对应的哈希值，x是对该值取n的余数；第4行表示将元组t迁移到第x个节点

### 5.2.3 并行计算划分的阶

**Procedure** ParallelComputePartition (X)

///此处不同于5.2.1中的ParallelComputePartition(A)

输入属性集X

/\*在n个工作节点并行执行下面代码\*/

//假设当前为第j个工作节点

1

2 for each do

3

4 end for

5

将计算结果发送给主节点

/\*然后在主节点执行\*/

输出

上述代码即4.1.2中提到的划分求阶操作，是DBFD算法耗时最多的操作。第1行中的S集合用于存放属性集X对应的值；第3行表示的是将属性集X对应的值(具有|X|个字段的元组)加入集合S。

### 5.2.4 FD判定

**Procedure** JudgeFD(X)

输入属性集X

1 for each do

2 if then //即时

3

4 delete A from

5 delete all R\X from

6 end if

7 end for

上述代码是对4.4.1的实现。第4、5行代码的逻辑来源于4.3.3。

### 5.2.5 增强右方集剪枝

**Procedure** RHSPlusPruning(X)

输入属性集X

1 if then

2 delete X from //增强右方集剪枝

3 end if

利用4.3.3中的思想进行剪枝。

### 5.2.6 键剪枝

**Procedure** KeyPruning(X)

输入属性集X

1 if then

2 for each do

3

4 end for

5 delete X from //键剪枝

6 end if

利用4.3.4中的思想进行剪枝。第1行表示当X属性为超键时。

### 5.2.7 计算遗留的候选FD

**Procedure** ComputeRemainingRHS(X)

输入属性集X

1 for each do

2 if then //即若没有计算过时

3 = ParallelComputePartition()

4 end if

5 if then

6

7 delete A from

8 end if

9 end for

由4.2.4可知，还需要对本节点计算所有LHS为X、RHS为(k<i)的FD。第3行的ParallelComputePartition即5.2.3中的；第5行利用的是函数依赖判定的充要条件。

### 5.2.8 生成下一层级

**Procedure** NextLevel() //参考自[17]中Apriori算法采用的生成候选集策略

if then

for each do

end for

else

for each do

for each , do

if for all , then

//初始化为全集

for each do

end for

end for

end for

return

### 5.2.9 筛选最小FD

**Procedure** PrintMinFD(FDs)

输入已发现的函数依赖集合FDs

1 for each do

2 if then

3 delete FD from FDs

4 else

5 print(“已发现最小FD：”+FD)

6 end if

7 end for

上述代码将会输出所有最小的FD。第2行的依据来自5.1.3。

# 6 实验结果分析

## 6.1 实验准备

### 6.1.1 实验环境

本实验采用的Spark集群由8台虚拟机通过局域网连接构成，每台虚拟机均配备了2.8 GHz Intel Core i7处理器(单核双线程)和4 GB内存，系统均为CentOS 7 64位。DBFD算法由Scala语言实现，使用sbt（Simple Build Tool）构建项目并生成jar包，运行于Spark平台。Java版本为1.8.0，Scala版本为2.12.3，sbt版本为1.1.4，Spark版本为2.3.0。实验数据最初存储于HDFS系统，每次实验前，先将数据水平均匀切分并传输到需要用于计算的服务器（工作节点）的内存中（而不是磁盘），这是为了模拟分布式数据的场景。因此实验结果将不会计入首次从HDFS读取数据的时间。

### 6.1.2 实验数据

本实验的大规模关系数据集来源于美国运输部网站[20]，从网站的大量真实数据中截取2亿条元组的10个属性值构成了本实验的原始数据集。

### 6.1.3 实验设计

设计了两组控制单一变量的实验：

第一组：依次使用1、3、5、7台服务器作为工作节点，对相同的数据集(0.5亿条元组)进行计算。

第二组：使用4台服务器作为工作节点，依次对含有0.5、1、1.5、2亿条元组的数据集进行计算。

使用1台服务器作为主节点，对上面两组实验分别执行TANE、FDPar\_Discover、DBFD算法。

另外设计了一组数据顺序变更实验：

使用4台服务器作为工作节点，将含有0.5亿条元组的数据集的第2列属性值与第1列交换位置，使用1台服务器作为主节点，分别执行FDPar\_Discover和DBFD算法。

附加说明：

1.执行TANE算法的完整过程：先将每个工作节点上的数据全部迁移至主节点，然后在这主节点集中执行TANE算法。

2.采用n台虚拟机执行FDPar\_Discover或DBFD算法，将会选取1台作为主节点执行搜索等操作，其它n-1台作为工作节点进行划分求阶操作，耗时计算与数据迁移均发生在工作节点上。

## 6.2 实验结果

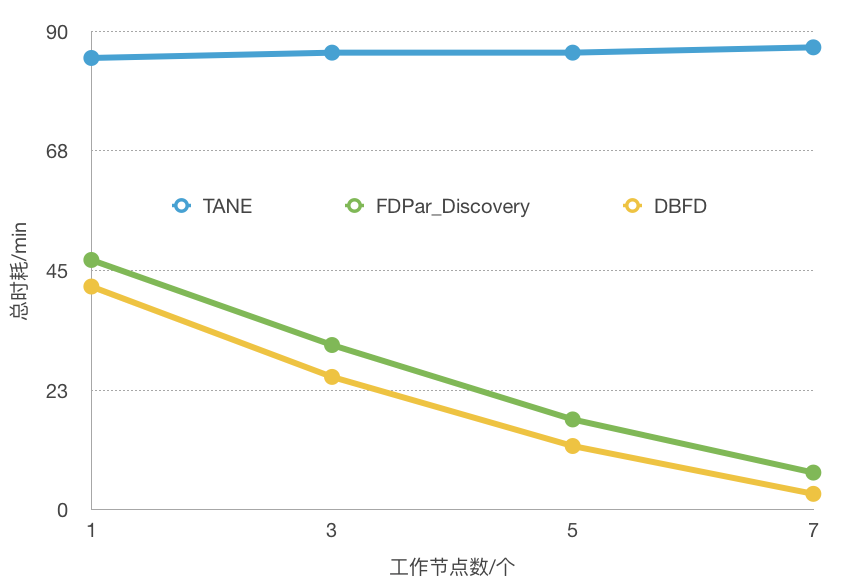


图6.1 实验一：节点扩充

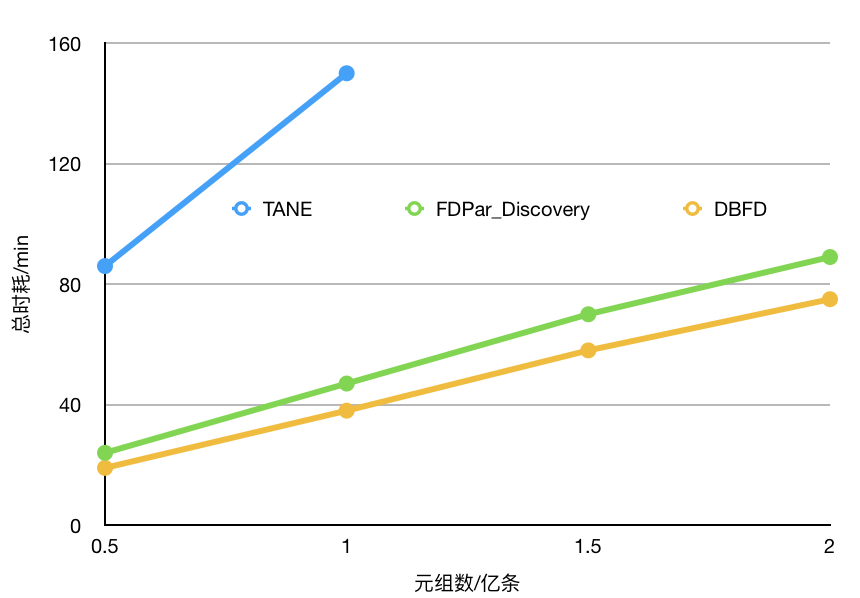


图6.2 实验二：数据扩展

表6.1 实验三：数据顺序变更

|  |  |  |
| --- | --- | --- |
| 算法 | 变更前 | 变更后 |
| FDPar\_Discover | 24min | 19min |
| DBFD | 41min | 19min |

## 6.3 理论分析

### 6.3.1 分析节点扩充对算法的影响

图6.1中，可以看出工作节点扩充对TANE算法没有任何性能提升，反而会略微增加时耗。这是因为TANE算法的计算只发生在主节点，增加工作节点并不能提升主节点的计算性能，反而会使数据迁移的时耗略微增加。相反的，FDPar\_Discover和DBFD算法由于节点扩充，计算效率得到显著提升，并且由4.1.1可知每次进行数据重分布的时耗Y与工作节点个数X的关系为：（A表示总的数据量，单位为GB），因此还能有效降低数据迁移时耗，最终必然使得总时耗明显下降。

### 6.3.2 分析数据扩展对算法的影响

图6.2中，可以看出三种算法时耗均随着数据规模增大而增加，但FDPar\_Discover和DBFD算法在数据规模较大时表现出来的优势更大，DBFD更为明显。值得注意的是，当数据规模超过1亿条元组时主节点执行TANE算法则会导致内存溢出程序终止。

### 6.3.3 分析数据顺序变更对算法的影响

由表6.1中的结果可以看出，FDPar\_Discover算法会受属性顺序的影响，这是由于3.2小节中的问题2所致，经过计算发现顺序变更前第一列属性值种类有2825种，第二列属性值种类有15种。然而DBFD算法使用了4.2.1中的策略，保证了计算不受属性顺序变更的影响。

### 6.3.4 DBFD的性能优势

综上可见，DBFD算法相比于TANE，在处理大规模数据时由于充分利用了多台计算资源因而性能优势尤为明显，并且当数据达到一定规模时，TANE算法会出现内存溢出，DBFD则不会。相比于FDPar\_Discover算法，DBFD采取了一种简单但十分有效的措施（4.2.1）保证了各个工作节点负载均衡，此外DBFD只发现最小的FD，不仅能够采取更有效的剪枝策略从而减少计算，而且能够减少中间计算结果的数据量。

结 论

本算法4.5小节中的数据迁移策略完美解决了3.1小节中提及的分布式（准确性）问题；4.3小节中的剪枝策略以及5.1.1中的数据压缩策略很好地缓解了3.1小节中提及的大数据（性能）问题；4.2.3中的局部（层次）搜索策略以及4.2.4中的属性内部搜索策略尽量保证了已发现的FD为最小的，再通过5.2.7的简单筛选，使得输出的FD一定为最小的，并且在4.3小节中采取了更为高效的剪枝策略，本文处处考虑了如何更适和发现最小FD，因此本算法主要是为了解决3.2小节中提到的问题1。针对3.2小节中的问题2，本文在4.2.1搜索前的准备工作中提出了完美的解决方案，在4.5小节的举的例子中也可以看出其效果显著。

但本算法也存在不足之处，图4.2表示的搜索顺序中，如果在16号节点计算出函数依赖ABCD→E不成立，那么就没有必要在20、22、23、24、27、28、30号节点内部计算RHS为E的候选FD，因为这些候选FD一定不成立。这是由于分布式FD发现算法在局部采用自顶向下[21]的发现策略造成的问题。非常惊奇地发现，3.2小节的问题1 正是由于[10]中分布式FD发现算法在局部上采用自底向上[21]发现策略所致，也就是说本算法的不足之处正是由于解决了[10]中的不足之处所产生的。因此可得出一个重要结论：分布式函数依赖发现算法不论采用自顶向下还是自底向上的发现策略，均会造成不必要的计算。针对这两种对立问题(解决其中一种便会出现另一种的互斥问题)，将在ADBFD算法中提出一种一劳永逸的解决方案。

ADBFD算法主要思想是：在第一轮局部搜索过程中采用自顶向下的最小FD发现算法——DBTANE(5.2.3节)，此外需要记录所有已判定为不成立的最大FDs；在第二轮局部搜素过程中采用自底向上的最小FD发现算法，并根据上一轮中记录的不成立的最大FDs进行相应剪枝，同时记录本轮所有判定为不成立的最大FDs；在第三轮局部搜索过程中仍采用自底向上的最小FD发现算法，根据上两轮中记录的不成立的最大FDs进行剪枝，并记录本轮不成立的最大FDs；以此类推，在第一轮中采用自顶向下的FD发现算法，使用成立的最小FDs进行剪枝，同时记录不成立的最大FDs；在之后每一轮中采用自底向上的FD发现算法，使用之前轮次中记录的不成立的最大FDs进行剪枝。

ADBFD是DBFD的高级版本，也是对[10]中FDPar\_Discover算法的高级改进版本。但由于ADBFD算法的复杂性较高，作者时间十分有限，因此未能在本文中详细讨论此算法的具体思想以及代码实现，对此深感抱歉。

致 谢

我首先要感谢我的论文指导老师、南京理工大学计算机科学与技术学院的蔡志成老师。蔡志成老师对我论文的研究方向做出了指导性的意见和推荐，在论文撰写过程中及时对我遇到的困难和疑惑给予悉心指点，提出了许多有益的改善性意见，投入了较多的心血和精力。在此对蔡志成给予我的帮助表示诚挚的谢意!同时，还要感谢南京理工大学计算机科学与技术学院软件工程专业的郑越辉同学，在实验环境搭建的过程中十分热心地提供了帮助。

此外，还要感谢朋友以及同学们在论文编写中给予的照顾和帮助，给我提供了绝佳的写作环境。也要感谢参考文献中的作者们，正是因为他们的不懈努力与付出，为本文提供了充实的理论基础，本论文才有机会顺利完成。更要感谢提供了一系列开源软件的Apache软件基金会以及提供了免费绘图工具的ProcessOn网站。

最后，谢谢论文评阅老师们的辛苦工作。衷心感谢我的家人、朋友，以及同学们，正是在他们的鼓励和支持下我才得以顺利完成此论文。

参 考 文 献

[1] 赵辉, 王黎明, ZHAOHui,等. 一个基于网格服务的分布式关联规则挖掘算法[J]. 小型微型计算机系统, 2006, 27(8):1544-1548.

[2] 张方舟, 高晓松. 基于条件函数依赖的挖掘算法研究[J]. 计算机技术与发展, 2015(5):56-59.

[3] Huhtala Y, Kärkkäinen J, Porkka P, et al. Tane: An Efficient Algorithm for Discovering Functional and Approximate Dependencies[J]. Computer Journal, 1999, 42(2):100-111.

[4] Novelli N,Cicchetti R. Fun: An efficient algorithm for embedded dependencies [C] //Proc of mining functional and the 8th Int Conf of Database Theory, New York: ACM, 2001: 189-203.

[5] Flach P A, Savnik I. Database Dependency Discovery: A Machine Learning Approach[J]. Ai Communications, 1999, 12(3):139-160(22).

[6] Wyss C,Giannella C,Robertson E,FastFDs: A heuristicdriven, depth-first algorithm for mining functional dependencies from relation instances[C]//Proc of the 3rd Int Conf on Data Warehousing and Knowledge Discovery, New York: ACM,2001: 101-110.

[7] Lopes S,Petit J,Lakhal L. Efficient discovery of functional dependencies and armstrong relations [C]//Proc of the 7th Int Conf on Extending Database Technology, New York: ACM,2000: 350-364.

[8] Hong Yao,Howard J,Hamilton. Mining functional dependencies from data[J]. Data Mining and Knowledge Discovery,2008,16(2): 197-219.

[9] Li W, Li Z, Chen Q, et al. Discovering Functional Dependencies in Vertically Distributed Big Data[J]. 2015.

[10] 李卫榜, 李战怀, 陈群,等. 分布式大数据函数依赖发现[J]. 计算机研究与发展, 2015, 52(2):282-294.

[11] Zaharia, Matei, Xin, Reynold S, Wendell, Patrick, et al. Apache Spark: a unified engine for big data processing[J]. Communications of the Acm, 2016, 59(11):56-65.

[12] Kolahi S. Functional Dependency[M]. Springer US, 2009.

[13] 孙家昶. 网络并行计算与分布式编程环境[M]. 科学出版社, 1996.

[14] Shvachko K, Kuang H, Radia S, et al. The Hadoop Distributed File System[C]// IEEE, Symposium on MASS Storage Systems and Technologies. IEEE Computer Society, 2010:1-10.

[15] 陈英芝. Spark Shuffle的内存调度算法分析及优化[D]. 浙江大学, 2016.

[16] Park J S, Chen M S, Yu P S. An effective hash-based algorithm for mining association rules[M]// ACM SIGMOD Record. ACM, 1995:175-186.

[17] Zhang Z R, Xie K L, Zhang Z N. A Fast Algorithm for Mining Association Rules[J]. Journal of Shanghai Jiaotong University, 2002, 15(6):619-624.

[18] Ye F, Liu J, Qian J, et al. A Framework for Mining Functional Dependencies from Large Distributed Databases[C]// International Conference on Artificial Intelligence and Computational Intelligence. IEEE Computer Society, 2010:109-113.

[19] Zaharia M, Chowdhury M, Franklin M J, et al. Spark: cluster computing with working sets[C]// Usenix Conference on Hot Topics in Cloud Computing. USENIX Association, 2010:10-10.

[20] United States Department of Transportation [OL]. [2017-12-30]. http://apps.bts.gov/xml/ontimesummarystatistics/src/index.xml.

[21] Liu J, Li J, Liu C, et al. Discover Dependencies from Data—A Review[J]. Knowledge & Data Engineering IEEE Transactions on, 2012, 24(2):251 - 264.