**基于CPU或GPU集群的“数据族谱”**

**技术方案设计及实现**

V1.0

设计人: 张宇宁(总设计、需求分析)

执行人：罗锋(设计、方案、代码)、胡新平(架构)

参与人：申传华、陈志伟、冷露露

部 门：中电智能-研发组-新能源大数据项目组

按信息公司领导指示精神，新能源大数据项目成员群策群力基于《新能源风电（光伏）新编码》在故障诊断模型和展示上的应用需求上开展研究，初步形成了技术实验方案，并正编写基于GPU集群的验证代码编写编写工作，代码上传github（部分完成，持续开发中）

https://github.com/lfforai/ML\_Foundation\_Mathematic/tree/master/.idea/zdzn\_family\_tree/GPU/famile\_tree。技术目标是仅依赖编码，在不需要人工参与情况下，由算法自动实现数据族谱建模，具体设计方案如下：

**一、数据族谱工具应用的基础数据前提**

假设源点数据均已经采用《新能源风电（光伏）新编码》进行编码,编码样例如果下：

@MDNY.SZWQ.11MD#92\_B#1\_SK#1.RtPwrHi

@MDNY.SZWQ.11MD#100\_A#1\_LK#1.RtdRttSpdSP

@MDNY.SZWQ.11MD#100.PF

@MDNY.SZWQ.11MD#99\_B#1\_SK#1.RtPwrHi

样例分解：

例 如：

描 述：新疆,布尔津风电场1期11号风机传动变速系统齿轮箱\_齿轮温度

编 码：XXX.XXXXXX.11MD#11\_K#1\_CLX#1.GeaOilTmp#1

#表示设备（属性）序号

**二、数据族谱工具的目的和应用场景**

目前基于传感器的源点设备均是按key-value的格式存储，按新编码编辑的key对比老编码最大的优势是—“体现了数据的设备层级性”，但如果仅只有新的key编码而没有解析和整合key之间关系的算法和模式，也是无法实现层级化的web展示和远程故障诊断数据建模指标筛选的，所以我们研究的主要目的就是为了解决和实现上述实际应用需求。

1. **族谱关系的定义**

数据族谱:是指各key之间按设备组织成“树型”，能够区分不同设备层级的一种数据表

现结构。它必须基于两点实现。

**1、首先是基于单个key值血缘分解**

对Key：MDNY.SZWQ.11MD#92\_B#1\_SK#1.GnRrrSpd而言，家庭族谱为

1）级祖先：MDNY 1级祖先的直接后代：MDNY.SZWQ

2）级祖先：MDNY.SZWQ 2级祖先的直接后代：MDNY.SZWQ.11MD

3）级祖先：MDNY.SZWQ.11MD 3级祖先的直接后代：MDNY.SZWQ.11MD#92\_B

4）级祖先：MDNY.SZWQ.11MD#92\_B 4级祖先的直接后代：MDNY.SZWQ.11MD#92\_B#1\_SK#1

5级祖先：MDNY.SZWQ.11MD#92\_B#1\_SK#1 5级祖先的直接后代：Key

1. **其次是对所有key建立“族谱树”**

MDNY

SZWQ1 | SZWQ2 | SZWQ3

SZWQ1.11MD#1 SZWQ1.11MD#2 | SZWQ2.11MD#1 SZWQ2.11MD#2 | SZWQ3.11MD#1 SZWQ3.11MD#2

**... ... ... ... ... ...**

树的非叶子节点：由每个Key血缘分解出的各级祖先

树的叶子节点：所有的源点Key值

差异：非叶子节点说明它下面还有可展开的子设备

叶子节点是当前设备的属性值（模型拟合使用的直接数据）

1. **按设备层级化的web页面查询展示**

基于树的web数据展示举例，用户在web搜索栏中输入“北京.四子王棋.风机#1”，web解析以后将结果按树型的格式展示在web界面上。

举例：北京.四子王棋.风机#1：风机#1.风机温度---叶子节点

风机#1.风机功率---叶子节点

风机#1.传动#1---非叶子节点

风机#1.变桨#1---非叶子节点

考虑到查询速度和实现难度，一次只展示当前设备的下一级所有的叶子节点和非叶子节点，也就是当前被查询设备的所有直接“后代”，不展示其全部后代子树。当用户点击非叶子节点时候再逐层提交，逐层展示。

**（三）基于远程诊断模型建模指标筛选**

远程故障诊断一般会有几种数据需求方式：

多设备-当前层级多指标：用于拟合模型或者做多指标相关分析

多设备-当前层级和其子设备多指标

单设备-当前层多指标：用于查看该设备是否存在异常值

单设备-当前层多和其子设备多指标

1、单设备举例

北京.四子王棋.风机#1：风机#1.风机温度---叶子节点

风机#1.风机功率---叶子节点

风机#1.传动#1\_油温#1---叶子节点

2、多设备举例

北京.四子王棋.风机#1： 风机#1.风机温度---叶子节点

风机#1.风机功率---叶子节点

风机#1.传动#1\_油温#1---叶子节点

北京.四子王棋.风机#2：风机#2.风机温度---叶子节点

风机#2.风机功率---叶子节点

风机#2.传动#1\_油温#1---叶子节点

北京.四子王棋.风机#3：风机#3.风机温度---叶子节点

风机#3.风机功率---叶子节点

风机#3.传动#1\_油温#1---叶子节点

3、需求特征

对比web展示功能，在模型拟合时候建模人员更关注：

\*）更多关注属性值;

\*) 更多需求多台设备;

\*) 对展示查询设备的下属子非叶子节点展示需求不大;

\*) 查询更具有范化性：用“北京.四子王棋.风机.齿轮箱”而不是“北京.四子王棋.风机#1.齿轮箱#1”。

总体看，对于数据分析者来说，一堆分析指标不管其内部层级如何，只要能归类到特点的一个大层级下即可。

三**、数据族谱的存储架构和模型原理**

（一）数据族谱实际应用的难点

**查询效率**是数据族谱落地实现的最大难点，如果不考虑效率直接用SQL语句对现有key进行直接检索查询也是可以的，此时就不用去做族谱分解和构造族谱树了，费力构造这些多余工作步骤的目的只有一个—加速查询过程。数据族谱是以“树”的逻辑结构为实现基础的，但当我们面对的数据源点特别大或者族谱深度较深（树深度deep很大）的时候，整个查询加载和返回的时间可能会是难以忍受的，所有数据族谱的实现需要慎重解决选择“哪种数据结构”和“使用哪种工具”的问题。

（二）两种数据族谱的实现设计

这里我们通过考虑实现难度、接口通用性和效率性等因素提出了两种不同的“数据族谱”计算机解决方案：

**第一部分 基于cpu和SQL数据库的实现方案**

该套方案优点是基于关系数据库,实现可以通过SQL查询编写，适用性强，接口通用性强，关系表易于维护和扩展，对开发人员技能要求不会太高。缺点是对于测点数据量较大情况，可能存在查询数据速度较慢的问题，特别是未来面对跨场站甚至是跨运营中心族谱查询的超大数据量效率可能会成为严重瓶颈。

**一、算法基本思路：**

（1）数据血缘分解：基于cpu多线程或者单线程对每个key分解出各级别祖先过程。

详见：第二部分的第1小结中的“**key值血缘分解”**

（2）将分解出的数据祖先形成“faher-son”数据pair对。

例如：(MDNY ，MDNY.SZWQ)

(MDNY.SZWQ.11MD,MDNY.SZWQ.11MD#92\_B)

(MDNY.SZWQ.11MD#92\_B#1\_SK#1,Key)

（3）在hive或者MYSQL等中存储“faher-son”数据对，形成基于关系数据库的**“族谱树”**数据结构模型。

基于关系数据库的“族谱树”底层数据格式表

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ID | 父 | 子 | 中心 | **场站** | 子是否 属性 | 族谱 层级 | 是否 地址 | 父描述 | 所属key | 子描述 |

以MDNY.SZWQ.11MD#92\_B#1\_SK#1.GnRrrSpd为例子：

#1)Pair对 (MDNY.SZWQ.11MD , MDNY.SZWQ.11MD#92\_B)存储如下：

ID：×××××××数字

父：MDNY.SZWQ.11MD ; 子：MDNY.SZWQ.11MD#92\_B;

中心：北京; 场站：四子王棋

子是否属性:否 族谱层级：3

是否地址：否 父 描 述：北京.四子王棋.风机

子 描 述：北京.四子王棋.风机.发电机系统

所属key：MDNY.SZWQ.11MD#92\_B#1\_SK#1.GnRrrSpd

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#2)Pair对 (MDNY.SZWQ.11MD#92\_B#1\_SK#1 , key)存储如下：

ID：×××××××数字

父：MDNY.SZWQ.11MD#92\_B#1\_SK#1 子：MDNY.SZWQ.11MD#92\_B#1\_SK#1.RtPwrHi

中心：北京; 场站：四子王棋

子是否属性:否 族谱层级：5

是否地址：否 父 描 述：北京.四子王棋.风机.发电机系统.双馈发电机

子 描 述：北京.四子王棋.风机.发电机系统.双馈发电机.发电机转速

所属key：MDNY.SZWQ.11MD#92\_B#1\_SK#1.GnRrrSpd

注：需要注意的是其中的“父描述”和“子描述”很重要，主要用于一些查询的模糊检索，

这些描述也不需要人工填写，需要先按照《新能源风电（光伏）新编码》在数据库建立一

个代码对照表就可以对照分解出。例如：MDNY.SZWQ.11MD#92\_B#1\_SK#1.GnRrrSpd，自动对应分级为MDNY：北京，SZWQ：四子王棋，11MD：风机，B：发电设备，SK：双馈风机，GnRrrSpd：发电机转速。

**二、用于查询web界面展示的算法原理及返回格式**

查询算法原理：**横向遍历**当前节点的下所有层子节点，输出所有叶子节点和非子节点

返回数据格式：使用Json

基础格式：{F：×××，S：×××，V：×××}

其中F存放父节点值，S存放非叶子节点值，V存放叶子节点值

#1）多设备查询：返回的只有非叶子节点

{F：MDNY.SZWQ.11MD，S：[{F:MDNY.SZWQ.11MD#1，S：null，V：null},

{ F:MDNY.SZWQ.11MD#2，S：null，V：null}.....]，V：null}

#2）单设备查询：返回的只有非叶子节点

{F：MDNY.SZWQ.11MD#1，S：[{F:MDNY.SZWQ.11MD#1\_A#1，S：null，V：null},

{F: MDNY.SZWQ.11MD#2\_B#1，S：null，V：null}.....]，V：null}

#3）单设备查询：返回有叶子节点，也有非叶子节点

{F：MDNY.SZWQ.11MD#1\_B#1\_ZQ#1，

S：[{F:MDNY.SZWQ.11MD#1\_B#1\_ZQ#1\_DZ#1，S:null，V:null},

{F:null,**S:**null,V: MDNY.SZWQ.11MD#1\_B#1\_ZQ#1.GriA}，{}.....]}

每次查询只返回第一层级，当需要查看第二层级时点击当前展示的非叶子节点，提交查询命令，返回Json格式数据，展开当前节点即可。

**三、用于诊断建模的算法原理和返回格式**

由于对单层级的数据建模查询可以参照web界面返回方式进行，但是如果数据建模需要查询多层级数据时候，算法原理就是：子树的**深度遍历**—也就是以当前节点为子树的起点，然后遍历整个子树，只收集叶子节点key。

**四、基于cpu的族谱实现的具体算法实现**

有了关系数据库的“族谱树”底层数据表以后，已比原来直接对key执行SQL操作更加高效了，此刻还是通过SQL语句来进行数据查询，解决设备分层查询的问题。如果该方法实现的效率还是让客户无法容忍，可以放弃传统SQL关系数据库工具，改用Mapd的开源GPU数据库，可以获取多GPU支持提供传统sql查询的多线程并行化加速特性，速度快于常规关系数据库进百倍（需要测试），原理是用LLVM将标准SQL语法转换为GPU多线程并行执行，后面研究的基于GPU集群的解决方案其实原理上也是相同的，利用GPU多线程优势把运算并行化提速，只是无法对接标准SQL语句，但是效率理论上会更高，因为规避了SQL语言翻译的过程。

**第二部分 基于GPU（GPU集群）的实现方案**

针对cpu树遍历算法算法可能存在的大数据处理速度问题，考虑到族谱血缘分解和树遍历算法都可以并行化实现并提高运算效率的行。我们这里也为数据族谱化的高效率实现开展了基于GPU（GPU）并行运算的算法研究。这里GPU特指Nvidia公司生产的显卡，由于其具有多达数千的thread数量，通过CUDA编程技术可以获得远超过cpu的计算能力，目前顶级8块显卡构成的集群已经相当于140台普通服务器的计算能力，起到节约空间占用、能量消耗、加快分解和查询速度的作用。GPU族谱分解的核心思路是：把算法中能并行的部分都并行加速，具体如下：

**一、基于GPU的算法原理实现**

（1）数据血缘分解：基于GPU多线程或者单线程对每个key分

原理见：第二部分的第1小结中的“**key值血缘分解”**

算法实现：详见后面算法实现部分

（2）将分解出的血缘关系按以每个key为行形成族谱矩阵

例如：MDNY.SZWQ.11MD#92\_B#1\_SK#1.GnRrrSpd 1

MDNY.SZWQ.11MD#92\_B#1\_SK#1.GeaVibXDir 2

.....

MDNY.SZWQ.11MD#1\_A#1\_YP#1.ActuAng 10000

此处假设有10000个keys，族谱矩阵Matrix的row：=10000行，而列column:=6

Column就是所有key分解出的最大家族谱长度。

例如：MDNY.SZWQ.11MD#1\_A#1\_YP#1.ActuAng 分解出家族数量：

str\_ID: int\_ID

MDNY. 11

MDNY.SZWQ. 21

MDNY.SZWQ.11MD#1 31

MDNY.SZWQ.11MD#1\_A#1 41

MDNY.SZWQ.11MD#1\_A#1\_YP#1 51

MDNY.SZWQ.11MD#1\_A#1\_YP#1.ActuAng -1

MDNY.SZWQ.11MD#1\_A#1\_YP#1.ActuAng 为6,同时它是所有其他key长度最大的，那么这批key的最大族谱长度就是6。其中int\_ID是把str\_ID放入hashtable以后进行编码转换得到的Matrix的和str\_ID双射的value值，理论上可以为一定长度内的任意整数，这里假设其为11-61的数字（实际中不一定和族谱层级呈现相同规律）。此外为每个key生成行数的算法也采用了双向hashtalbe（key2value和vlaue2key）的转换。如果该节点为叶子节点，则赋值为-1，程序如果检索到-1，就可以不用再检索该行的下一个子层级了。

那么以MDNY.SZWQ.11MD#1\_A#1\_YP#1.ActuAng分解为例，分解出的系数族谱Matrix如下，由于族谱中各key的族谱长度长短不一，所以矩阵是一个稀疏矩阵，这里采用了coo格式存储，并且采用使CUDA运算更高效的SOA—Struct Of Array数据结构进行内存存储：

Value: [11, 21, 31, 41, 51, -1]

Row: [10000,10000,10000,10000,10000,10000]

Column: [1, 2 ,3 ,4 ,5 ,6]

这只是稀疏矩阵的一部分，其余分解后组合为一个大的

Struct [cuSPARSE](https://docs.nvidia.com/cuda/cusparse/index.html" \l "abstract) Matrix{

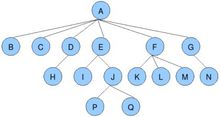
Int\* Value=: [,,,,,,,,,,,,,,,,,,,,,,,]

Int\* Row=：[,,,,,,,,,,,,,,,,,,,,,,,,,]

Int\* Column：[,,,,,,,,,,,,,,,,,,,] }

1. 族谱树和族谱矩阵矩阵的关系

“族谱矩阵”每行对应到“族谱树”其实就是“族谱树”上一棵子树，从子树顶点到一个属性叶子节点的一个遍历线路



例如：该矩阵的最大族谱深度为4（也树的最大深度为4）其中属于属性指标的是：B、C、H、P、Q、K、L、M、N。假设P点在稀疏矩阵中的行数是6，那么（行，列[族层级]，值）=[(6,4,P),(6,3,J),(6,2,E),(6,4,A)]就是一个顶点在A的子树的一个轨迹路线。

**二、基于GPU算法的具体实现**

**（一）基于多GPU集群的超线程hashtable实现**

采用动态哈希算法和多GPU集群工具，实现对key值的批量分解和建立双向ID映射等加速处理（族谱数字矩阵到key值的正反向映射，当由血缘字段生成族谱矩阵数字时叫正映射，而反之生成查询结果或诊断所需模型数据时候为反映射）。Hashtable采用的了桶方式实现动态扩展和数的多GPU分解，基本原理如下：

**1、二叉桶的分裂和数据迁移**

Hashcode：经过hash函数处理以后的整数值；

N：表示最初的桶的数量，假设最初桶构成一个满二叉树，N=2^M；

M:表示二叉树的层级；H:表示每个桶的深度（可以存储的元素个数）

Index\_tunk=Hashcode%N:表示一个key值应该放入的桶号，当一个桶的存储数据大于H深度时候桶就分裂为两个桶，同时当前桶的深度+1，分裂过程如下：

Root

1

10

01

11

0

00

GPU1

GPU0

当 0号桶的数据满了以后，0号桶分解为00和10号桶，同时里面存储的数据需进行一次分割，并分别分解到00号或者10号桶中，这是N有最初的2，扩展到了4，这时候我们可以举例看下原来桶中的数据发生的变化：假设有两个key计算的hash码为Hashcode1=4,Hashcode2=6，由于两者除2的余数都等于0所以都在0号桶中，但是当0号桶的数据溢出后扩展为了00和10号桶，此时两者除4的余数变为0和2，此时Hashcode1保留在了00号桶，而Hashcode2分裂到了10号桶。对于1号桶，由于数据没有溢出，01号桶和11号桶均为虚数桶，一般文献资料中处理方法是将两者地址链接到1号桶，但是此种方法不太适合CUDA少移动数据的优化处理原则，本文采取的方法是“后部扩展法”，也就是每轮扩展后增加一倍数量的桶其实都扩展在原桶列的后面，对于桶1其实和01是一个桶换了名字而不是出现了数据转移或者增加虚桶，在计算机中实际桶实际存储的顺序如下：

01

11

0

1

10

00

变名字

00桶就是0桶，01桶就是1桶，内存中只是增加了10和11桶，也就是上图所展示的多层级6桶结构只是变化的过程描述而不是实际分配，内存中只由2个桶增加到了4个。

**2、桶预分裂和多GPU并行**

在key不断分解到hashtable的过程中，由于单个桶容量的问题会发生桶的分裂和数据的迁移，而CUDA计算的瓶颈就是过多的数据拷贝，所以为了避免这种情况发生，我们这里依赖GPU集群的强大运算能力，采用了桶预分裂技术。算法原理是在把输入数据集分解为device\_num份（GPU个数）,在每个GPU上分别模拟统计每个桶会存入的key-value数据量，并通过NCCL（是Nvidia基于c语言实现的多GPU通信工具）的all-reduce算子，实现各桶的结果叠加，得到需要分裂的桶或者是否需要扩充桶层级，所谓扩充桶层级也就是当某个桶的LD局部深度由于桶分裂大于了整个二叉树的GD全局深度时候，桶列长度就要增加一倍了，此时由于虚桶的存在，实际存储空间其实并未扩展到原来2倍。

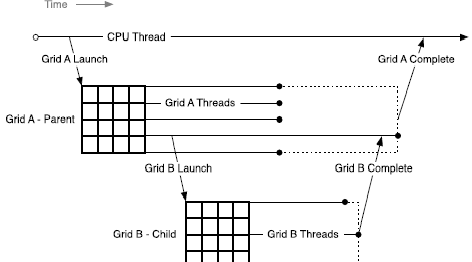
1. **并法的数据插入和多GPU并行**

在预计处理阶段除了会进行桶预计分裂以外，还会记录每条记录进入的桶号，而每个桶在进行分裂模拟时候就已经分配了所属的GPU，基本原则是被分裂的母桶属于哪个GPU由其分裂出的子桶就属于哪个GPU。随后在用NCCL的Bocast广播算子把需要计算的Key数据集拷贝device\_num份到每个GPU上，在正式插入中GPU火力全开只批量处理在预处理阶段已经认定会被分入自己所管辖GPU的哪部分key条目，最后计算全部完成后，再通过NCCL的all\_Gather算子把每个GPU上的不同桶聚合到一起，插入结束。

**（二）基于多GPU集群和“族谱矩阵”的多级线程并发查询**

在进行web页面查询和数据建模指标查询的时候，实际进行的是对“族谱树”子树的的纵向深度遍历或横向广度编历，当“数据源点数量较大，到顶路径很多”情况下，如果只是采用cpu多线程和甚至是单线程少量节点并发势必影响输出效率，这里采用了CUDA动态并行和多GPU集群来实现远快于关系数据库的数据聚合，具体算法原理就流程如下：

## [CUDA Dynamic Parallelism](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html" \l "cuda-dynamic-parallelism)



**注：动态并行技术就是在每个GPU上已经运行的线程内部在调用一次gloable函数开启下一层的线程并行，总共最高可以迭代21层。**

**#1）算法条件。**目前我们测试新编码的层级最多为6层族谱，远低于21层的峰值，就新能源设备的层级看，短期内不会超过10层，所以动态并行算子在相当长时期内对数据族谱解析是有效的。考虑到生成的复杂JSON格式数据解析效率很低，一般也就只开启直接子类检索，所以本模型实现中一般开启2层动态并行即可。

**#2）动态并行检索查询所有直接子孙。假设用户输入**MDNY.SZWQ.11MD对北京四子王旗下所有风机的属性代码和子设备进行查询和检索并归类到具体特定风机下面。分解确定MDNY.SZWQ.11MD属于第3层级父族，此时多GPU集群把稀疏族谱矩阵数据分段放入各GPU中开启n1个第一层并行threads检索所有Column=3的列，如果找到了查看该（row\_index,3,value）中的value是否等于MDNY.SZWQ.11MD，如果是则开启第二层动态并行的n2个threads在第4层级中检索和上一层级row\_index相等的行，检索其value是否为-1，如果是则表明该节点是叶子属性节点，加入输出list；如果value不等于-1也不等于-2，表明该节点是非叶子节点，叶加入输出list；不会存在有-2情况。将找到的value放入双向hashtable存储中逆向检索出所属key值。

#3）**并行计算检索所有深度的叶子节点**

假设用户还是MDNY.SZWQ.11MD对北京四子王旗下所有叶子节点进行检索并归类到具体特定风机下。只需要先定位被检索字段属所在的层级3，开启多GPU并行导入系数族谱矩阵，检索在Column=3这个层级中，（row\_index,3,value）中的value（剔除序列号）是否等于MDNY.SZWQ.11MD，如果是返回所有row\_index，并在放入双向hashtable存储内存逆向检索出所属的key值。

#4）获取结果后对于需要web展示的将结果key放入JSON格式进行javascrip解析，对于用来诊断建模的按所属打设备直接输出list即可。