



知识图谱数据管理系统

—— 湖南大学信息科学与工程学院 ——



目 录

- 第一节 知识图谱数据管理背景**

- 第二节 知识图谱数据管理方法**



知识图谱数据模型

资源描述框架

(Resource Description Framework)

- 它是一种被广泛用于知识图谱的数据模型
- 所有东西都是唯一命名的资源
- 可以定义资源的属性
- 可以定义与其他资源的关系

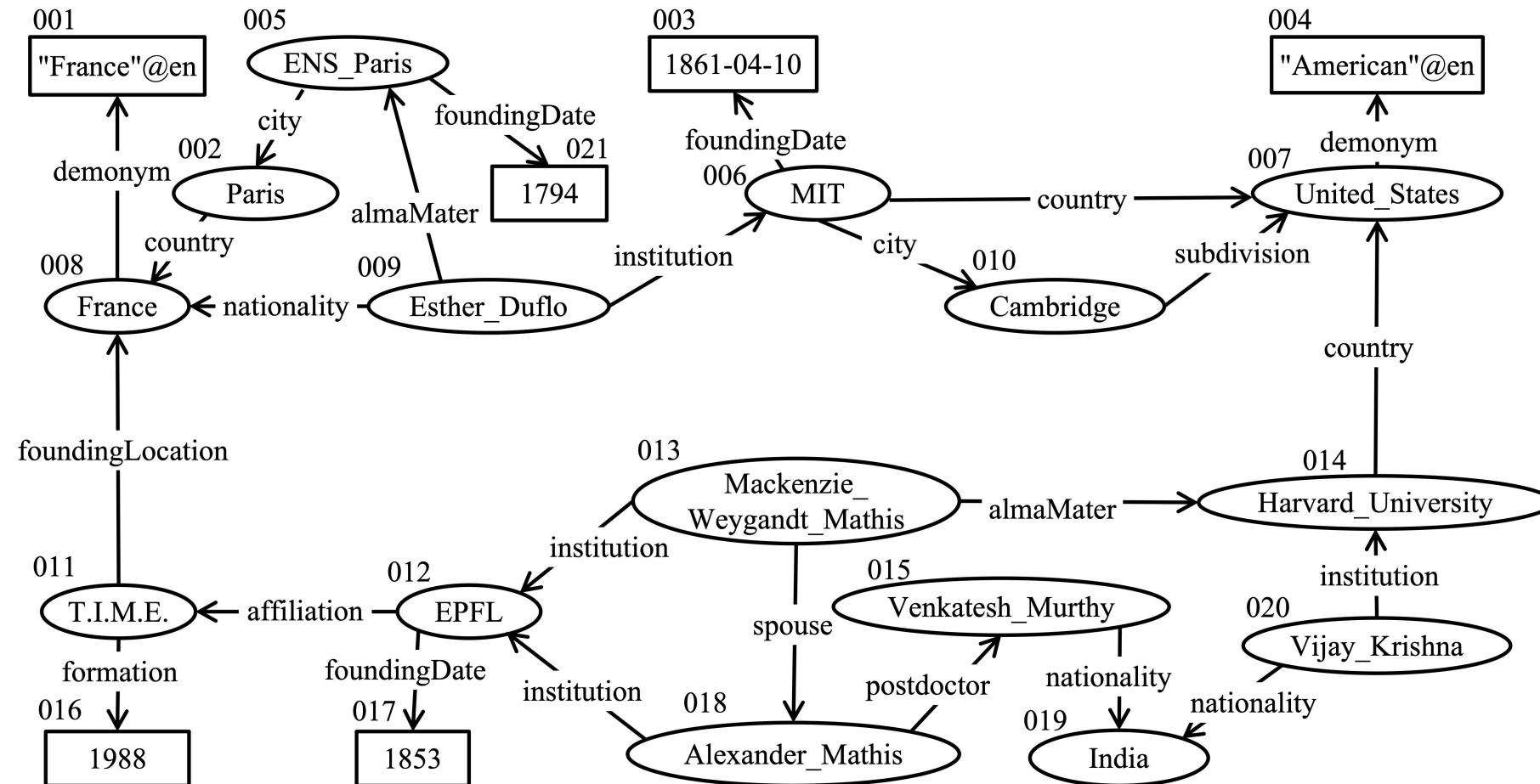
属性图

(Property Graph)

- Neo4j等图数据库常用模型
- 由点来表示现实世界中的实体，由边来表示实体与实体之间的关系
- 点和边上都可以通过键值对的形式被关联上任意数量的属性和属性值

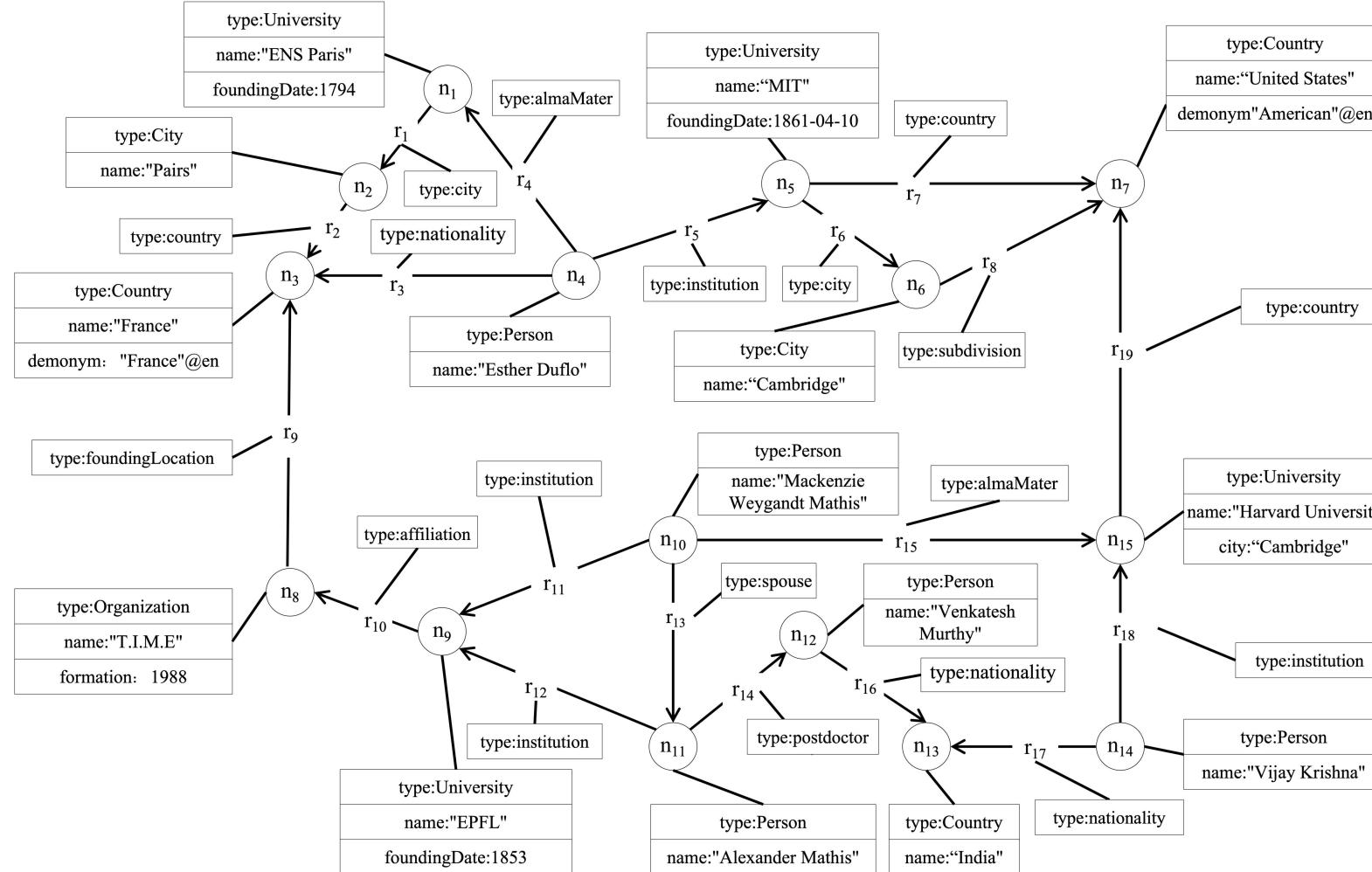


RDF模型





属性图模型





知识图谱数据查询

SPARQL

- W3C 制定的RDF 数据的标准查询语言

```
Select ?x1 ?y1 ?z1 where {  
?x1 institution ?y1 .  
?x1 nationality ?z1 .  
?y1 foundingDate ?d1 . }
```

Cypher

- Cypher 是Neo4j 系统所支持的面向 Neo4j所存储属性图的查询语言

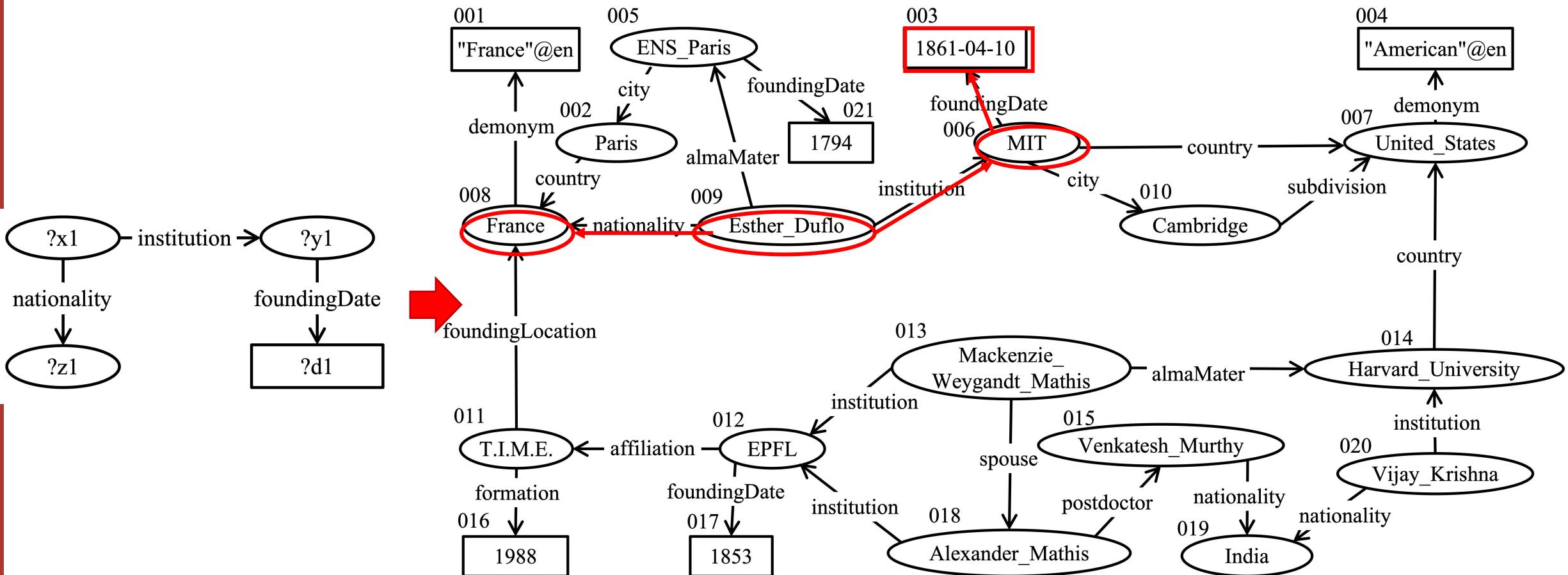
```
MATCH (x1)-[:institution]->(y1  
{foundingDate:d1})  
MATCH (x1)-[:nationality]->(z1)
```

回答图模式查询=图数据对查询图进行子图匹配



知识图谱查询处理

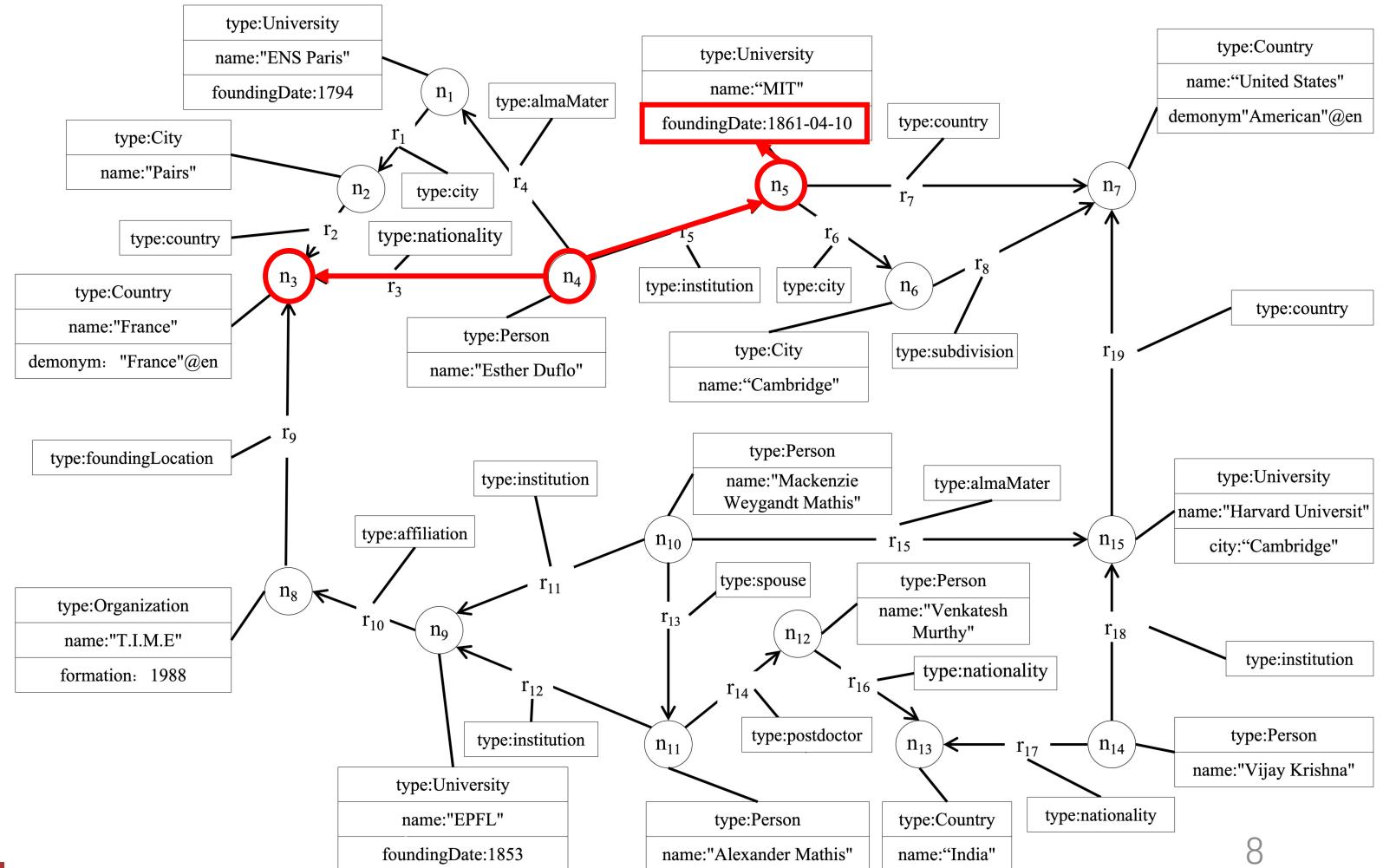
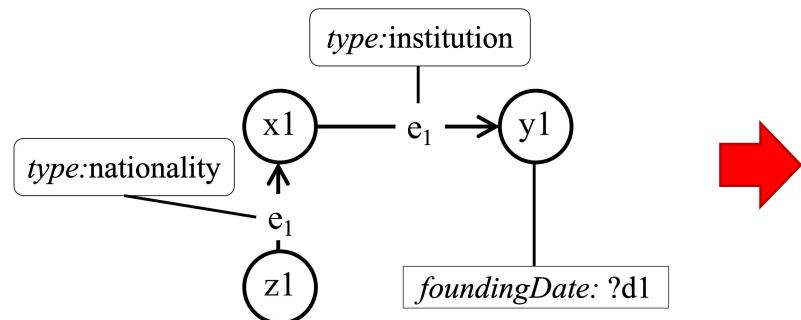
RDF图数据上SPARQL查询处理



知识图谱查询处理



属性图数据上Cypher查询处理



知识图谱数据管理基本操作



- 类似于关系数据，知识图谱上的基本操作也有以下几个
 - 选择
 - 连接



知识图谱数据管理基本操作

- **选择**在知识图谱中选择满足给定条件的知识图谱片段

主体	谓词	客体
苏格拉底	学生	柏拉图
苏格拉底	出生时间	公元前469年
苏格拉底	英文译名	Aristotle
苏格拉底	出生地	雅典
苏格拉底	isA	哲学家
柏拉图	代表作品	《理想国》
柏拉图	出生时间	公元前427年
柏拉图	英文译名	Plato
柏拉图	出生地	雅典
柏拉图	isA	唯心主义哲学家
《理想国》	地位	最具影响力20本学术书
《理想国》	isA	书籍
雅典	外文名称	Athens
雅典	isA	城市
唯心主义哲学家	subclassOf	哲学家

→

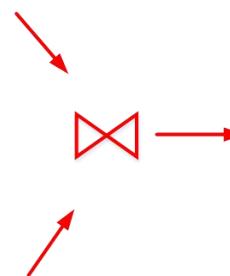
主体	谓词	客体
苏格拉底	出生地	雅典
柏拉图	出生地	雅典



知识图谱数据管理基本操作

- **连接**是按照一定条件从两个知识图谱的笛卡儿积中选取知识图谱片段

主体	谓词	客体
苏格拉底	出生地	雅典
柏拉图	出生地	雅典



主体1	谓词1	客体1.主体2	谓词2	客体2
苏格拉底	学生	柏拉图	出生地	雅典

主体	谓词	客体
苏格拉底	学生	柏拉图

知识图谱数据管理基本操作



- 常见的应用中知识图谱数据操作的特点如下：
 1. 选择：选择度高
 2. 连接：连接数量多



目 录

- 第一节 知识图谱数据管理背景**

- 第二节 知识图谱数据管理方法**



知识图谱数据关系表存储

- 现有方法：利用关系数据库技术定义一张三列的表

主体	属性	客体
亚里士多德	导师	柏拉图
亚里士多德	出生时间	公元前384年
亚里士多德	英文译名	Aristotle
亚里士多德	出生地	雅典
亚里士多德	isA	哲学家
柏拉图	代表作品	《理想国》
柏拉图	出生时间	公元前427年
柏拉图	英文译名	Plato
柏拉图	出生地	斯塔基拉
柏拉图	isA	唯心主义哲学家
《理想国》	地位	最具影响力的20本学术书
《理想国》	isA	书籍
雅典	外文名称	Athens
雅典	isA	斯塔基拉
唯心主义哲学家	subclassOf	哲学家

Virtuoso



- Virtuoso是OpenLink公司开发的知识图谱管理系统，有免费的社区版
- 一个数据集只有一张大表，加上若干索引
- 下载地址：<https://virtuoso.openlinksw.com/>



Virtuoso表结构

主体	属性	客体
亚里士多德	导师	柏拉图
亚里士多德	出生时间	公元前384年
亚里士多德	英文译名	Aristotle
亚里士多德	出生地	雅典
亚里士多德	isA	哲学家
柏拉图	代表作品	《理想国》
柏拉图	出生时间	公元前427年
柏拉图	英文译名	Plato
柏拉图	出生地	斯塔基拉
柏拉图	isA	唯心主义哲学家
《理想国》	地位	最具影响力的20本学术书
《理想国》	isA	书籍
雅典	外文名称	Athens
雅典	isA	斯塔基拉
唯心主义哲学家	subclassOf	哲学家

主键
12字符以内
的直接存值；
12字符以上
的对应到一个
整数ID

另外，分别建立以主语和宾语为引导列的多列索引
2024-3-21



三列表优缺点

- 优点：基于现有的关系数据库，方法成熟
- 缺点：选择需要扫描整个数据集，效率低下；自连接多，连接操作效率低

三种典型基于关系数据库的优化策略



- 属性表方法 Jena
- 垂直划分方法 SW-Store
- 全索引方法 Hexastore、RDF-3x



属性表

- 属性表方法的核心理念是将 RDF 图数据中每个实体与其所有相关谓词和客体存储为数据库中的一行。这样对于星型查询而言，该方法可以避免处理“主体-主体”的连接操作代价。



属性表

- 属性表方法可以分为两类：
 - 一类是通过将具有相似的属性集合的实体聚类在一起，形成聚类属性表；
 - 另外一类是按照实体的类别信息 (`rdf:type`)，具有相同类别信息的实体放在同一个分类属性表中
- 对于上述两种情况，由于 RDF 数据表示的灵活性，会存在部分三元组无法放入任何一个属性表的情况，这时候剩下来的三元组被放入溢出表 (overflow table) 中。



聚类属性表示例

T1

主体	rdf:type	rdfs:label	dbo:birthPlace	dbo:birthDate	dbo:institution	dbo:almaMater	dbo:award	dbo:award
dbr:Charles_Darwin	dbr:Scientist	"Charles Darwin"@en	dbr:Shrewsbury	"1809-02-12"	dbr:University_of_Cambridge	null	dbr:Royal_Medal	dbr:Baly_Medal
dbr:Albert_Howard	dbr:Scientist	"Albert Howard"@en	null	null		dbr:University_of_Cambridge	null	null

T2

主体	rdf:type	rdfs:label	dbp:location	dbo:country
dbr:University_of_Cambridge	dbr:University	"University of Cambridge"@en	Cambridge,England	dbr:England

T3

主体	rdf:type	rdfs:label	dbo:demonym	dbo:subdivision
dbr:Shrewsbury	dbr:Town	"Shrewsbury"@en	"Salopian"@en	dbr:Shropshire
dbr:Shropshire	dbr:District	"Shropshire"@en	null	dbr:England
dbr:England	dbr:Country	"England"@en	"English people"@en	null

SPARQL

```
SELECT ?name
WHERE {
?m dbo:birthPlace ?city.
?m rdfs:label ?name.
?m dbo:birthDate ?bd.
?city dbo:demonym "Salopian"@en .
FILTER (regex (str (?bd), "1809"))
}
```



SQL

```
SELECT T1.rdfs:label
FROM T1, T3
WHERE T1 dbo:birthPlace=T3.subject
AND T3 dbo:demonym="Salopian"@en
AND T1 dbo:birthDate LIKE '%1809%'
```



聚类属性表示例

- 聚类属性表示例中， dbr:Shrewsbury、dbr:Shropshire、dbr:England 三个地点实体聚类在了一起，因为它们拥有相似的属性/谓词集合



分类属性表示例

T1

dbr:Scientist

主体	rdfs:label	dbo:birthPlace	dbo:birthDate	dbo:institution	dbo:almaMater	dbo:award	dbo:award
dbr:Charles_Darwin	"Charles Darwin"@en	dbr:Shrewsbury	"1809-02-12"	dbr:University_of_Cambridge	null	dbr:Royal_Medal	dbr:Baly_Medal
dbr:Albert_Howard	"Albert Howard"@en	null	null	dbr:University_of_Cambridge	null	null	

T2

dbo:University

主体	rdfs:label	dbp:location	dbo:country
dbr:University_of_Cambridge	"University of Cambridge"@en	Cambridge,England	dbr:England

T3

T3

dbr:Town

主体	rdfs:label	dbo:demonym	dbo:subdivision
dbr:Shrewsbury	"Shrewsbury"@en	"Salopian"@en	dbr:Shropshire

T4

dbr:District

主体	rdfs:label	dbo:subdivision
dbr:Shropshire	"Shropshire"@en	dbr:England

SPARQL

```
SELECT ?name
WHERE {
?m dbo:birthPlace ?city. ?m rdfs:label ?name.
?m dbo:birthDate ?bd.
?city dbo:demonym "Salopian"@en .
FILTER (regex (str (?bd), "1809"))
}
```

SQL

```
SELECT T1.rdfs:label
FROM T1, T3
WHERE T1 dbo:birthPlace=T3.subject
AND T3 dbo:demonym="Salopian"@en
AND T1 dbo:birthDate LIKE '%1809%'
```



分类属性表示例

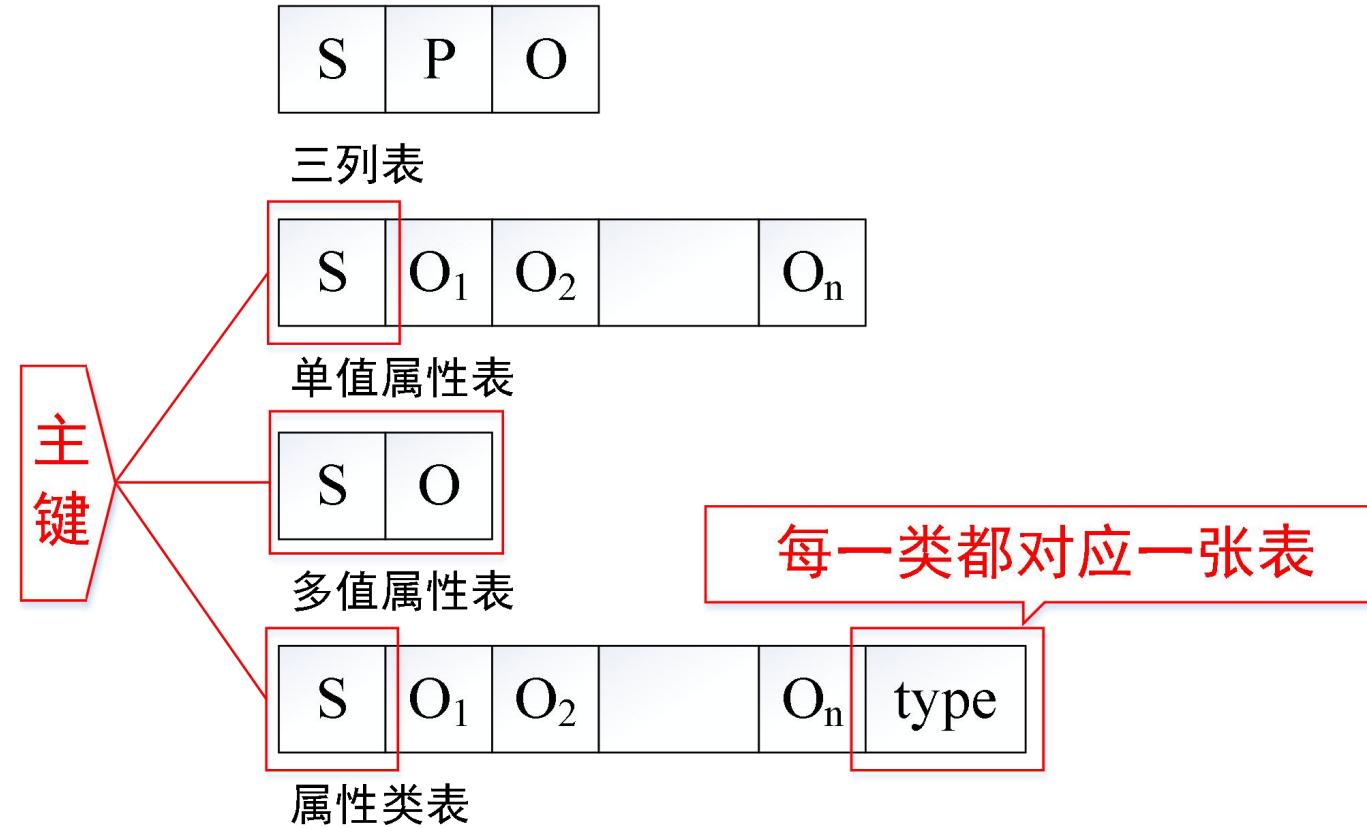
- 分类属性表中，由于dbr:Charles_Darwin和 dbr:Albert_Howard 的类型相同，都是“dbr:Scientist”，因此它们被放在同一张类属性表中；其他实体由于属于不同的类别，因此各自放在不同的表中
- 上述两个例子中，转换的 SQL 中明显减少了连接操作的次数。



- Jena是惠普实验室开发的知识图谱管理系统，现已由Apache管理
- Jena在三元组表之外，维护了三种属性表：单值属性表，多值属性表和属性类表
- 下载地址：<http://jena.apache.org/index.html>



Jena属性表





属性表

- 虽然属性表可以减少连接操作的次数，但是属性表有两个缺点难以克服
 1. 表中的空值 (null) 数目多；
 2. 属性多值问题；



DB2RDF——正向主哈希表

为了克服属性表中上述问题，IBM 在所提出的 DB2RDF 系统提出了一种新的属性表组织模式

DB2RDF 将所有的实体都放在一张属性表 T 中。这张属性表 T 称为正向主哈希表 (Direct Primary Hash, 简称为 DPH)

简而言之，正向主哈希表是一个宽表，其中每一行对应一个实体 s ，将其存储在正向主哈希表的 entry 列中。正向主哈希表每行有 $2 \times k + 2$ 列，分别为 entry 列、spill 列和 k 对 (pred_i 和 val_i) 列， $(1 \leq i \leq k)$ 。



DB2RDF——正向主哈希表

同时，我们定义一个从全体谓词集合 (P) 到 $(1, \dots, k)$ 的哈希函数 H 。假设实体 s 作为主体时，其关联的谓词集合是 $\text{pred}(s)$ 。对谓词集合 $\text{pred}(s)$ 中的每个谓词 p (即 $p \in \text{pred}(s)$)，我们根据哈希值 $H(p)$ 确定 s 的谓词 p 和对应的属性值放在 DPH 表的第 $H(p)$ 对的 $\text{predH}(P)$ 和 $\text{valH}(P)$ 列中。



DB2RDF——正向主哈希表

- 如果发生哈希冲突， s 对应谓词和属性值可能需要放在下面一行；为了表示同一个实体的谓词集合可能分配在多行中，其对应的 spill 列设置为 1
- 正向主哈希表的每个谓词列(pred_1)对应多个谓词，而不是单一谓词



DB2RDF——正向主哈希表示例

entry	spill	pred ₁	val ₁	pred ₂	val ₂	pred ₃	val ₃	pred ₄	val ₄
dbr:Gregor_Mendel	1	rdf:type	dbr:Scientist	rdfs:label	"Gregor Mendel"@en	dbo:deathPlace	dbr:Brno	dbo:birthDate	"1822-07-20"
dbr:Gregor_Mendel	1	null	null	dbo:almaMat	dbr:University_of_Vienna	null	null	null	null
dbr:University_of_Vienna	0	rdf:type	dbo:University	rdfs:label	"University of Vienna"@en	dbo:country	dbr:Austria	null	null
dbr:Friedrich_Hayek	0	rdf:type	dbo:Economist	rdfs:label	"Friedrich Hayek"@en	dbo:almaMater	University_of_Vienna	null	null
dbr:Brno	0	rdf:type	dbr:City	rdfs:label	"Brno"@en	dbo:subdivision	dbr:South_Moravian_Region	null	null
dbr:South_Moravian_Region	0	rdf:type	dbr:Region	rdfs:label	"South Moravian Region"@en	dbo:subdivision	dbr:Czech_Republic	null	null
dbr:Czech_Republic	0	rdf:type	dbr:Country	rdfs:label	"Czech Republic"@en	null	null	null	null
dbr:City	0	rdfs:subClass	dbr:Place	null	null	null	null	null	null
dbr:Region	0	rdfs:subClass	dbr:Place	null	null	null	null	null	null
dbr:Country	0	rdfs:subClass	dbr:Place	null	null	null	null	null	null



DB2RDF——正向二级哈希表

- 此外，对于同一谓词属性值的多值问题，DB2RDF 也进行了特殊处理
- DB2RDF 将相关的多值属性存储在另一张表中，称为正向二级哈希表（Direct Secondary Hash）。在将多值谓词存储在正向主哈希表中时，DB2 会为谓词对应属性值整体打包，并分配一个新的唯一标识符
- 该标识符存储在正向二级哈希表中，并与每个谓词对应的具体属性值关联



DB2RDF——正向二级哈希表

实体 dbr:Charles_Darwin 作为主体的时候通过 dbo:award 谓词连接了两个属性值，分别是 dbr:Royal_Medal 和 dbr:Baly_Medal。

因此，正向主哈希表中 dbo:award 谓词对应的 val₃ 列值是 lid:1；而 lid:1 在下图所示的正向二级哈希表中对应了两个客体，即 dbr:Royal_Medal 和 dbr:Baly_Medal。

<u><i>l_id</i></u>	<u><i>elm</i></u>
lid:1	dbr:Royal_Medal
lid:1	dbr:Baly_Medal



DB2RDF——反向主哈希表和反向二级哈希表

从 RDF 图的角度来看，正向主哈希表和正向二级哈希表本质上编码了实体的出边（即主体的谓词）。为了实现高效访问，还有必要编码实体的入边（指向客体的谓词）。为此，DB2RDF 还提供了两个额外的表，称为反向主哈希表（Reverse Primary Hash）和反向二级哈希表（Reverse Secondary Hash）。



DB2RDF——反向主哈希表示例

entry	spill	pred ₁	val ₁	pred ₂	val ₂	pred ₃	val ₃	pred ₄	val ₄
dbr:Scientist	0	rdf:type	lid:2	null	null	null	null	null	null
"Charles Darwin"@en	0	null	null	rdfs:label	dbr:Charles_Darwin	null	null	null	null
dbr:Shrewsbury	0	null	null	null	null	dbo:birthPlace	dbr:Charles_Darwin	null	null
"1809-02-12"	0	null	null	null	null	null	null	dbo:birthDate	dbr:Charles_Darwin
dbr:University_of_Cambridge	0	null	null	dbo:institution	dbr:Charles_Darwin	dbo:almaMater	dbr:Albert_Howard	null	null
dbr:Royal_Medal	0	null	null	null	null	dbo:award	dbr:Charles_Darwin	null	null
dbr:Baly_Medal	0	null	null	null	null	dbo:award	dbr:Charles_Darwin	null	null
"Albert Howard"@en	0	null	null	rdfs:label	dbr:Albert_Howard	null	null	null	null
dbo:University	0	rdf:type	dbr:University_of_Cambridg	null	null	null	null	null	null
"University of Cambridge"@en	0	null	null	rdfs:label	dbr:University_of_Cambrid	null	null	null	null
Cambridge,England	0	null	null	null	null	dbp:location	dbr:University_of_Cambridg	null	null
dbr:England	0	null	null	null	null	null	null	dbo:country	dbr:University_of_Cambrid
dbr:Town	0	rdf:type	dbr:Shrewsbury	null	null	null	null	null	null
"Shrewsbury"@en	0	null	null	rdfs:label	dbr:Shrewsbury	null	null	null	null
"Salopian"@en	0	null	null	null	null	null	null	dbo:demonym	dbr:Shrewsbury
dbr:Shropshire	0	null	null	null	null	dbo:subdivision	dbr:Shrewsbury	null	null
dbr:District	0	rdf:type	dbr:Shropshire	null	null	null	null	null	null
"Shropshire"@en	0	null	null	rdfs:label	dbr:Shropshire	null	null	null	null
dbr:England	0	null	null	null	null	dbo:subdivision	dbr:Shropshire	null	null
dbr:Country	0	rdf:type	dbr:England	null	null	null	null	null	null
"England"@en	0	null	null	rdfs:label	dbr:England	null	null	null	null
"English people"@en	0	null	null	null	null	null	null	dbo:demonym	dbr:England
dbr:Place	0	rdfs:subClass	lid:3	null	null	null	null	null	null



DB2RDF——反向二级哈希表示例

<i>l_id</i>	<i>elm</i>
lid:2	dbr:Town
lid:2	dbr:District
lid:2	dbr:Country



DB2RDF上SPARQL查询处理

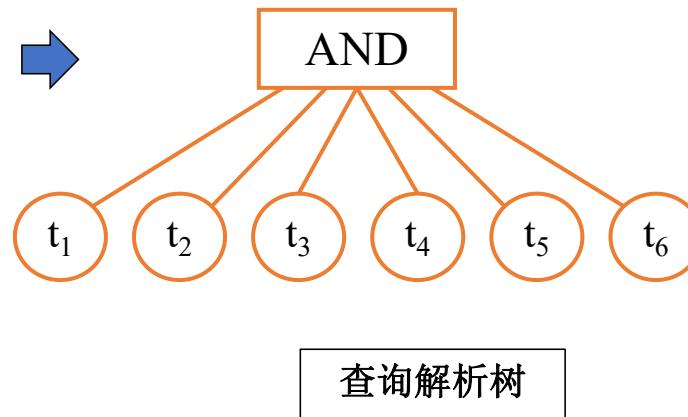
将 SPARQL 查询首先转化一个数据流图（Data Flow Graph），然后根据这个数据流图转化成查询执行树。最后基于系统预定义的 SQL 语句模板将查询执行树重写出 SQL 语句来执行。下图给出了一个 DB2RDF 查询处理流程示例。



DB2RDF上SPARQL查询处理

SPAQRL 查询会首先被转化生成一个查询解析树。这一步现有 SPARQL 查询解析器就能完成

```
Select * Where {  
    t1 ?x rdfs:label "Charles  
    Darwin"@en .  
    t2 ?x dbo:institution ?y .  
    t3 ?y dbo:country dbr:England .  
    t4 ?z dbo:almaMater ?y .  
    t5 ?y rdfs:label ?n .  
    t6 ?y dbp:location ?m .  
}
```





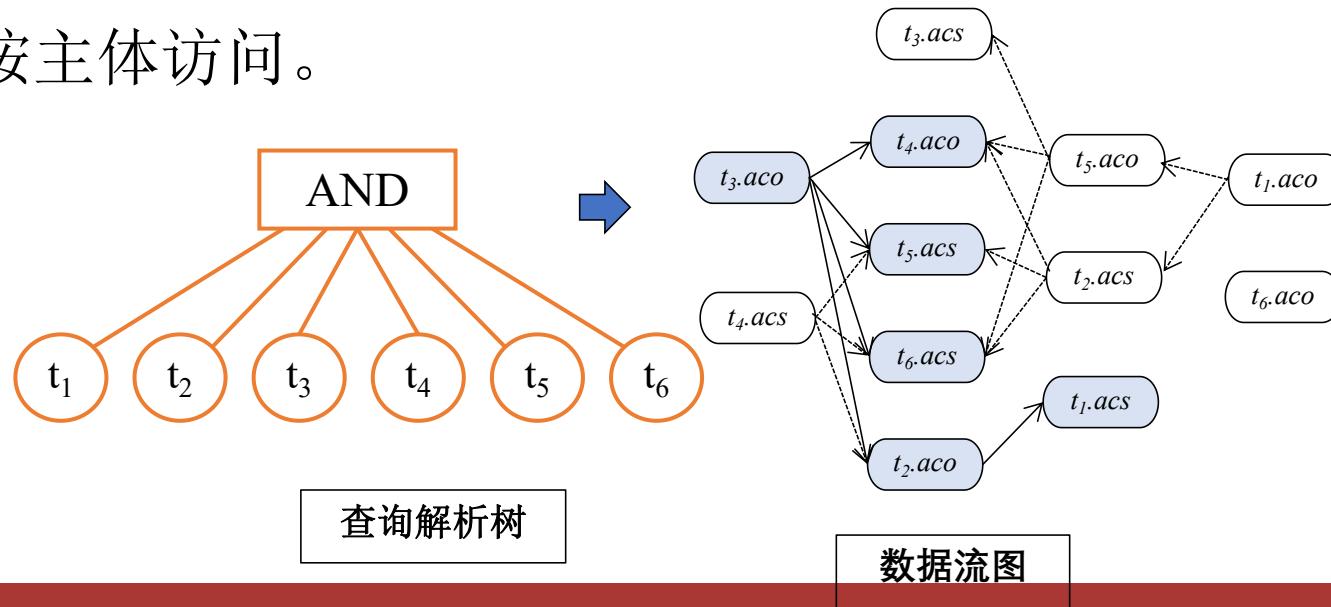
DB2RDF上SPARQL查询处理

然后，基于这个查询解析树，DB2RDF 生成一个数据流图。所谓数据流图，它用来表示每个三元模式执行之间的依赖关系。此图中的一个点是一个三元模式和一个访问方法的组合；一条边表示一个三元模式生成一个另一个三元模式需要的共享变量。对于数据流图中的访问方法，对于像 DB2RDF 这样只有主体和客体索引（没有谓词索引）的系统，可以使用的方法包括按主体访问（acs）、按客体访问（aco）或全扫描（sc）。



DB2RDF上SPARQL查询处理

DB2RDF 查询处理流程示例图中包含一个数据流图示例。这里，对于 t_3 这个三元组模式，数据流图中其实包含两个点 $t_3.acs$ 和 $t_3.aco$ ，表示 t_3 这个三元组模式可以按主体访问和按客体访问的到结果。其中， $t_3.aco$ 这个点有一条边指向 $t_5.acs$ ，因为 t_3 按客体访问得到主体之后得到 $?y$ 结果， $?y$ 的结果可以用于 t_5 的按主体访问。



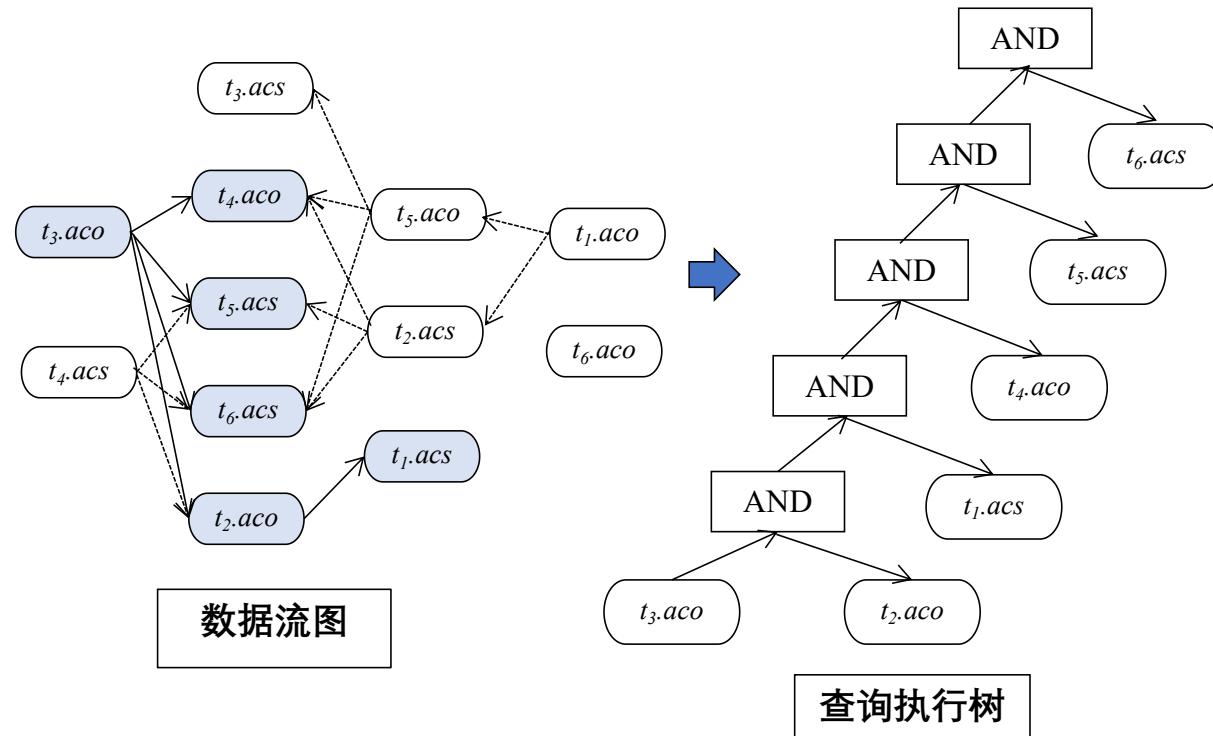


DB2RDF上SPARQL查询处理

DB2RDF 基于数据库中的统计信息计算出最优流树（假设为示例图中的蓝色点）。该树确定了遍历查询中所有三元模式的最佳方式，以最小化成本为标准。然后，DB2RDF依据这个最优流树生成查询执行树。DB2RDF 计算查询执行树的算法是一个递归算法。其输入为由 DB2RDF 计算得到的最优流树 F 以及查询解析树 P，初始情况下 P 包含整个查询。然后，算法在 P 中自顶向下按照点类型递归地分拆 P，并根据 F 确定查询访问方法。



DB2RDF上SPARQL查询处理





DB2RDF上SPARQL查询处理

最终，DB2RDF 依据所得到的查询执行树，利用下图所示的 SQL 模板，翻译出SQL 查询，并去相应的关系表中的得到最终结果。查询执行树在这个过程中起着重要作用，其中的每个点，无论是三元组、合并还是控制点，都包含了指导 SQL 生成所需信息的元素。对于 SQL 生成，SQL 构建器对查询执行树进行后序遍历，并为每个点生成相应的 SQL 查询。整个过程通过使用 SQL 代码模板来辅助完成。



DB2RDF的SQL模板

```
SELECT ... 框1  
FROM DPH AS T    框2  
WHERE T.ENTITY = ... AND           框3  
      T.PREDn1 = ... AND T.PREDn2 = ...  
LEFT OUTER JOIN DS AS S0  
ON T.VALn1 = S0.lid            框4
```



DB2RDF上SQL转化

更详细地说，SQL翻译的基本情况考虑了一个对应于单个三元组或合并的点。框 1 中的代码将查询的目标设置为 DPH 或 RPH 表，具体取决于三元组点中的访问方法。框 2 中的代码限制了正在查询的实体。例如，当主语是常量且访问方法是 acs 时，表中 entry 列与常量主语值相连接。当主语是变量且方法是 acs 时，然后表中 entry 列与先前在前一个 SELECT 子查询中绑定的变量相连接。



DB2RDF上SQL转化

当对象的访问方法是 aco 时，对于 entry 列也是同样的操作。框 3 说明了如何选择一个或多个谓词。也就是说，当计划点对应于一个合并时，多个谓词列通过合取或析取的 SQL 操作符相连接。最后，框 4 显示了如何与多值谓词的辅助表进行外连接。于是，对于 DB2RDF 查询处理流程示例图所示查询生成的 SQL 查询如下图所示。



DB2RDF上SQL转化

```
WITH QT3RPH AS
    SELECT T.val1 AS val1 FROM RPH AS T WHERE T.entry ='dbr:England' AND T.pred1 =' dbo:country',
QT3DS AS
    SELECT COALESCE (S.elm, T.val1) AS y
    FROM QT3RPH AS T LEFT OUTER JOIN DS AS S ON T.val1 =S.l_id
QT2RPH AS
    SELECT T.entry AS y, QT3DPH.x FROM RPH AS T, QT3DPH
    WHERE T.entry =QT3DPH.y AND T.pred1 ='dbo:institution',
QT2 AS
    SELECT LT.val0 AS x, T.y FROM QT2RPH as T, TABLE (T.valm, T.val0) as LT(val0)
    WHERE LT.val0 IS NOT NULL
QT1DPH AS
    SELECT T.entry AS x, QT2.y FROM DPH AS T, QT2
    WHERE T.entry =QT2.x AND T.predk ='rdfs:label' AND T.val1 ='"Charles Darwin"@en',
QT4RPH AS
    SELECT T.entry AS y, QT1DPH.x FROM RPH AS T, QT1DPH
    WHERE T.entry =QT1DPH.y AND T.pred1 ='dbo:almaMater',
QT5DPH AS
    SELECT T.entry AS y, QT4.y FROM DPH AS T, QT4
    WHERE T.entry =QT4.y AND T.predk ='rdfs:label',
QT6DPH AS
    SELECT T.entry AS y, QT4.y FROM DPH AS T, QT5
    WHERE T.entry =QT5.y AND T.predk ='dbp:location',
    SELECT x, y, z FROM QT6DPH
```



垂直划分方法

- 对RDF知识图谱数据按照属性分割成若干表的方法

主体	属性	客体
亚里士多德	导师	柏拉图
亚里士多德	出生时间	公元前384年
亚里士多德	英文译名	Aristotle
亚里士多德	出生地	雅典
亚里士多德	isA	哲学家
柏拉图	代表作品	《理想国》
柏拉图	出生时间	公元前427年
柏拉图	英文译名	Plato
柏拉图	出生地	斯塔基拉
柏拉图	isA	唯心主义哲学家
《理想国》	地位	最具影响力的20本学术书
《理想国》	isA	书籍
雅典	外文名称	Athens
雅典	isA	斯塔基拉
唯心主义哲学家	subclassOf	哲学家





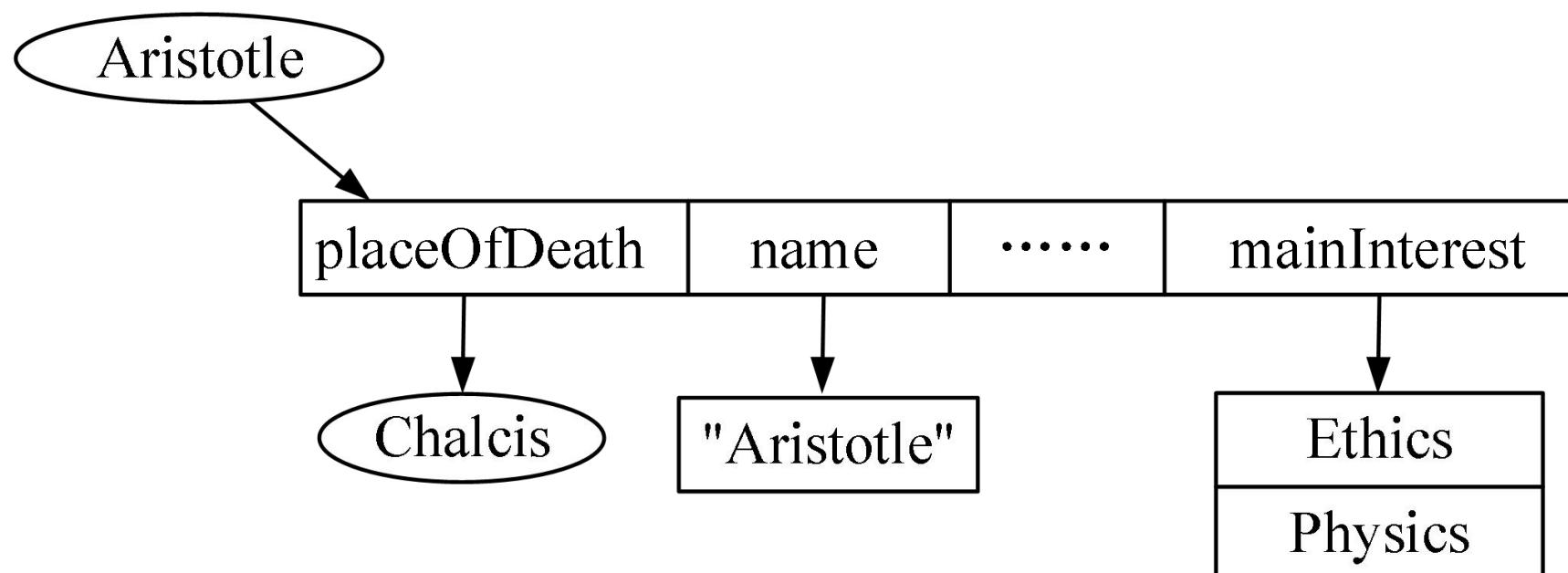
垂直划分方法优缺点

- 优点：避免了大量自连接；存储时无空值
- 缺点：属性为变量的时候访问效率低下



全索引方法

- 将三元组在主体、属性、客体之间各种排列下能形成各种形态构建都枚举出来，然后为它们构建索引





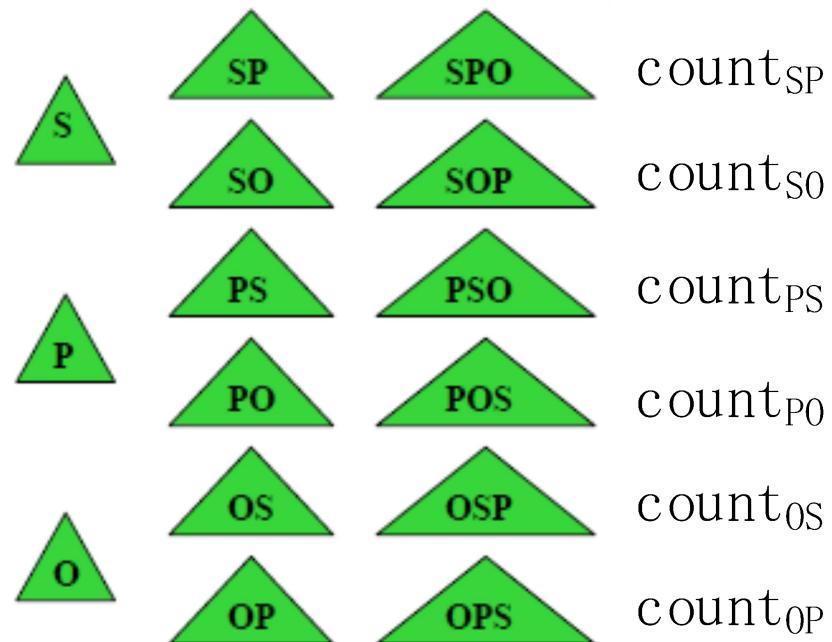
RDF-3X

- RDF-3X是德国马克斯普朗克实验室开发的知识图谱管理系统，主要特点就是建立全索引
- 下载地址：<https://code.google.com/p/rdf3x/>



RDF-3X索引

- 在全索引之外，RDF-3X还有建立了针对任意两列组合出的值对的聚集索引来记录统计信息





全索引优缺点

- 优点：选择操作效率极高
- 缺点：连接操作依然低效；索引更新代价高

原生RDF图数据库系统——基于邻接表的方法



- 通过将RDF三元组看作带标签的边，RDF知识图谱数据自然的符合图模型结构，下图展示了一个知识图谱与其对应的邻接表

点	邻接表
柏拉图	(isA, 唯心主义哲学家), (出生时间, 公元前427年), (出生地, 雅典), (英文译名, Plato), (代表作品, 《理想国》)
苏格拉底	(isA, 哲学家), (出生时间, 公元前469年), (出生地, 雅典), (英文译名, Socrates), (学生, 柏拉图)
.....



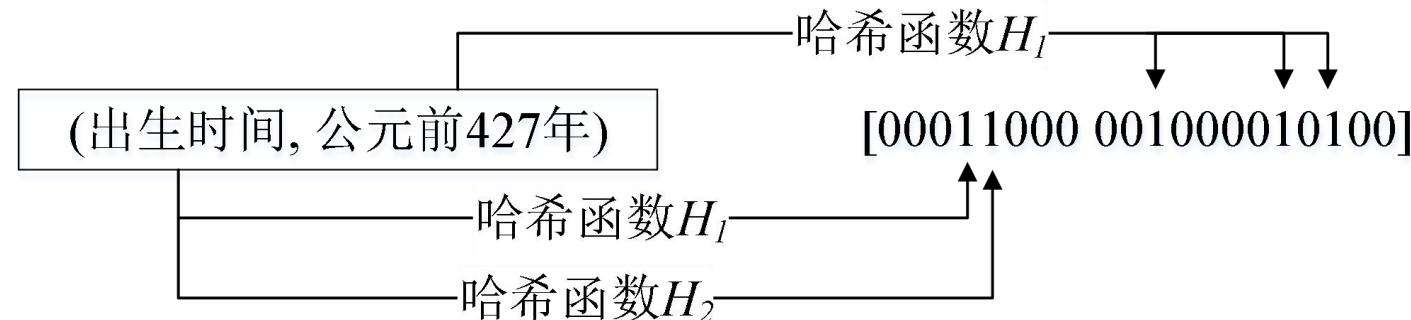
原生RDF图数据库系统——gStore

- gStore是北京大学王选计算机研究所开发的知识图谱管理系统，主要特点是建立针对知识图谱图结构构建基于签章树的索引
- 下载地址：<http://www.gstore-pku.com/>



原生RDF图数据库系统——gStore

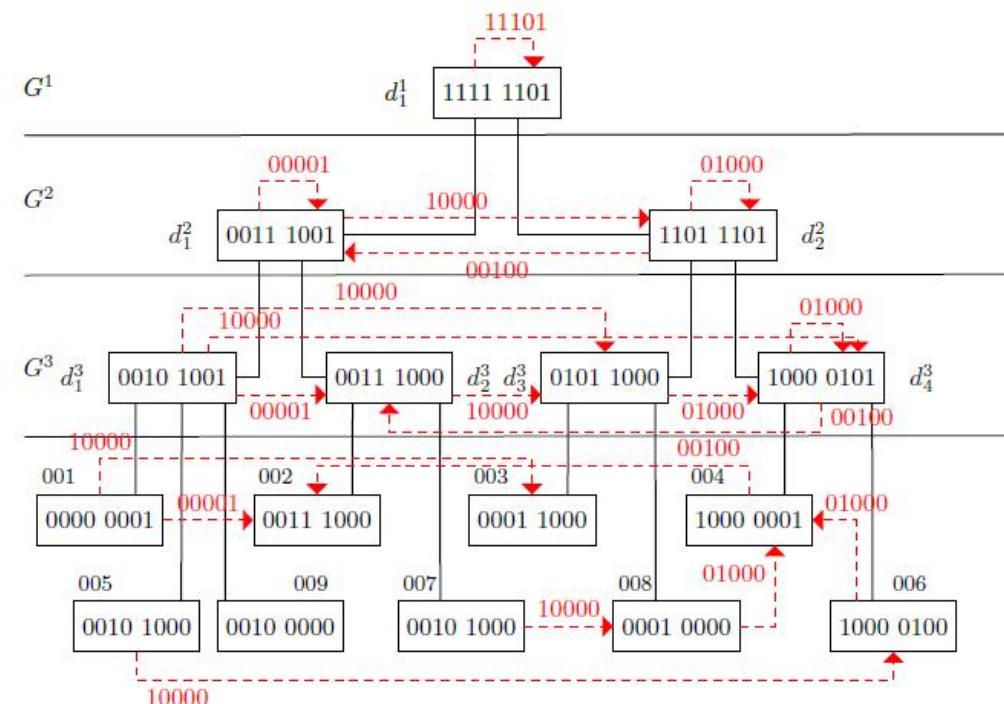
- 利用哈希函数将知识图谱中每个实体邻接表转化成一个位串，同时关系也转化成位串





原生RDF图数据库系统——gStore

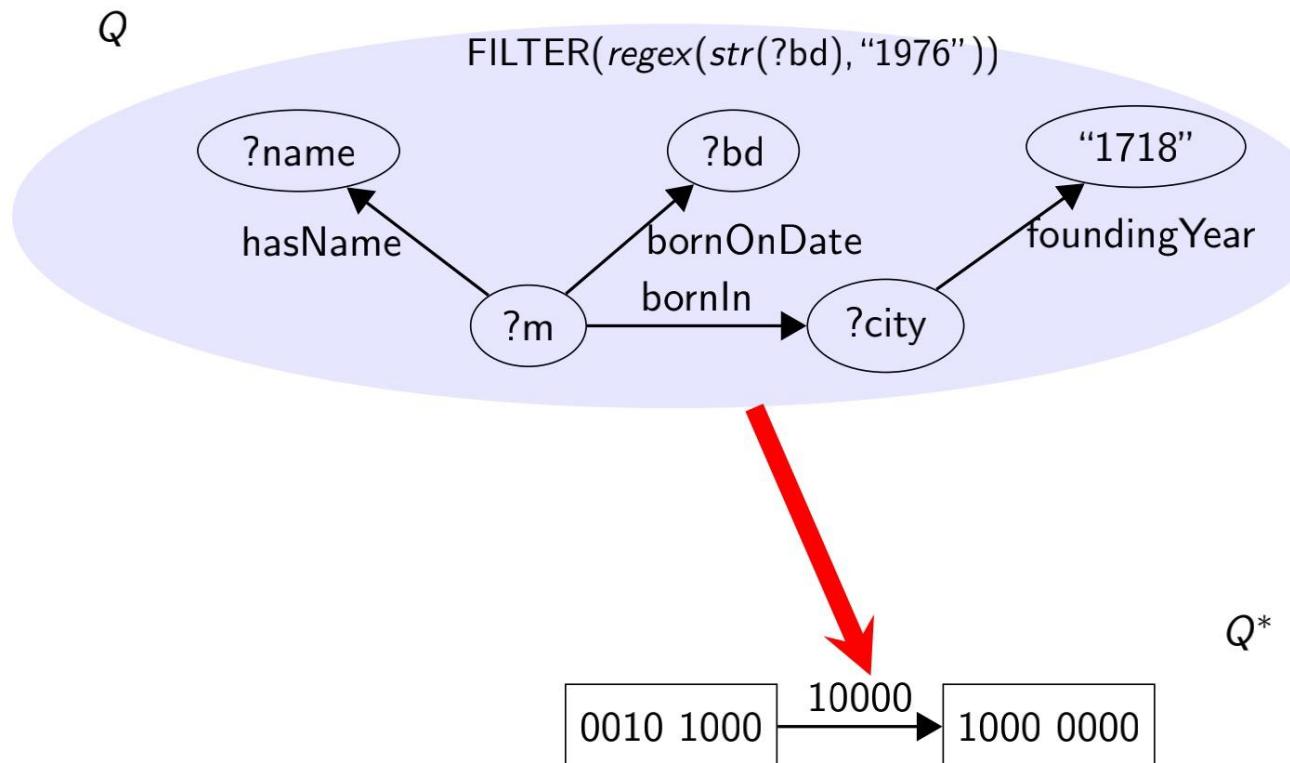
- 然后将上述实体的位串组织成一颗签章树，同时签章树中不同实体按照图谱中三元组连接上带关系位串的边





原生RDF图数据库系统——gStore

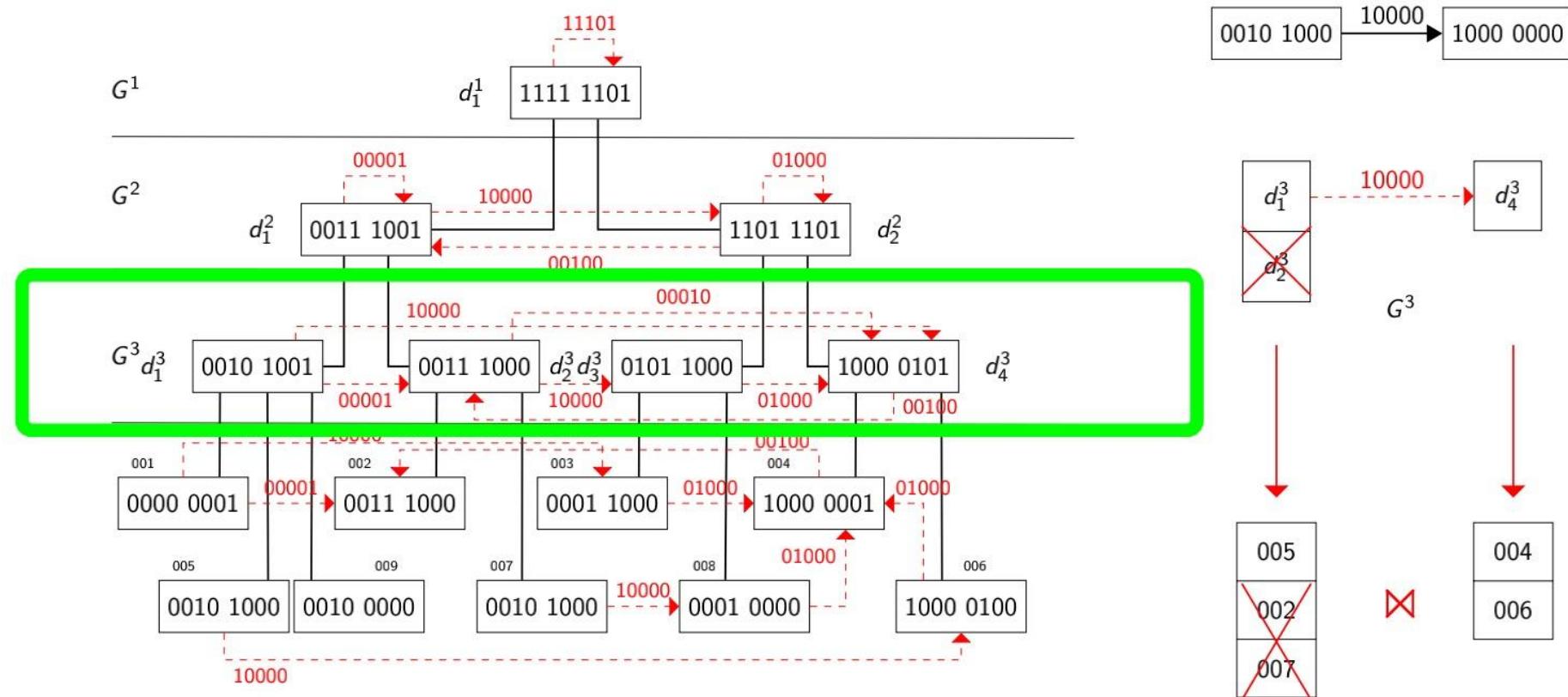
- 查询也转化成二进制位串





原生RDF图数据库系统——gStore

- 先得到候选，再连接



原生RDF图数据库系统——基于邻接矩阵的方法



- 这些系统首先给知识图谱中的主体、谓词和客体（或属性和属性值）进行编号，然后构建一个 $|V_s| \times |V_p| \times |V_o|$ 的三维矩阵M，其中 $|V_s|$ 、 $|V_p|$ 和 $|V_o|$ 分别表示主体、谓词及客体的数量



- BitMat就是一个典型的基于邻接矩阵存储知识图谱的系统。因为知识图谱中的常见查询是以谓词为常量的查询， BitMat 将上述 $|V_s| \times |V_p| \times |V_o|$ 的三维矩阵切分成 $|V_p|$ 个 $|V_s| \times |V_o|$ 和 $|V_o| \times |V_s|$ 的二维矩阵
- 同时，为了提高基于主体和客体查询的性能， BitMat 还切分出 $|V_s|$ 个 $|V_p| \times |V_o|$ 的二维矩阵和 $|V_o|$ 个 $|V_p| \times |V_s|$ 的二维矩阵
- 由于实际的知识图谱数据往往比较稀疏， BitMat 对每个矩阵中的每一行交替存储0 和1 的数量以实现压缩



原生RDF图数据库系统优缺点

- 优点：基于整个图结构构建索引，对于复杂访问很高效
- 缺点：简单的访问不一定比基于关系数据库的方法快

基于关系数据库技术的属性图数据库系统



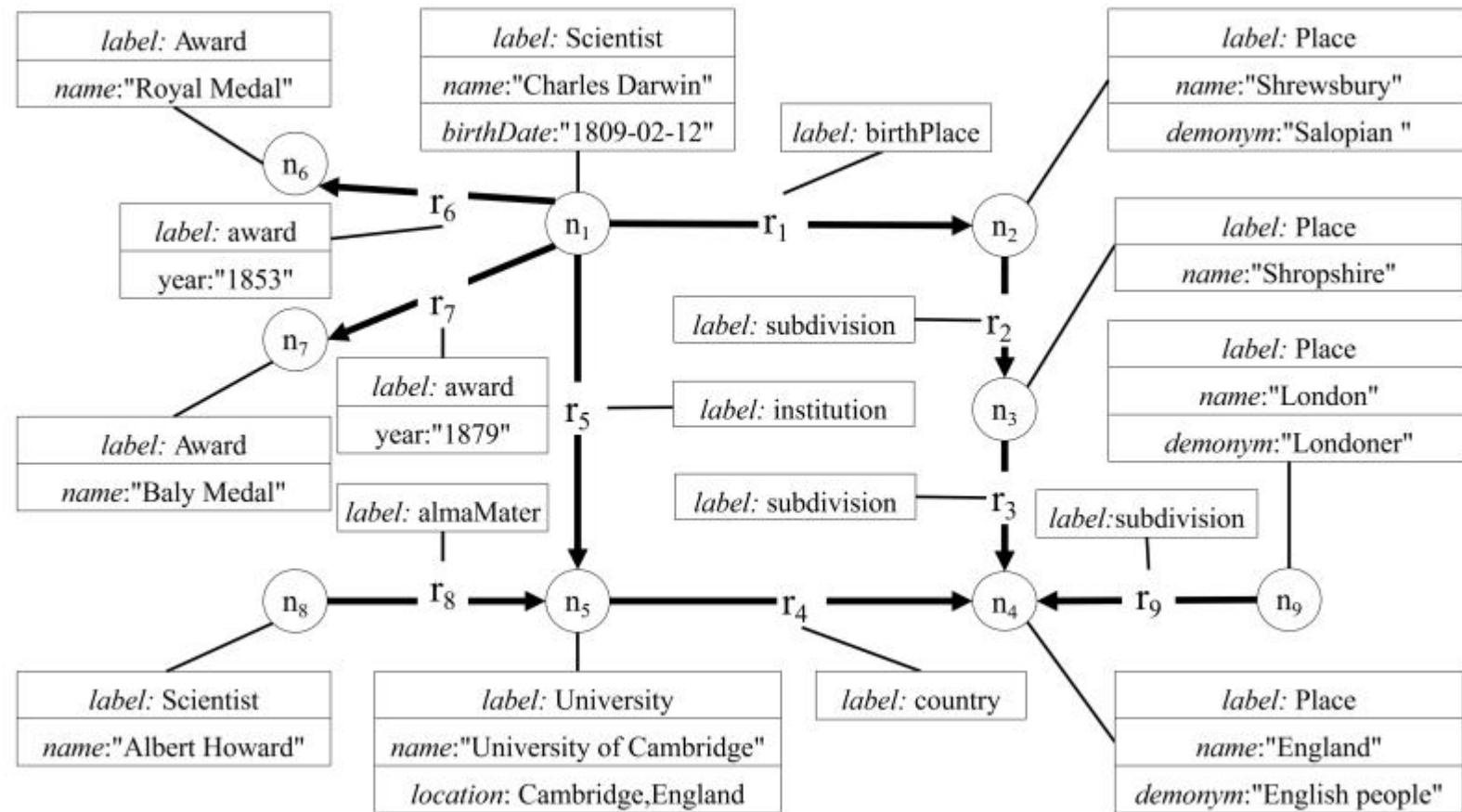
在前文介绍的 DB2 上基于属性表的 RDF 图数据存储与查询的基础上，IBM 还将相关技术扩展到了属性图数据的存储与查询，并整合了最新 DB2 里的 JSON 对象存储功能，最终开发了新系统——SQLGraph

这里，SQLGraph 支持的是 Gremlin 语言，所以它采用的是 JanusGraph 属性图模型，也就是每个点或者边都会有一个标签。接下来，将用属性图作为示例进行介绍。



基于关系数据库技术的属性图数据库系统

属性图示例





SQLGraph存储

SQLGraph 存储模式的主要思路是：将属性图上图结构基于邻接表用关系数据库进行存储，同时属性图上点和边的属性用 DB2 的 JSON 对象存储功能来进行存储



SQLGraph存储

在存储属性图上图结构的邻接表时，SQLGraph 主要用了两张表：出边主邻接表（Outgoing Primary Adjacency，简称 OPA）和入边主邻接表（Incoming Primary Adjacency，简称 OPA）。与 DB2 上基于属性表的 RDF 存储与查询方法类似，出边主邻接表（或入边主邻接表）也是宽表。



原生属性图数据库系统

其中，每一行对应一个点 s ，将其存储在出边主邻接表（或入边主邻接表）的 VID 列中。出边主邻接表（或入边主邻接表）每行有 $3 \times k + 1$ 列。在出边主邻接表（或入边主邻接表）中的每行，一个点作为边起点（或终点）时的所关联的边、边标签和终点（起点）分别存储在 EID_i 、 LBL_i 和 VAL_i 列中，其中 $0 \leq i \leq k$ 。假设 s 作为起点（或终点）的边上标签集合是 $E-LBL(s)$ 。



SQLGraph存储

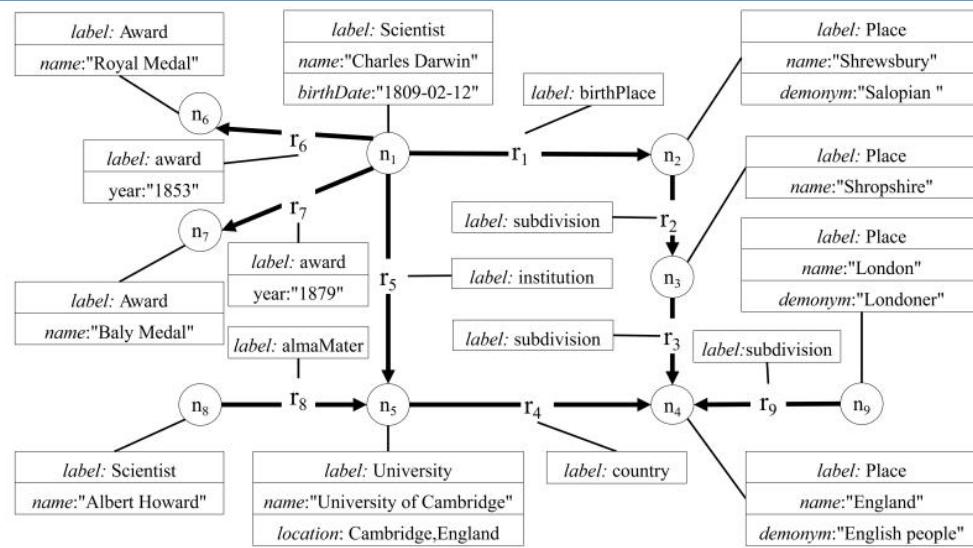
如果点 s 关联的边超过 k 个标签，即 $|E-LBL(s)| > k$ ，则在出边主邻接表（或入边主邻接表）使用 $((|E-LBL(s)| / k) + 1)$ 个行用于存储 s ，并把这些行的 SPILL 这一列都设置成 1。其中，第一个行存储 s 的前 k 个标签对应的边，第 $k+1$ 到第 $2 \times k$ 个标签对应的边存在第二行，该过程继续进行，直到为 s 存储了所有边。



属性图的出边主邻接表

VID	SPILL	EID ₁	LBL ₁	VAL ₁	EID ₂	LBL ₂	VAL ₂	EID ₃	LBL ₃	VAL ₃	EID ₄	LBL ₄	VAL ₄
n ₁	0	r ₁	birthPlace	n ₂	r ₅	institution	n ₅	null	award	lid:1	null	null	null
n ₂	0	r ₂	subdivision	n ₃	null								
n ₃	0	r ₃	subdivision	n ₄	null								
n ₅	0	r ₄	country	n ₄	null								
n ₈	0	null	null	null	null	null	null	r ₈	almaMater	n ₅	null	null	null
n ₉	0	r ₉	subdivision	n ₄	null								

- 假设 $k = 4$

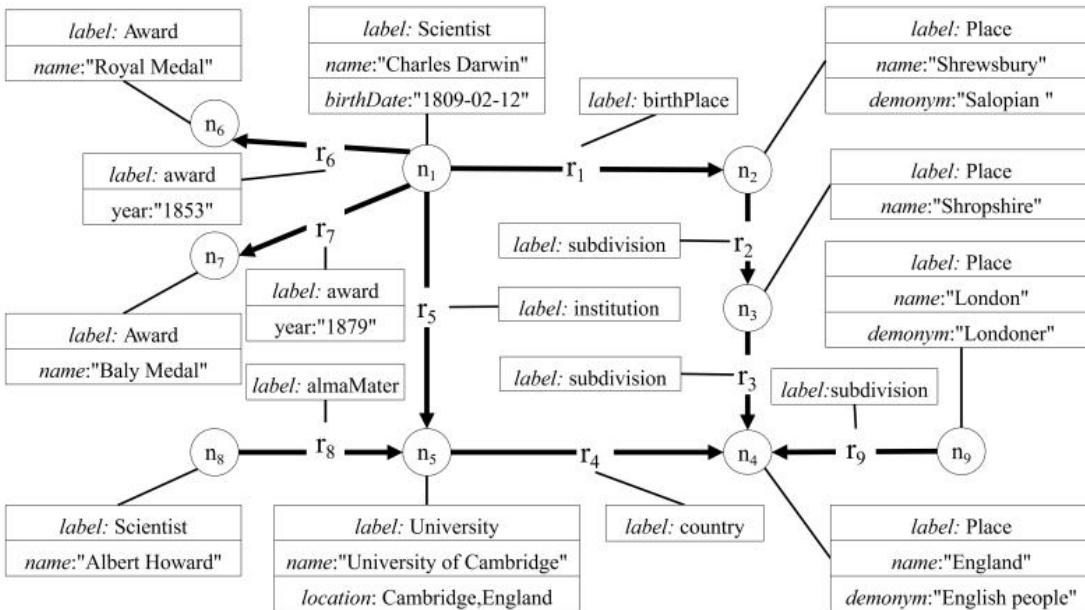




属性图的入边主邻接表

VID	SPILL	EID ₁	LBL ₁	VAL ₁	EID ₂	LBL ₂	VAL ₂	EID ₃	LBL ₃	VAL ₃	EID ₄	LBL ₄	VAL ₄
n ₂	0	r ₁	birthPlace	n ₁	null								
n ₃	0	r ₂	subdivision	n ₂	null								
n ₄	0	null	subdivision	lid:2	r ₄	country	n ₅	null	null	null	null	null	null
n ₅	0	null	null	null	r ₅	institution	n ₁	r ₈	almaMater	n ₈	null	null	null
n ₆	0	r ₆	award	n ₁	null								
n ₇	0	r ₇	award	n ₁	null								

假设 $k = 4$





SQLGraph存储

注意，出边主邻接表（或入边主邻接表）的每个边标签列对应多个边标签，而不是单一边标签。SQLGraph 也采用 DB2 上基于属性表的 RDF 存储与查询方法提出的两种方式来确定每个边标签列对应的边标签集合：一种是基于哈希，另一种是采用图着色算法。



SQLGraph存储

此外，对于多值边标签，即一个点作为起点（或终点）可能连接了多个相同边标签的边，SQLGraph 需要进行特殊处理。DB2 将相关边标签对应的起点（或终点）存储在另一张表中，称为出边二级邻接表（或入边二级邻接表）。在将多值边标签存储在出边二级邻接表（或入边二级邻接表）中时，SQLGraph 会为边标签对应终点（或起点）整体打包，并分配一个新的唯一标识符。



SQLGraph存储

然后，该标识符存储在出边主邻接表（或入边主邻接表）中，并与每个边标签对应的终点（或起点）具体关联。

出边二级
邻接表

	VALID	EID	VAL
lid:1	r ₆	n ₆	
lid:1	r ₇	n ₇	

	VALID	EID	VAL
lid:2	r ₃	n ₃	
lid:2	r ₉	n ₉	

入边二级
邻接表

点 n₁作为起点和边标签award连接了两个终点，分别是 n₆和n₇。因此，出边主邻接表中 award 边标签列对应的VAL₃ 列里值是 lid:1。而lid:1 在出边二级邻接表中对应了两个条边和两个终点，即r₆、n₆、r₇ 和n₇。



SQLGraph存储

而对于点和边上的属性，SQLGraph 都使用 DB2 的 JSON 对象存储功能来实现。我们给出了属性图的点属性表和边属性表。这里，边属性（EA）表不仅在 JSON 列中存储边属性，而且保留每条边的邻接信息，以方便查询处理。

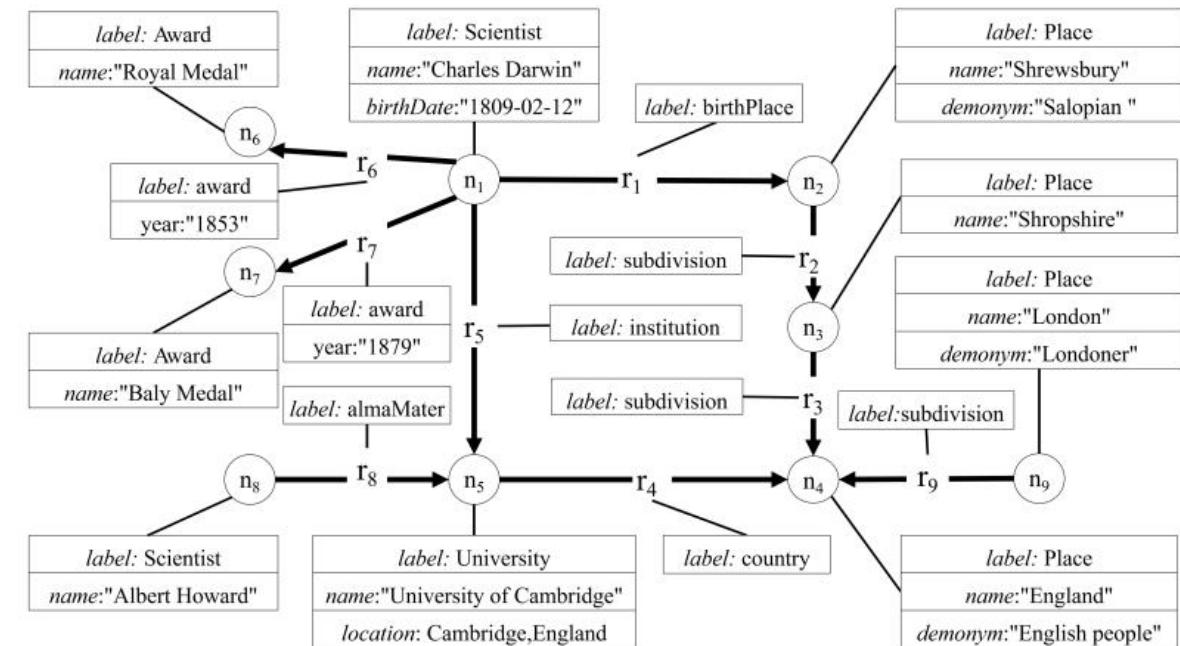


SQLGraph点属性表

下图为属性图的点属性表

VID ATTR (JSON object)

n ₁	{"name": "Charles Darwin", "birthDate": "1809-02-12"}
n ₂	{"name": "Shrewsbury", "demonym": "Salopian"}
n ₃	{"name": "Shropshire"}
n ₄	{"name": "England", "demonym": "English people"}
n ₅	{"name": "University of Cambridge", "location": "Cambridge,England"}
n ₆	{"name": "Royal Medal"}
n ₇	{"name": "Baly Medal"}
n ₈	{"name": "Albert Howard"}
n ₉	{"name": "London", "demonym": "Londoner"}

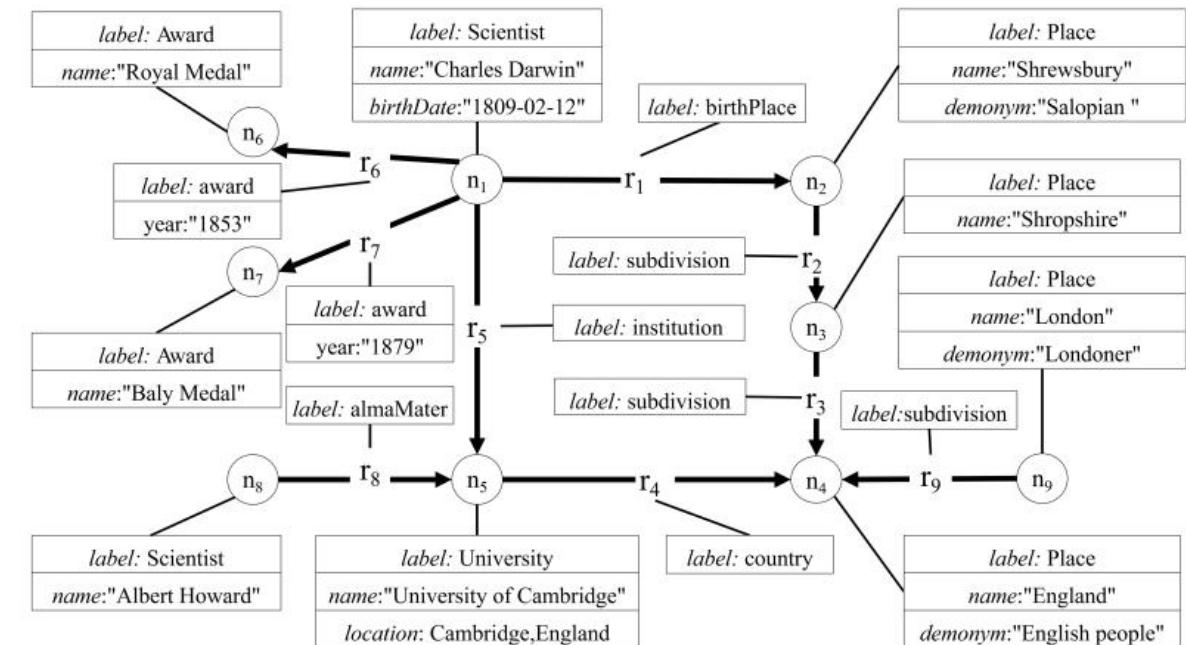




SQLGraph边属性表

下图为属性图的边属性表。这里，边属性（EA）表不仅在 JSON 列中存储边属性，而且保留每条边的邻接信息，以方便查询处理。

EID	INV	OUTV	LBL	ATTR (JSON object)
r ₁	n ₁	n ₂	birthPlace	{}
r ₂	n ₂	n ₃	subdivision	{}
r ₃	n ₃	n ₄	subdivision	{}
r ₄	n ₅	n ₄	country	{}
r ₅	n ₁	n ₅	institution	{}
r ₆	n ₁	n ₆	award	{"year": "1853"}
r ₇	n ₁	n ₇	award	{"year": "1879"}
r ₈	n ₅	n ₈	almaMater	{}
r ₉	n ₉	n ₄	subdivision	{}





SQLGraph查询处理

在查询处理的时候，SQLGraph 也和之前 DB2 上基于属性表的 RDF 图数据存储与查询方法一样，利用 SQL 模板将 Gremlin 查询转化为 SQL 查询。下表给出了 SQLGraph 中每个 Gremlin 语言步骤对应的 SQL 查询模板。

Operation	CTE Template for Query Translation
out pipe	(t_0 , SELECT t.val FROM t_{in} v, OPA p, TABLES(VALUES($p : val_0$, ..., $(p : val_n)$)AS t(val) WHERE v.val=p.vid AND t.val is not null), (t_1 ,SELECT COALESCE(s.val, p.val) AS val FROM t_0 p LEFT OUTER JOIN OSA s on p.val=s.valid))
in pipe	(t_0 , SELECT t.val FROM t_{in} v, IPA p, TABLES(VALUES($p : val_0$, ..., $(p : val_m)$)AS t(val) WHERE v.val=p.vid AND t.val is not null), (t_1 ,SELECT COALESCE(s.val, p.val) AS val FROM t_0 p LEFT OUTER JOIN ISA s on p.val=s.valid))
both pipe	(t_0 , SELECT t.val FROM t_{in} v, OPA p, TABLES(VALUES($p : val_0$, ..., $(p : val_n)$)AS t(val) WHERE v.val=p.vid AND t.val is not null), (t_1 , SELECT COALESCE(s.val, p.val) AS val FROM t_0 p LEFT OUTER JOIN OSA s on p.val=s.valid), (t_2 , SELECT t.val FROM t_{in} v, IPA p, TABLES(VALUES($p : val_0$, ..., $(p : val_m)$)AS t(val) WHERE v.val=p.vid AND t.val is not null), (t_3 ,SELECT COALESCE(s.val, p.val) AS val FROM t_2 p LEFT OUTER JOIN ISA s on p.val=s.valid), (t_4 ,SELECT * FROM t_1 UNION ALL SELECT * FROM t_3))
out vertex pipe	(t_0 , SELECT p.outv AS val FROM t_{in} v, EA p WHERE v.val=p.eid)
in vertex pipe	(t_0 , SELECT p.inv AS val FROM t_{in} v, EA p WHERE v.val=p.eid)
both vertex pipe	(t_0 , SELECT t.val FROM t_{in} v, EA p, TABLES(VALUES(p:outv), (p:inv)) AS t(val) WHERE v.val=p.eid)
out edges pipe	(t_0 , SELECT p.eid AS val FROM t_{in} v, EA p WHERE v.val=p.outv)
in edges pipe	(t_0 , SELECT p.eid AS val FROM t_{in} v, EA p WHERE v.val=p.inv)
both edges pipe	(t_0 , SELECT p.eid AS val FROM t_{in} v, EA p WHERE v.val=p.outv OR v.val=p.inv)
range filter pipe	(t_0 , SELECT * FROM t_{in} v LIMIT ? OFFSET ?)
duplicate filter pipe	(t_0 , SELECT DISTINCT v.val FROM t_{in} v)
id filter pipe	(t_0 , SELECT * FROM t_{in} v WHERE v.val == ?)
property filter pipe	(t_0 , SELECT v.* FROM t_{in} v, VA p WHERE v.val=p.vid AND JSON VAL(p.data, ? == ?))

interval filter pipe	$(t_0, \text{SELECT } v.* \text{ FROM } t_{in} v, VA p \text{ WHERE } v.val=p.vid \text{ AND JSON VAL}(p.data, ?) > ? \text{ AND JSON VAL}(p.data, ?) < ?)$
label filter pipe	$(t_0, \text{SELECT } v.* \text{ FROM } t_{in} v, EA p \text{ WHERE } v.val=p.eid \text{ AND } p.lbl == ?)$
except filter pipe	$(t_0, \text{SELECT } * \text{ FROM } t_{in} v \text{ WHERE } v.val \text{ NOT IN } (\text{SELECT } p.val \text{ FROM } t_k p))$
retain filter pipe	$(t_0, \text{SELECT } * \text{ FROM } t_{in} v \text{ WHERE } v.val \text{ IN } (\text{SELECT } p.val \text{ FROM } t_k p))$
cyclic path filter pipe	$(t_0, \text{SELECT } * \text{ FROM } t_{in} v \text{ WHERE } \text{isSimplePath}(v.path) == 1)$
back filter pipe	$\text{meta:cte} \cup (t_0, \text{SELECT } * \text{ FROM } t_{in} v \text{ WHERE } v.val \text{ IN } (\text{SELECT } p.path[0] \text{ FROM } \text{meta:}t_{out} p))$
and filter pipe	$\text{meta1.cte} \cup \text{meta2.cte} \cup (t_0, \text{SELECT } * \text{ FROM } t_{in} v \text{ WHERE } v.val \text{ IN } (\text{SELECT } p.path[0] \text{ FROM } \text{meta1.}t_{out} p1 \text{ INTERSECT } \text{SELECT } p.path[0] \text{ FROM } \text{meta2.}t_{out} p2))$
or filter pipe	$\text{meta1.cte} \cup \text{meta2.cte} \cup (t_0, \text{SELECT } * \text{ FROM } t_{in} v \text{ WHERE } v.val \text{ IN } (\text{SELECT } p.path[0] \text{ FROM } \text{meta1.}t_{out} p1 \text{ UNION } \text{SELECT } p.path[0] \text{ FROM } \text{meta2.}t_{out} p2))$
if-then-else pipe	$\text{test.cte} \cup \{\text{then.cte}_{in}, \text{else.cte}_{in}\} \cup \text{then.cte} \cup \text{else.cte} \cup (t_{out}, \text{SELECT } * \text{ FROM } \text{then.out} \text{ UNION ALL } \text{SELECT } * \text{ FROM } \text{else.out})$
split-merge pipe	$\text{meta1.cte} \cup \text{meta2.cte} \cup (t_0, \text{SELECT } * \text{ FROM } \text{meta1.}t_{out} p1 \text{ UNION ALL } \text{SELECT } * \text{ FROM } \text{meta2.}t_{out} p2)$
loop pipe	expand to fixed-length ctes or call stored procedure
as pipe	\emptyset , record the mapping between the as pipe and the (t_{out}) of the current translated ctes
aggregate pipe	\emptyset , record the mapping between the aggregate pipe and the (t_{out}) of the current translated ctes
add vertex	\emptyset , call stored procedure
add edge	\emptyset , call stored procedure
delete vertex	\emptyset , call stored procedure
delete edge	\emptyset , call stored procedure



原生属性图数据库系统

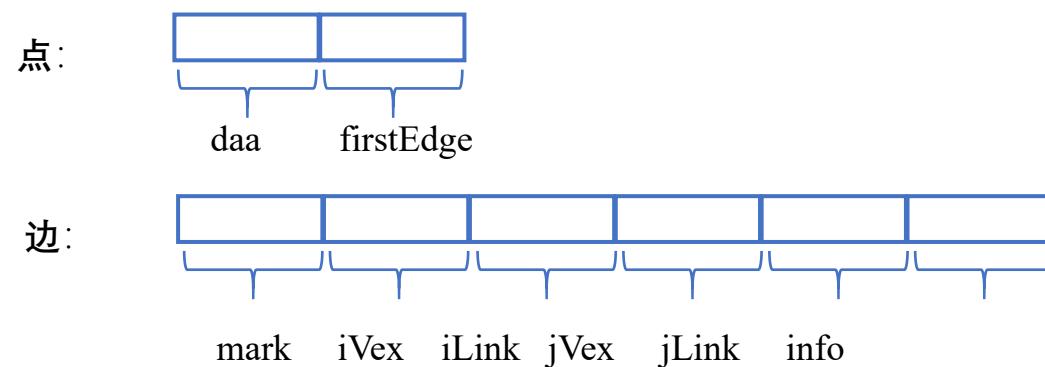
常用的属性图数据库系统也是基于邻接表的存储结构，典型就是 Neo4j。

总的来说，Neo4j 在存储图数据时借鉴邻接多重表实现了“无索引邻接”（index-free adjacency）的架构。为此，本节将首先简单介绍一下邻接多重表数据结构，然后介绍 Neo4j 中图数据存储形式。



邻接多重表

所谓邻接多重表，是一种针对无向图的链式存储结构。邻接多重表为图中各点建立一张链表，存储各点本身作为各链表的第一个元素，各链表中其他节点的结构与十字链表中相同。下图给出了邻接多重表存储结构。



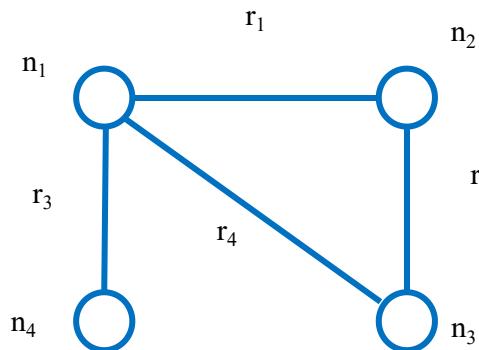


邻接多重表

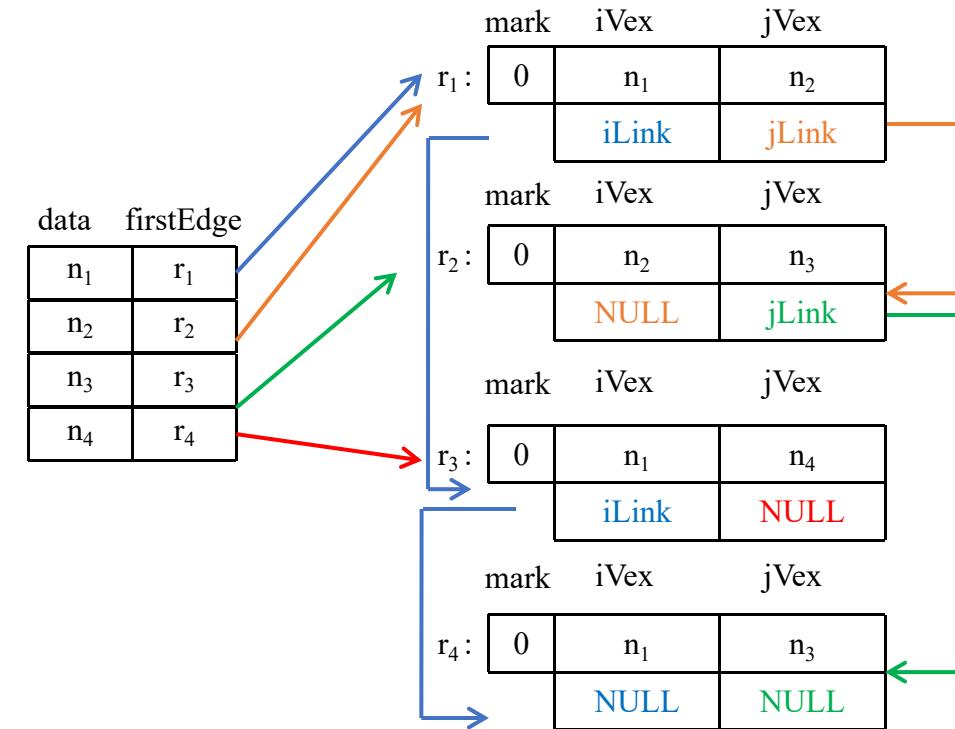
图中各点存储结构包括两个部分： data 和 firstEdge。 data 用来存储此点的数据； firstEdge 是一个指针，用于指向同该顶点有直接关联的第一条边的存储地址。每条边存储结构包括五个部分： mark 是标志位，用于标记此边是否被操作过； iVex 和 jVex 存储图中各边两端的点。 iLink 是一个指针，指向下一个存储与 iVex 有直接关联点的边； jLink 是一个指针，指向下一个存储与 jVex 有直接关联点的边； info 用于存储与该顶点有关的其他信息，比如无向网中各边的权。



邻接多重表



示例图

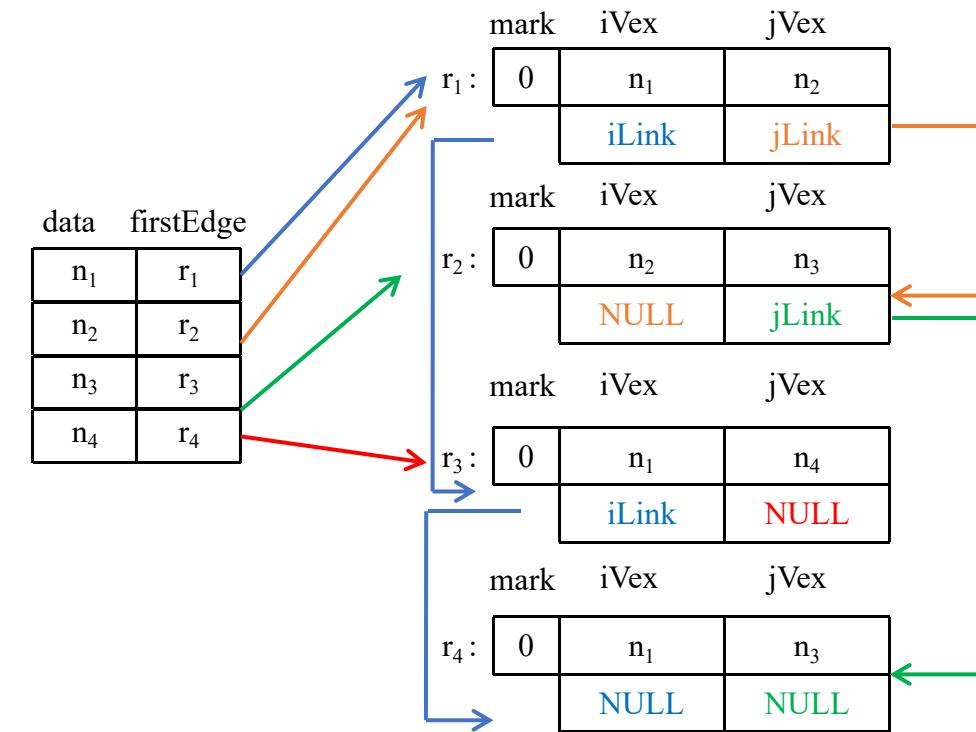


示例邻接多重表



邻接多重表

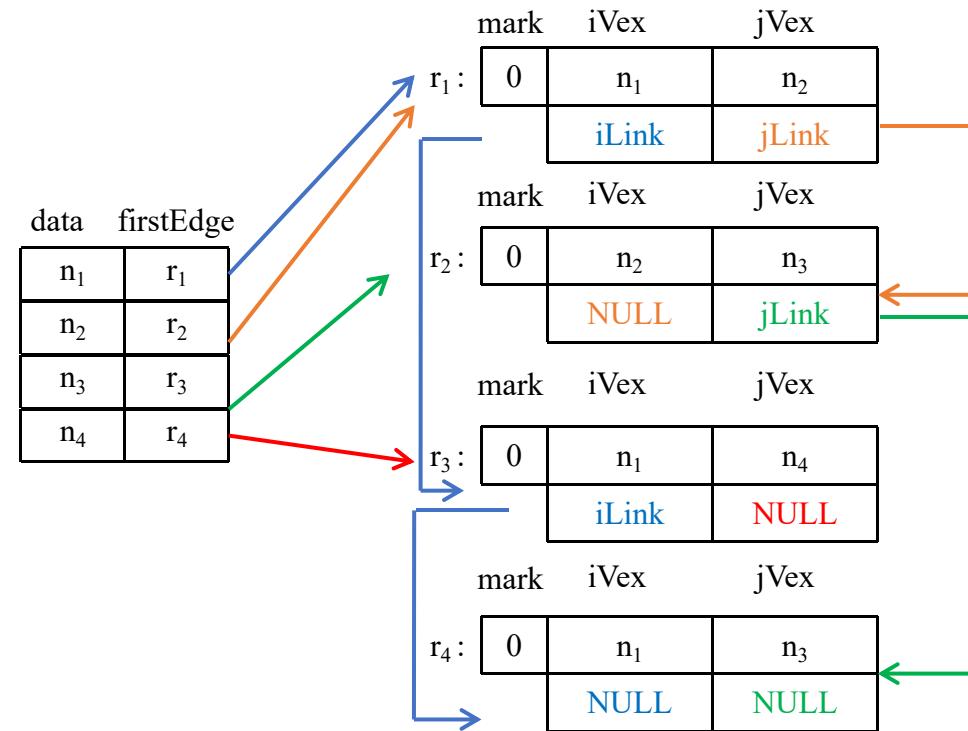
图中不同的各的链表用不同颜色表示。实际上所有点常常存储到一个数组中，同时边也按照某种预定义的顺序进行排列存储。边按照下标顺序排列。





邻接多重表

以 n_1 为例，边 r_1 的两个端点 $iVex$ 和 $jVex$ 分别为 n_1 和 n_2 。因此将 n_1 的 $firstEdge$ 指向 r_1 。 r_2 的两个端点都没有 n_1 ，故而转向扫描 r_3 。由于 r_3 的 $iVex$ 为 n_1 ，故而将 r_1 的 $iLink$ （关联的下一条边）指向 r_3 。由于 r_4 的 $iVex$ 为 n_1 ，故而将 r_3 的 $iLink$ 指向 r_4 ，由于 r_4 已经是最后一条边，故而将它的 $iLink$ 设置为 $NULL$ 。





Neo4j 图数据存储形式

Neo4j 扩展邻接多重表的结构，在多个不同的存储文件形式保存图数据。每个存储文件都包含图数据的部分内容（例如，点、边、标签和属性都有各自的存储文件）。这种存储责任的分割，特别是将图结构与属性数据分开存储，有助于实现高性能的图拓扑遍历。



Neo4j 图数据存储形式

同时，也是由于将图的结构和属性信息进行分开存储，所以针对图结构和属性相结合的图查询问题，这种分离式存储可能会带来更多的 I/O 代价。因此也有一些图数据库系统采用了图结构和属性进行联合存储的模式，例如 tuGraph 1 图数据库系统，我们在此不做更多展开介绍；本节重点关注 Neo4j 中的存储结构，具体的如图所示。

点 (Node, 15字节)



关系 (Relationship, 34字节)





Neo4j 图数据存储形式

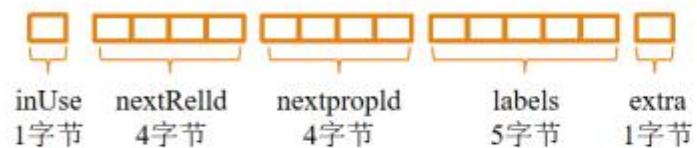
点存储文件 (neostore.nodestore.db) 用于保存图中的每个结点信息，本质上它和前面介绍的邻接多重表的结点存储是一致的。该文件与大多数 Neo4j 存储文件一样，是存储一个固定大小的记录，以支持在存储文件中快速查找点。每个记录的长度为 15 个字节，于是如果我们有一个 ID 为 100 的点，那么我们就可以知道其记录从文件的第 1500 个字节开始。因此，Neo4j 数据库可以用 $O(1)$ 时间复杂度计算出记录的位置。



Neo4j 图数据存储形式

类似于邻接多重表，如图所示，点记录的第一个字节是正在使用的标志，这个标志告诉数据库该记录当前是否正在被用于存储点，或者是否可以被回收利用来存储新的点（Neo4j 的.id 文件会跟踪未使用的记录）。接下来的四个字节（nextRelId）表示与点连接的第一条边的 ID，而随后的四个字节（nextPropId）表示点的第一个属性的 ID。

点 (Node, 15字节)



关系 (Relationship, 34字节)





Neo4j 图数据存储形式

Neo4j 中的属性是单独顺序存储在属性表中，后面有详细的介绍，这儿我们暂时忽略这部分的内容。另外五个字节（labels）被用于标签，指向此点的标签存储区；或者在标签相对较少的情况下，标签可以内联存储。最后一个额外字节中的部分比特位用于识别具有密集连接的点（高度数点），其余空间则保留以备将来使用。



Neo4j 图数据存储形式

关系存储文件用于保存图中的每条边的信息。与点存储类似，关系存储也由固定大小的记录组成。如图所示，每个关系记录包含了起始点（firstNode）和结束点的ID（secondNode）、指向关系类型的指针（关系类型存储在关系类型存储区），以及每个起始点和结束点的下一个（后继边）和前一个关系（前驱边）记录的指针，还有一个指示当前记录是否为关系链的第一个记录的标志。

点 (Node, 15字节)



关系 (Relationship, 34字节)

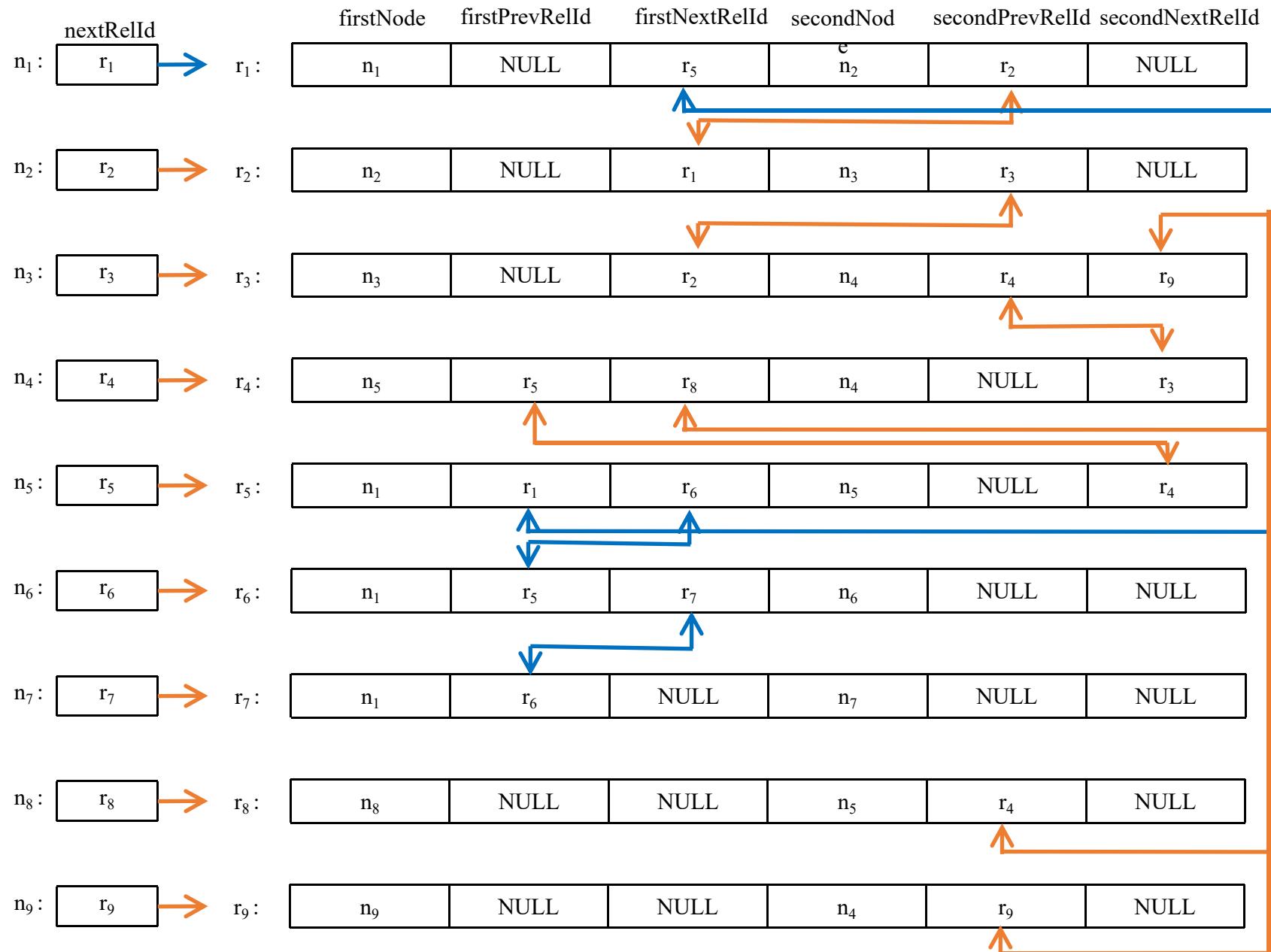




Neo4j 图数据存储形式

本质上，Neo4j 存储图数据的数据结构是使用邻接多重表进行存储。示例邻接多重表中边表中的 iLink 和 jLink 表示的是对应点所关联边的下一条边（即后继边）；在 Neo4j 中为了双向可访问，在边存储中也定义了每条边的关联边的上一条边（即前驱边）。

示例 Neo4j 属性图对应的图结构存储组织





Neo4j 图数据存储形式

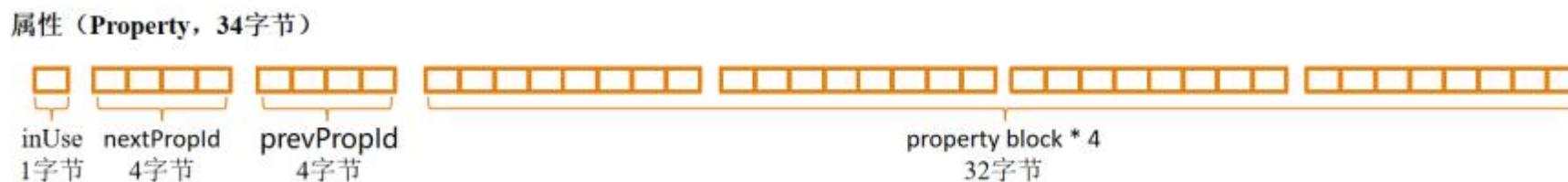
Neo4j 作为一种属性图数据库，能够将属性以键值对的形式关联到点和边上。因此，属性存储可以从点记录和边记录中引用的。因此，在点存储和边存储构成了图结构之外，Neo4j 还有属性存储文件以键值对的形式持久保存属性。



Neo4j 图数据存储形式

属性存储文件 (neostore.propertystore.db) 存储的是点或者边相关的属性集合。与点存储和边存储一样，属性记录具有固定的大小。

属性记录第 1 字节存储 nextPropId 与 prevPropId 的高 4bit; 接下来 4 字节为nextPropId，存储属性链表下一个属性记录的指针；再接下来 4 字节为 prevPropId，存储属性链表上一个属性记录的指针；最后 32 字节对应 4 个属性块，真正存放属性的内容。

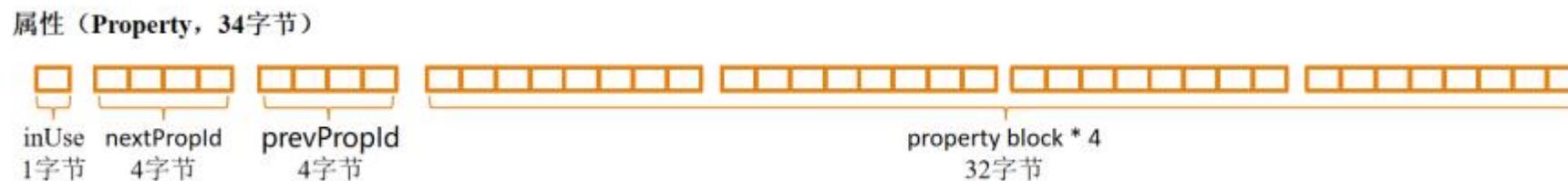




Neo4j 图数据存储形式

最后 32 字节对应 4 个属性块的第一个属性块（即前 8 个字节， 64 位）分成以下三部分：

- 前 24 位是一个指针，指向属性索引文件（neostore.propertystore.db.index）中所存储的属性名称；
- 之后 4 位是属性值的类型，包括 Integer、Boolean、Float、Long、Double、Byte、Character、Short、string、array；





Neo4j 图数据存储形式

最后 32 字节对应 4 个属性块的第一个属性块（即前 8 个字节， 64 位）分成以下三部分：

- 最后 36 位存储具体的值。对于 Integer、 Boolean、 Float、 Byte、 Character、 Short 类型的值，直接保存在这里；对于 Long，如果 36 位可以表示，则直接保存在这 36 位中，如果不够，则保存到后面那个属性块中； double 类型，直接保存到后面那个属性块中；对于 array 和 string，如果编码后在这 36 位加上后面 3 个属性块能保存，则直接保存，否则，相应的值就保存到动态字符串存储和动态字符串存储（neostore.propertystore.db.strings）和动态数组存储（neo_x0002_store.propertystore.db.arrays）中，而这 36 位保存其在动态字符串存储和动态数组存储中的指针。



原生属性图数据库系统的查询处理

我们以 Neo4j 中的查询执行方式来阐述原生属性图的查询处理。当执行查询时，Neo4j 将查询任务分解为操作符，每个操作符实现特定的工作。然后，操作符组合成一个称为执行计划的树状结构。执行计划中的每个操作符都表示为树中的点。每个操作符接受零行或更多行作为输入，并产生零行或更多行作为输出。这意味着一个操作符的输出成为下一个操作符的输入。在树中连接两个分支的操作符将两个输入流组合起来并产生单个输出。



原生属性图数据库系统的查询处理

执行计划中叶子结点通常是读取数据库中的操作符，形如 scan、seek 等，它们从数据库读取数据，输出结果。执行计划中非叶结点以孩子节点的输出作为输入。Neo4j有 100 多个操作符，常用操作符功能如表7.2所示。大多数操作符是惰性的（Lazy），即还没完全完成该算子的执行，就可以将部分结果行传给父节点，让父节点尽早执行。



原生属性图数据库系统的查询处理

这其实对应数据库查询并行优化中的流水线技术 (Pipeline)。也有部分操作符是积极的 (Eager)，必须获取子节点全部输入，进行计算，将最终结果传递给父节点，比如 Sort 操作符，Aggregation 操作符；这类操作通常会成为流水线技术的阻塞点。



原生属性图数据库系统的查询处理

执行计划的执行从树的叶节点开始。叶节点没有输入行，通常包括扫描和查找等操作符。这些操作符直接从 Neo4j 数据库中获取数据。随后，任何由叶节点产生的行都会按照流水线（pipeline）执行的模式输送到它们的父节点，这些父节点又会将它们的输出行输送到它们的父节点，依此类推，直到根点。最终，根点产生查询的最终结果。在可视化的查询计划的同时，Neo4j 还会返回显示每个操作符执行后预估的结果数量，即基数估计；这是查询优化的基础。



原生属性图数据库系统的查询处理

给定一个 Cypher 查询，Neo4j 提供 EXPLAIN 关键字来可视化地展示其产生的查询计划。例如给定本书中的 Cypher 查询示例，Neo4j 查询计划示例图给出了其查询计划。例如下的查询计划从叶子运算符 NodeByLabelScan 开始（这个操作符从点标签索引中根据 Place 这个标签提取具有所有 Place 节点）。



Neo4j查询处理

EXPLAIN

```
MATCH (p1:Place)
OPTIONAL MATCH (r:Scientist)-[:birthPlace]->(p1)
WITH p1, r AS SN
MATCH (p1)-[:subdivision*]->(p2:Place)
RETURN SN.birthDate, p1, count(p2) AS placeNum
```



查询中第一行 MATCH (p1:Place) 对应 Node_x0002_ByLabelScan 操作符，用来获取具有标签为 Place 的所有点，这个操作符执行后预估的结果数量为 4；

▼NodeByLabelScan	p1	:Place	4 estimated rows
▼OptionalExpand(All)	p1, r	r:Scientist	4 estimated rows
▼Projection	SN, p1, r	{SN : r}	4 estimated rows
▼VarLengthExpand(All)	p2, r, p1, SN	(p1)-[UNAMED97:subdivision*]->(p2:Place)	3 estimated rows
▼Filter()	p2, r, p1, SN	p2:Place	3 estimated rows
▼EagerAggregation	SN.birthDate, p1, placeNum		2 estimated rows
▼ProduceResults	SN.birthDate, p1, placeNum		2 estimated rows
Results			



Neo4j查询处理

EXPLAIN

```
MATCH (p1:Place)
OPTIONAL MATCH (r:Scientist)-[:birthPlace]->(p1)
WITH p1, r AS SN
MATCH (p1)-[:subdivision*]->(p2:Place)
RETURN SN.birthDate, p1, count(p2) AS placeNum
```



查询第二行 OPTIONAL MATCH (r:Scientist)-[:birthPlace]->(p1) 对应 OptionalExpand(All) 操作符，
用来从第一行所得到的标签为 Place 的点出发查找边
标签为 birthPlace 的入边，这个操作符执行后预估的
结果数量也为 4。

▼ NodeByLabelScan	p1	:Place	4 estimated rows
▼ OptionalExpand(All)	p1, r	r:Scientist	4 estimated rows
▼ Projection	SN, p1, r	{SN : r}	4 estimated rows
▼ VarLengthExpand(All)	p2, r, p1, SN	(p1)-[:subdivision*]->(p2:Place)	3 estimated rows
▼ Filter()	p2, r, p1, SN	p2:Place	3 estimated rows
▼ EagerAggregation	SN.birthDate, p1, placeNum		2 estimated rows
▼ ProduceResults	SN.birthDate, p1, placeNum		2 estimated rows
Results			

Neo4j查询处理

```
EXPLAIN  
MATCH (p1:Place)  
OPTIONAL MATCH (r:Scientist)-[:birthPlace]->(p1)  
WITH p1, r AS SN  
MATCH (p1)-[:subdivision*]->(p2:Place)  
RETURN SN.birthDate, p1, count(p2) AS placeNum
```



查询中第三行 WITH p_1, r AS SN 对应 Projection 操作符，用来对之前结果的每一行通过映射操作形成新表中一行，这个操作符执行后预估的结果数量依然为 4。

▼ NodeByLabelScan	
p1	:Place
	4 estimated rows
▼ OptionalExpand(All)	
p1, r	r:Scientist
	4 estimated rows
▼ Projection	
SN, p1, r	{SN : r}
	4 estimated rows
▼ VarLengthExpand(All)	
p2, r, p1, SN	(p1)-[UNAMED97:subdivision*]->(p2:Place)
	3 estimated rows
▼ Filter()	
p2, r, p1, SN	p2:Place
	3 estimated rows
▼ EagerAggregation	
SN.birthDate, p1, placeNum	
	2 estimated rows
▼ ProduceResults	
SN.birthDate, p1, placeNum	
	2 estimated rows
Results	





Neo4j查询处理

EXPLAIN

```
MATCH (p1:Place)
OPTIONAL MATCH (r:Scientist)-[:birthPlace]->(p1)
WITH p1, r AS SN
MATCH (p1)-[:subdivision*]->(p2:Place)
RETURN SN.birthDate, p1, count(p2) AS placeNum
```



查询中第四行 `MATCH (p1)-[:subdivision*]->(p2:Place)` 对

应 `VarLengthExpand(All)` 操作符，用来从第三行所得到的 `p1` 对应点出发查找边标签为 `subdivision` 的路径，这个操作符执行后预估的结果数量为 3；之后，Neo4j 执行查询计划中 `Filter()` 操作符，用来过滤出上一个运算符结果中 `p2` 标签为 `Place` 的行，这个操作符执行后预估的结果数量也为 3。

▼NodeByLabelScan	
p1	
:Place	4 estimated rows
▼OptionalExpand(All)	
p1, r	
r:Scientist	4 estimated rows
▼Projection	
SN, p1, r	
{SN : r}	4 estimated rows
▼VarLengthExpand(All)	
p2, r, p1, SN	
(p1)-[UNAMED97:subdivision*]->(p2:Place)	3 estimated rows
▼Filter()	
p2, r, p1, SN	
p2:Place	3 estimated rows
▼EagerAggregation	
SN.birthDate, p1, placeNum	2 estimated rows
▼ProduceResults	
SN.birthDate, p1, placeNum	2 estimated rows
Results	

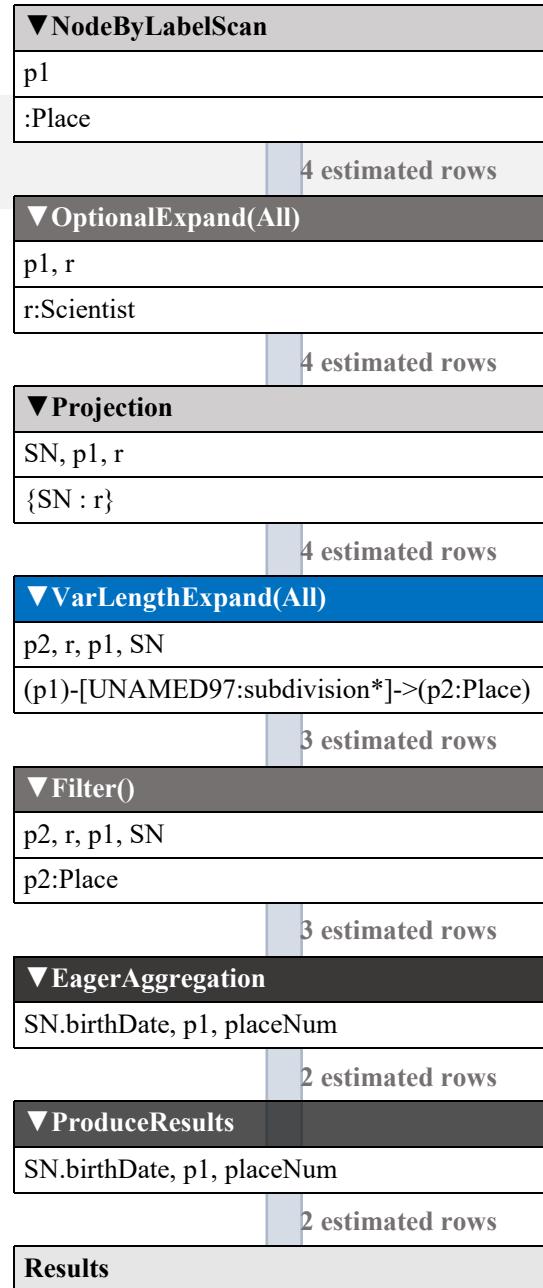


Neo4j查询处理

```
EXPLAIN
MATCH (p1:Place)
OPTIONAL MATCH (r:Scientist)-[:birthPlace]->(p1)
WITH p1, r AS SN
MATCH (p1)-[:subdivision*]->(p2:Place)
RETURN SN.birthDate, p1, count(p2) AS placeNum
```



查询中最后一行 RETURN SN.birthDate, p1, count(p2) AS placeNum 对应 EagerAggregation 操作符，将上一操作符的结果按照 SN.birthDate 和 p1 的结果进行分组，这个操作符执行后预估的结果数量也为 2；最终，Neo4j 执行查询计划中 ProduceResults，返回所有结果。





原生属性图数据库系统的查询处理

为了生成一个高效的执行计划，Cypher 查询规划器需要有关 Neo4j 数据库的信息。这包括哪些索引和约束可用，以及数据库维护的各种统计信息。Cypher 查询规划器使用这些信息来确定哪些访问模式将产生最佳执行计划。Neo4j 维护的统计信息包括具有特定标签的点数量、按类型划分的边数量、每个索引的选择性、从具有特定标签的起点或者终点出发的关联边数量等。



湖南大學
HUNAN UNIVERSITY

感谢观赏

—— 实事求是 敢为人先 ——