

第5章 知识存储与查询

随着知识体量的持续扩大，知识数据存储管理与查询处理的重要性不断提高。传统的文件存储方式已无法满足用户知识数据的各种应用需求。同时，知识图谱是知识数据的最主要形式，传统关系数据库在管理大规模知识图谱数据时与其图形结构模型间存在着明显的不匹配问题。为此，语义万维网领域发展出了专用于存储 RDF 数据的三元组库；而在数据库领域，则形成了用于处理属性图的图数据库技术。本章首先介绍知识数据模型，包括 RDF 图和属性图；然后讨论知识查询语言，包括 SPARQL 语言和 Cypher 语言；随后概要介绍知识图谱数据库系统的设计与实现，包括存储管理器、查询处理引擎和数据访问接口；最后，将使用 RDF 三元组库 Apache Jena 和属性图数据库 Neo4j 来进行关于知识存储与查询的实战演练。

5.1 知识数据模型

本节主要讨论将知识图谱作为知识数据模型的主流形式。知识图谱本质上是一种基于图结构的数据模型。在图论（graph theory）中，图的定义是一个由节点集合 V 和边集合 E 组成的二元组 $G = (V, E)$ 。在知识图谱中，节点用于表示实体，边则描述实体之间的关系；知识图谱还可以表达更加丰富的语义信息，通过实体与关系的属性、类型以及本体的定义来表达；这样的知识图谱数据模型能够有效地描绘出现实世界中各种事物之间的复杂联系。

实际上，知识图谱并没有一个统一的标准数据模型，通常可以将知识图谱刻画为一系列三元组的形式。例如，图 5.1 给出了红军长征途中“四渡赤水”战役的知识图谱数据模型描述，其中用椭圆表示实体，有毛泽东、朱德、四渡赤水、红军、长征、赤水河、贵州省、四川省共 8 个实体；用矩形表示属性值，有“1935.1.19”、“1935.3.22”和“436.5 公里”这 3 个属性值；还包含指挥、部分于、执行、地理位置、位于、相邻共 8 种表示关系的边，开始日期、结束日期、长度共 3 种表示属性的边。

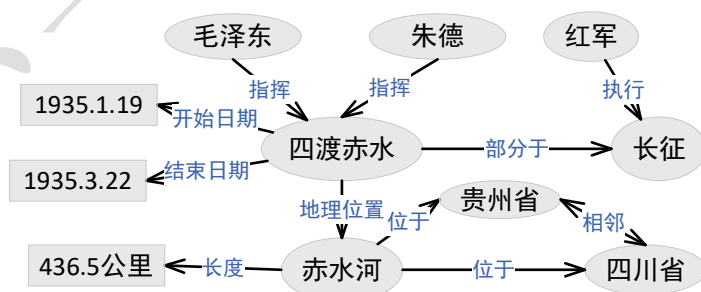


图 5.1 “四渡赤水”知识图谱的数据模型

在本节中，我们将深入探讨知识图谱的两种主要数据模型：RDF 图和属性图。

5.1.1 RDF 图

RDF 图数据模型是国际万维网联盟（W3C）制定的在语义万维网上表示和交换数据的工业标准。

1. 三元组

RDF 图是由三元组(s, p, o)组成的集合，一个三元组表示一个知识点，即陈述一个事实。在三元组(s, p, o)中， s 、 p 、 o 分别称为三元组的主语、谓语、宾语。如果宾语是实体，则三元组表示主语实体 s 与宾语实体 o 之间具有谓语关系 p ；如果宾语是属性值，则三元组表示主语实体 s 具有谓语属性 p 且其取值为宾语属性值 o 。事实上，RDF 图是一种有向图，一个三元组就是有向图中的一条边。

2. IRI

在 RDF 图数据模型中，使用国际资源标识符（International Resource Identifier，缩写 IRI）表示实体。IRI 是用于在万维网上唯一识别资源的标准标识系统。用 IRI 作为 RDF 图中的实体标识符，可以确保实体的全局唯一性。例如，在“四渡赤水”示例知识图谱中，可以用 `http://example.org/maoZedong` 作为人物实体毛泽东的标识符。在 RDF 图中，IRI 还用于标识作为谓语的关系或属性。例如，指挥关系用 `http://example.org/command` 表示。注意到，示例 IRI 使用了相同的前缀 `http://example.org`；实际上，在 RDF 数据模型中，一般使用同一个 IRI 前缀来标识位于同一个命名空间中的三元组（实体和关系），组成一个命名图（named graph），例如，示例 RDF 图中的三元组均位于 `http://longmarch.cn` 命名空间中。一个 RDF 图中的实体或关系可以来自不同的命名空间，例如，在“四渡赤水”示例知识图谱中，实体赤水河的 IRI 是 `http://geonames.org/river`，表示其是来自于地理数据命名空间 `http://geonames.org` 的实体。

图 5.2 给出了图 5.1 “四渡赤水”示例知识图谱对应的 RDF 图，其中，椭圆节点表示实体，矩形节点表示属性值。为了节省存储空间与编写方便，可以采用简短的前缀来表示命名空间 IRI，RDF 图中的 IRI 可采用“命名空间前缀:局部名称”的命名方法，例如，用 `ex` 表示 `http://example.org`，人物实体毛泽东由 `ex:maoZedong` 表示；用 `geo` 表示 `http://geonames.org`，赤水河用 `geo:river` 表示。

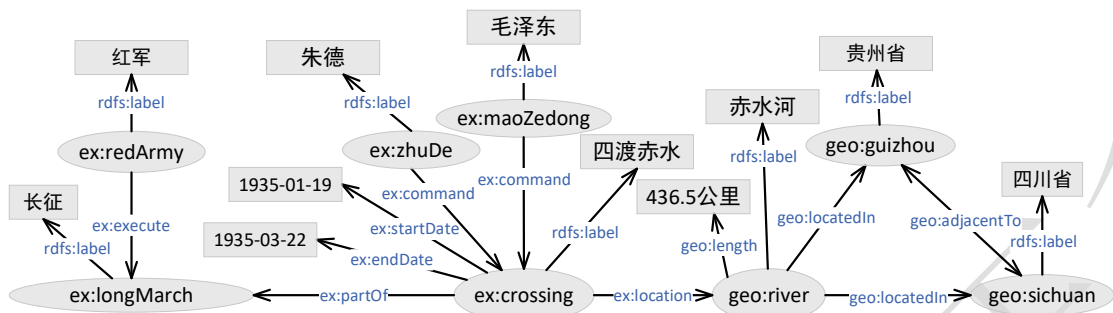


图 5.2 “四渡赤水”示例知识图谱的 RDF 图数据模型

有一些关系或属性较为通用，是大部分 RDF 图中都要使用的，例如实体名称属性；在示例 RDF 图中，使用来自 RDFS（RDF Schema）标准词汇中的属性 `rdfs:label`，其用于表示实体名称。表 5.1 给出了图 5.2 中 IRI 标识与实体/关系的对照表。

表 5.1 “四渡赤水”RDF 图中的 IRI 标识与实体/关系对照表

	IRI 标识（前缀缩写）	实体/关系
实体	<code>ex:maoZedong</code>	毛泽东
	<code>ex:zhuDe</code>	朱德
	<code>ex:redArmy</code>	红军
	<code>ex:longMarch</code>	长征
	<code>ex:crossing</code>	四渡赤水
	<code>geo:river</code>	赤水河
	<code>geo:guizhou</code>	贵州省
	<code>geo:sichuan</code>	四川省
关系	<code>ex:command</code>	指挥
	<code>ex:execute</code>	执行
	<code>ex:partOf</code>	部分于
	<code>ex:location</code>	地理位置
	<code>geo:locatedIn</code>	位于
	<code>geo:adjacentTo</code>	相邻
属性	<code>rdfs:label</code>	实体名称
	<code>ex:startDate</code>	开始日期
	<code>ex:endDate</code>	结束日期
	<code>geo:length</code>	河流长度

3. 字面量

在 RDF 图中，属性值称为字面量（literal）。例如，在图 5.2 中，三元组

```
ex:redArmy rdfs:label "红军" .
```

表示实体 `ex:redArmy` 的名称是“红军”，其中属性值“红军”是字面量，作为字面量的字符串要放在英文双引号“”中。字面量可具有不同的数据类型，RDF 标准采用 XML

Schema 标准中规定的类型系统，例如，“红军”是字符串类型 `xsd:string`，“1935-01-19”是日期类型 `xsd:date`。采用“^^”符合将数据类型附加在字面量后面，例如，“红军”^^`xsd:string`，“1935-01-19”^^`xsd:date`；由于字符串类型太过常用，RDF 标准中将不显式给出数据类型的字面量均作为字符串类型处理，因而“红军”就相当于“红军”^^`xsd:string`。在 RDF 中，字面量还支持多种语言标签，用“@”符合连接，例如，“红军”@zh 表示中文，“Red Army”@en 表示英文。

4. RDF 词汇

RDF 数据模型定义了标准词汇用于描述关于 RDF 图中元素（实体、关系）本身的信息，可以认为是 RDF 数据模型的元数据（metadata）描述。RDF 标准提供的 RDFS 语言可用于定义 RDF 数据模型的基本语义模式信息。例如，定义实体 `ex:crossing` 的类型是一个战役(`ex:Battle`)；`ex:command` 是一个关系，且其连接的主语和宾语类型分别为人物(`ex:Person`)和战役。

表 5.2 给出了 RDFS 标准中定义的主要词汇表。与面向对象程序设计语言类似，在 RDFS 中，用类(class)的概念表示实体所属的类型，所有类本身的类型被定义为 `rdfs:Class`；用属性(property)的概念表示实体间的关系，所有属性本身的类型被定义为 `rdfs:Property`。用 `rdf:type` 属性来声明一个实体实例与其所属的类之间的关系；子类关系 `rdfs:subClassOf` 和子属性关系 `rdfs:subPropertyOf` 分别用于构建类的层次和属性的层次；一个属性（即关系）的定义域（即允许出现什么类型的主语实体）和值域（即允许出现什么类型的宾语实体）分别用 `rdfs:domain` 属性和 `rdfs:range` 属性来定义。

表 5.2 RDFS 的主要词汇表

	RDFS 词汇	语法规式	描述
类	<code>rdfs:Class</code>	<code>C rdf:type rdfs:Class</code>	C 是一个 RDF 类
	<code>rdf:Property</code>	<code>P rdf:type rdf:Property</code>	P 是一个 RDF 属性
关系 (属性)	<code>rdf:type</code>	<code>I rdf:type C</code>	I 是 C 的实例
	<code>rdfs:subClassOf</code>	<code>C1 rdfs:subClassOf C2</code>	C1 是 C2 的子类
	<code>rdfs:subPropertyOf</code>	<code>P1 rdfs:subPropertyOf P2</code>	P2 是 P2 的子属性
	<code>rdfs:domain</code>	<code>P rdfs:domain C</code>	P 的定义域是 C
	<code>rdfs:range</code>	<code>P rdfs:range C</code>	P 的值域是 C

例如，在图 5.2 的基础上，可以增加 RDFS 相关定义：

```
ex:Battle rdf:type rdfs:Class .
ex:crossing rdf:type ex:Battle .
```

表示 `ex:Battle` 是一个类，即表示战役类型；四渡赤水实体（`ex:crossing`）是一场战役（`ex:Battle`），即其类型是 `ex:Battle`。

```
ex:command rdf:type rdf:Property .
ex:command rdfs:domain ex:Person .
```

```
ex:command rdfs:range ex:Battle .
```

表示 `ex:command` 是一个属性，即表示指挥关系；`ex:command` 属性的定义域是人物类型（`ex:Person`），值域是战役类型（`ex:Battle`）。

```
ex:maoZedong rdf:type ex:Person .  
ex:maoZedong ex:command ex:crossing .
```

表示毛泽东是一个人物（`ex:Person`），其指挥了四渡赤水战役（`ex:crossing`），这恰好符合 `ex:command` 的定义域和值域的要求。

```
ex:zhuDe rdf:type ex:Commander .  
ex:Commander rdfs:subClassOf ex:Person .  
ex:command rdfs:subPropertyOf ex:lead .
```

表示朱德是司令员（`ex:Commander`），`ex:Commander` 是 `ex:Person` 的子类，`ex:command` 是 `ex:lead` 的子属性。

需要注意的是，在 RDFS 词汇表中，`rdf:type` 和 `rdf:Property` 的前缀是 `rdf`，其余词汇的前缀为 `rdfs`，这是由于历史兼容性原因造成的。

除了 RDF 标准中规定的 RDFS 词汇表之外，还有若干常用的词汇表：

(1) FOAF (Friend of a Friend)

FOAF 是一个用于描述社交网络的 RDF 词汇表，它是最早被全球广泛使用的 RDF 词汇表之一。FOAF 旨在链接人们的社交网络信息，例如朋友、同事以及他们的活动和关系。这个词汇表包括了如 `Person`、`name`、`knows` 等类和属性，使得描述个人社交环境的数据能够在万维网上被机器理解和链接。

(2) Dublin Core

Dublin Core 由 Dublin Core Metadata Initiative 维护，是一套用于描述各种资源的元数据 RDF 词汇表。这个词汇表提供了一系列的属性，如 `creator`、`publisher`、`title`，适用于书籍、期刊文章、影视作品等多种资源。它的简洁性和灵活性使其在图书馆、档案馆和其他信息管理领域得到广泛应用。

(3) Schema.org

Schema.org 是由几大搜索引擎提供商（如 Google、Microsoft、Yahoo 和 Yandex）共同开发的一个词汇表，旨在通过为网页内容标记丰富的数据，改善搜索引擎的索引和理解网页内容的能力。这个词汇表包括了广泛的类别，如 `Person`、`Event`、`Organization` 和 `Product`，促进了结构化数据在网络上的使用。

(4) SKOS

SKOS（Simple Knowledge Organization System）是一套用于发布分类方案（如术语表

和词库)的 RDF 词汇表,自 2009 年起被 W3C 推荐使用。SKOS 允许组织通过网络发布他们的知识架构,例如图书馆的主题标头。美国国会图书馆就发布了其主题标头作为 SKOS 词汇表。SKOS 提供了一种简便的方式来表示、连接和共享复杂的知识结构。

这些 RDF 词汇表的价值在于重用(reuse)。越多的人使用同一套词汇表中的 IRI,这套词汇表就越有价值,因为它们通过网络效应(network effect)增加了数据的互操作性和可访问性。这意味着,在可能的情况下,应优先重用现有的词汇表,而不是创建新的。

W3C 制定的 Web 本体语言 OWL 词汇表可以与 RDFS 词汇表一起使用,用于为 RDF 数据模型定义更加丰富的语义信息。

3. 输出格式

RDF 图数据模型有多种输出格式,包括 Turtle 系列、JSON-LD、RDFa 和 RDF/XML。由于篇幅所限,这里只介绍 Turtle 系列格式,对于其他格式,读者可参阅相关标准文档。

RDF 图的 Turtle 系列输出格式包括: N-Triples、Turtle、TriG 和 N-Quads。

(1) N-Triples

N-Triples 是最简单的一种 RDF 图输出格式,其每行输出一条三元组。图 5.2 中 RDF 图的 N-Triples 格式输出如下:

```
<http://example.org/maoZedong> <http://www.w3.org/2000/01/rdf-schema#label> "毛泽东" .
<http://example.org/zhuDe> <http://www.w3.org/2000/01/rdf-schema#label> "朱德" .
<http://example.org/redArmy> <http://www.w3.org/2000/01/rdf-schema#label> "红军" .
<http://example.org/crossing> <http://www.w3.org/2000/01/rdf-schema#label> "四渡赤水" .
<http://example.org/crossing> <http://example.org/startDate> "1935-01-19"^^<http://www.w3.org/2001/XMLSchema#date> .
<http://example.org/crossing> <http://example.org/endDate> "1935-03-22"^^<http://www.w3.org/2001/XMLSchema#date> .
<http://example.org/longMarch> <http://www.w3.org/2000/01/rdf-schema#label> "长征" .
<http://geonames.org/river> <http://www.w3.org/2000/01/rdf-schema#label> "赤水河" .
<http://geonames.org/river> <http://geonames.org/length> "436.5" .
<http://geonames.org/guizhou> <http://www.w3.org/2000/01/rdf-schema#label> "贵州省" .
<http://geonames.org/sichuan> <http://www.w3.org/2000/01/rdf-schema#label> "四川省" .
<http://example.org/maoZedong> <http://example.org/command> <http://example.org/crossing> .
<http://example.org/zhuDe> <http://example.org/command> <http://example.org/crossing> .
<http://example.org/redArmy> <http://example.org/execute> <http://example.org/longMarch> .
<http://example.org/crossing> <http://example.org/partOf> <http://example.org/longMarch> .
<http://example.org/crossing> <http://geonames.org/location> <http://geonames.org/river> .
<http://geonames.org/river> <http://geonames.org/locatedIn> <http://geonames.org/guizhou> .
<http://geonames.org/river> <http://geonames.org/locatedIn> <http://geonames.org/sichuan> .
<http://geonames.org/guizhou> <http://geonames.org/adjacentTo> <http://geonames.org/sichuan> .
<http://geonames.org/sichuan> <http://geonames.org/adjacentTo> <http://geonames.org/guizhou> .
```

在每行表示的三元组中,IRI 标识符位于尖括号“<>”中,字面量在英文双引号中,字面量的数据类型用“^^”连接,每一行最后的英文句点“.”表示该条三元组的结尾。可以看到,N-Triples 格式每行一条三元组,简单明了,但实体和关系 IRI 存在大量重复的前缀,

对于大规模的 RDF 图，会造成较大的存储和通信开销。

(2) Turtle

Turtle 是“Terse RDF Triple Language”的简称，是 N-Triples 格式的扩展，其目标是提供一种比 N-Triples 更加紧凑、可读的 RDF 格式。为此，Turtle 格式引入了命名空间前缀、共同主语/谓语等语法形式。图 5.2 中 RDF 图的 Turtle 格式输出如下：

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ex: <http://example.org/> .
@prefix geo: <http://geonames.org/> .
# 创建人物节点
ex:maoZedong rdfs:label "毛泽东" .
ex:zhuDe rdfs:label "朱德" .
ex:redArmy rdfs:label "红军" .
# 创建四渡赤水节点
ex:crossing rdfs:label "四渡赤水" ;
    ex:startDate "1935-01-19"^^xsd:date ;
    ex:endDate "1935-03-22"^^xsd:date .
ex:longMarch rdfs:label "长征" .
geo:river rdfs:label "赤水河" ;
    geo:length "436.5" .
geo:guizhou rdfs:label "贵州省" .
geo:sichuan rdfs:label "四川省" .
ex:maoZedong ex:command ex:crossing .
ex:zhuDe ex:command ex:crossing .
ex:redArmy ex:execute ex:longMarch .
ex:crossing ex:partOf ex:longMarch .
ex:crossing ex:location geo:river .
geo:river geo:locatedIn geo:guizhou ,
    geo:sichuan .
geo:guizhou geo:adjacentTo geo:sichuan .
geo:sichuan geo:adjacentTo geo:guizhou .
```

Turtle 允许使用前缀来缩短 IRI 的表示。这通过@prefix 指令来声明，后面跟随一个短名称（前缀）、冒号和一个完整的 IRI。例如：在上述 Turtle 格式中，开头 4 行使用@prefix 声明了 4 个命名空间前缀，其中 rdfs 来自 RDF 词汇表，xsd 来自 XML Schema 词汇表（用于数据类型），ex 和 geo 是本例中自定义的词汇表。这些前缀在文档中定义一次后，可以在整个文档中使用，使 IRI 的重复使用更加简洁。

当一个主语有多个谓语宾语时，可以使用分号“;”来分隔，避免重复主语；当一个主语谓语有多个宾语时，可以使用逗号“,”来分隔，避免重复主语谓语。例如：主语 ex:crossing 具有 3 对谓语和宾语，主语 geo:river 具有 2 对谓语和宾语，用“;”分隔开，而省去了主语的重复出现；主语谓语 geo:river geo:locatedIn 具有 2 个宾语，用“,”分隔开，而省去了主语谓语的重复出现。

使用井号(＃)开始的行被视为注释，这些行在解析时会被忽略。

下面为“四渡赤水”RDF图添加RDFS相关三元组，在上述Turtle文档的基础上，添加如下三元组：

```
ex:maoZedong rdf:type ex:Person .
ex:zhuDe rdf:type ex:Commander .
ex:Commander rdfs:subClassOf ex:Person .
ex:redArmy rdf:type ex:Organization .
ex:crossing rdf:type ex:Battle .
ex:longMarch rdf:type ex:Campaign .
geo:river rdf:type geo:Basin .
geo:guizhou rdf:type geo:Province .
geo:sichuan rdf:type geo:Province .
ex:startDate rdfs:domain ex:Battle ; rdfs:range xsd:date .
ex:endDate rdfs:domain ex:Battle ; rdfs:range xsd:date .
geo:length rdfs:domain geo:Basin ; rdfs:range xsd:decimal .
ex:command rdfs:domain ex:Person ; rdfs:range ex:Battle .
ex:execute rdfs:domain ex:Organization ; rdfs:range ex:Campaign .
ex:partOf rdfs:domain ex:Battle ; rdfs:range ex:Campaign .
ex:location rdfs:domain ex:Battle ; rdfs:range geo:Area .
geo:Basin rdfs:subClassOf geo:Area .
ex:locatedIn rdfs:domain geo:Area ; rdfs:range geo:Province .
ex:adjacentTo rdfs:domain geo:Province ; rdfs:range geo:Province .
```

这些三元组定义了“四渡赤水”RDF图中实体的类型、关系的定义域和值域等信息。

(3) TriG

TriG是在Turtle语法的基础上进行扩展，增加了对命名图(named graph)的支持，允许在一个RDF文档中表示多个不同的RDF命名图。这种格式适合存储和传输具有多个子集的复杂RDF数据集，每个子集可作为一个单独的命名图。图5.2中RDF图的TriG格式输出如下：

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ex: <http://example.org/> .
@prefix geo: <http://geonames.org/> .

GRAPH <http://example.org/long_march>
{
    # 创建人物节点
    ex:maoZedong rdfs:label "毛泽东" .
    ex:zhuDe rdfs:label "朱德" .
    ex:redArmy rdfs:label "红军" .
    # 创建四渡赤水节点
    ex:crossing rdfs:label "四渡赤水" ;
        ex:startDate "1935-01-19"^^xsd:date ;
        ex:endDate "1935-03-22"^^xsd:date .
}
```



```

ex:longMarch rdfs:label "长征" .
ex:maoZedong ex:command ex:crossing .
ex:zhuDe ex:command ex:crossing .
ex:redArmy ex:execute ex:longMarch .
ex:crossing ex:partOf ex:longMarch .
ex:crossing ex:location geo:river .
}
GRAPH <http://geonames.org/chishui river>
{
    geo:river rdfs:label "赤水河" ; geo:length "436.5" .
    geo:guizhou rdfs:label "贵州省" .
    geo:sichuan rdfs:label "四川省" .
    geo:river geo:locatedIn geo:guizhou , geo:sichuan .
    geo:guizhou geo:adjacentTo geo:sichuan .
    geo:sichuan geo:adjacentTo geo:guizhou .
}

```

命名图以 GRAPH 关键字开头，跟着命名图的 IRI 标识，使用花括号“{}”将命名图中的三元组括起来。TriG 文件包含两个命名图，分别是表示长征的命名图 `<http://example.org/long_march>` 和表示地理数据赤水河的命名图 `<http://geonames.org/chishui_river>`。

(4) N-Quads

N-Quads 是 N-Triples 格式的命名图扩展，它允许每个三元组附带一个可选的所属命名图 IRI 名称。图 5.2 中 RDF 图的 N-Quads 格式输出如下：

```

<http://example.org/maoZedong> <http://www.w3.org/2000/01/rdf-schema#label> "毛泽东" <http://example.org/long_march> .
<http://example.org/zhuDe> <http://www.w3.org/2000/01/rdf-schema#label> "朱德" <http://example.org/long_march> .
<http://example.org/redArmy> <http://www.w3.org/2000/01/rdf-schema#label> "红军" <http://example.org/long_march> .
<http://example.org/crossing> <http://www.w3.org/2000/01/rdf-schema#label> "四渡赤水" <http://example.org/long_march> .
<http://example.org/crossing> <http://example.org/startDate> "1935-01-19"^^<http://www.w3.org/2001/XMLSchema#date>
<http://example.org/long_march> .
<http://example.org/crossing> <http://example.org/endDate> "1935-03-22"^^<http://www.w3.org/2001/XMLSchema#date>
<http://example.org/long_march> .
<http://example.org/longMarch> <http://www.w3.org/2000/01/rdf-schema#label> "长征" <http://example.org/long_march> .
<http://geonames.org/river> <http://www.w3.org/2000/01/rdf-schema#label> "赤水河" <http://geonames.org/chishui_river> .
<http://geonames.org/river> <http://geonames.org/length> "436.5" <http://geonames.org/chishui_river> .
<http://geonames.org/guizhou> <http://www.w3.org/2000/01/rdf-schema#label> "贵州省" <http://geonames.org/chishui_river> .
<http://geonames.org/sichuan> <http://www.w3.org/2000/01/rdf-schema#label> "四川省" <http://geonames.org/chishui_river> .
<http://example.org/maoZedong> <http://example.org/command> <http://example.org/crossing>
<http://example.org/long_march> .
<http://example.org/zhuDe> <http://example.org/command> <http://example.org/crossing> <http://example.org/long_march> .
<http://example.org/redArmy> <http://example.org/execute> <http://example.org/longMarch> <http://example.org/long_march> .
<http://example.org/crossing> <http://example.org/partOf> <http://example.org/longMarch> <http://example.org/long_march> .
<http://example.org/crossing> <http://geonames.org/location> <http://geonames.org/river> <http://example.org/long_march> .
<http://geonames.org/river> <http://geonames.org/locatedIn> <http://geonames.org/guizhou>
<http://geonames.org/chishui_river> .

```

```

<http://geonames.org/river> <http://geonames.org/locatedIn> <http://geonames.org/sichuan>
<http://geonames.org/chishui_river> .
<http://geonames.org/guizhou> <http://geonames.org/adjacentTo> <http://geonames.org/sichuan>
<http://geonames.org/chishui_river> .
<http://geonames.org/sichuan> <http://geonames.org/adjacentTo> <http://geonames.org/guizhou>
<http://geonames.org/chishui_river> .

```

N-Quads 通过将 N-Triples 格式扩充为四元组，简便地实现了多个 RDF 命名图的存储与传输；但与 N-Triples 一样，N-Quads 中重复的 IRI 前缀会占用大量空间并增加传输数据的通信开销。

4. 具体化

RDF 图没有内置支持实体（节点）和关系（边）上的属性。要表示一个实体的属性，可用一个属性三元组，其谓语是该属性的 IRI，宾语是属性值的字面量。要表示关系上的属性，需要用到所谓“具体化”（reification）方法，通过引入表示整个三元组的额外实体，并将关系的属性添加为该实体的谓语。例如：在图 5.2 所示的“四渡赤水”RDF 图中，要为“毛泽东指挥四渡赤水”的“指挥”关系添加“角色”属性“战略指挥”。通过“具体化”方法实现添加关系上属性的做法是：首先，引入一个新的实体 `ex:triple` 指代三元组“`ex:maoZedong ex:command ex:crossing`.”，其类型为 RDF 内置的三元组类型 `rdf:Statement`；然后，给 `ex:triple` 添加 `rdf:subject`、`rdf:predicate` 和 `rdf:object` 谓语（这些均为 RDF 词汇表中的标准属性），分别连接到所指代三元组的主语、谓语和宾语；最后，为 `ex:triple` 添加属性谓语“`ex:role`”和属性值宾语“战略指挥”，从而实现为“指挥”关系添加“角色”属性的效果。采用同样方法，添加“朱德指挥四渡赤水的角色是军事领导”。图 5.3 给出了在“四渡赤水”RDF 图中通过“具体化”方法添加关系属性后的效果图。

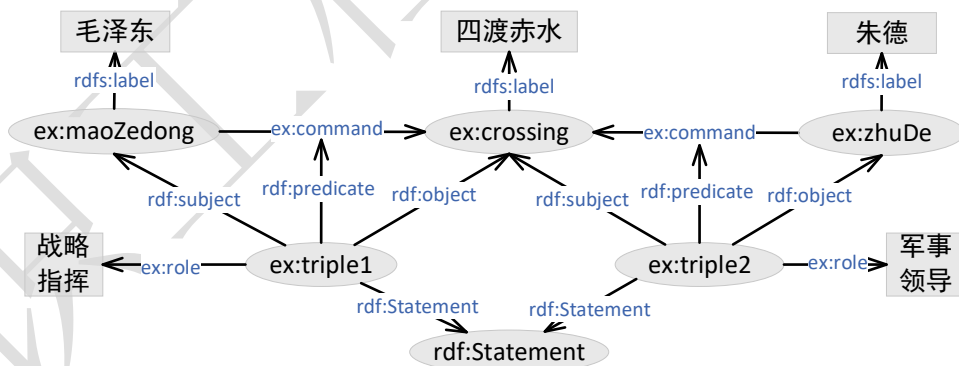


图 5.3 通过“具体化”方法添加关系上的属性

上述“具体化”方法所添加的三元组采用 Turtle 格式表示为：

```

# 毛泽东指挥四渡赤水 - 使用 Reification
ex:triple1 rdf:type rdf:Statement ;
  rdf:subject ex:maoZedong ;

```

```

    rdf:predicate ex:command ;
    rdf:object ex:crossing ;
    ex:role "战略指挥" .

# 朱德指挥四渡赤水 - 使用 Reification
ex:triple2 rdf:type rdf:Statement ;
    rdf:subject ex:zhuDe ;
    rdf:predicate ex:command ;
    rdf:object ex:crossing ;
    ex:role "军事领导" .

```

5. 空节点

在上述“具体化”方法中，表示三元组的实体 `ex:triple1` 和 `ex:triple2` 实际上没有必要给出全局的 IRI 标识。在 RDF 中，这种情况的实体可以采用空节点（blank node）来表达。空节点被用来指代临时的或匿名的实体，即允许 RDF 数据模型描述那些不需要具有 IRI 的实体。

在 Turtle 格式中，空节点被表示为一个以 `_:` 开头的标签。每个标签在同一 RDF 文档中应该是唯一的，但不同文档之间的空节点不能保证是相同的。用空节点将上述三元组的实体 `ex:triple1` 和 `ex:triple2` 分别表示为 `_:t1` 和 `_:t2`，如下：

```

# 毛泽东指挥四渡赤水 - 使用 Reification
_:t1 rdf:type rdf:Statement ;
    rdf:subject ex:maoZedong ;
    rdf:predicate ex:command ;
    rdf:object ex:crossing ;
    ex:role "战略指挥" .

# 朱德指挥四渡赤水 - 使用 Reification
_:t2 rdf:type rdf:Statement ;
    rdf:subject ex:zhuDe ;
    rdf:predicate ex:command ;
    rdf:object ex:crossing ;
    ex:role "军事领导" .

```

在图 5.2 中，给出了赤水河的长度为 436.5 公里，但上述各种 RDF 格式中均只有 436.5 的数值，没有表达出这个长度单位为公里。可以采用空节点方法并使用 RDF 内置属性 `rdf:value` 来实现数值和单位的同时表达，其 Turtle 格式如下：

```

geo:river geo:length [ rdf:value "436.5"^^xsd:decimal ;
                        geo:unit geo:kilometers ] .

```

这里采用了空节点的另一种表示方法，即采用方括号“`[]`”来表示一个空节点，方括号中是分号分隔的谓语宾语对，它们的共同主语为该空节点。

5.1.2 属性图

本节介绍另一种主要的知识数据模型，即属性图（property graph）。属性图数据模型目前被图数据库工业界所广泛采纳。属性图也是在一般有向图的基础上加以扩展的，基本要素包括节点、边和属性。属性图的形式定义如下：

一个属性图是一个 5 元组 $(V, E, \rho, \lambda, \sigma)$ ，其中：

- V 是节点的集合；
- E 是有向边的集合；
- $\rho: E \rightarrow (V \times V)$ 是将边关联到节点对的映射；
- $\lambda: (V \cup E) \rightarrow L$ 是将节点或边赋予标签的映射；
- $\sigma: (V \cup E) \times P \rightarrow Val$ 是将节点或边关联到属性键值对的映射。

这里， L 是标签（label）的集合， P 是属性键（即属性名称）的集合， Val 是属性值的集合。

图 5.4 给出了“四渡赤水”示例知识图谱的属性图数据模型。可以看到，该属性图包含 8 个节点、9 条边；节点和边不仅具有唯一编号，还具有标签，可以将标签看作是类型；除了节点上的属性，边 e_1 和 e_2 上也具有属性。可见，属性图对于实体和关系属性的表达非常方便，尤其是对于关系上属性的内置支持，不需要像 RDF 图那样，还要使用额外的“具体化”方法。

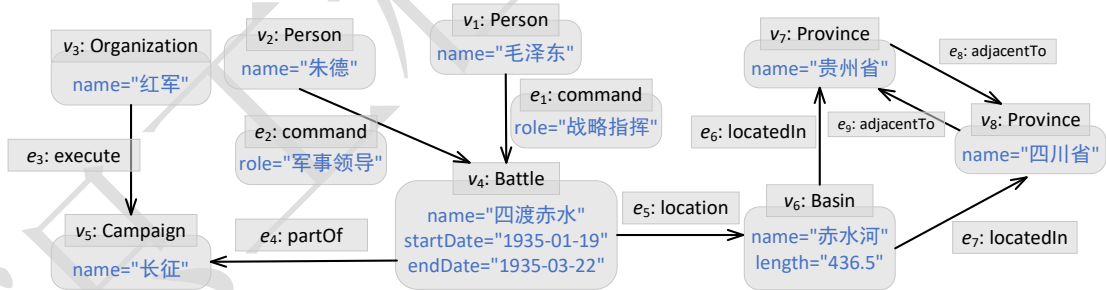


图 5.4 “四渡赤水”示例知识图谱的属性图数据模型

根据属性图的形式定义，上述“四渡赤水”属性图的 5 元组表示如下：

- $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$
- $E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9\}$
- $\rho(e_1) = (v_1, v_4), \rho(e_2) = (v_2, v_4), \rho(e_3) = (v_3, v_5), \rho(e_4) = (v_4, v_5), \rho(e_5) = (v_4, v_6), \rho(e_6) = (v_6, v_7), \rho(e_7) = (v_6, v_8), \rho(e_8) = (v_7, v_8), \rho(e_9) = (v_8, v_7)$

$$\rho(e_9) = (v_8, v_7)$$

- $\lambda(v_1) = \text{Person}, \quad \lambda(v_2) = \text{Person}, \quad \lambda(v_3) = \text{Organization}, \quad \lambda(v_4) = \text{partOf},$
 $\lambda(v_5) = \text{location}, \quad \lambda(v_6) = \text{locatedIn}, \quad \lambda(v_7) = \text{locatedIn}, \quad \lambda(v_8) = \text{adjacentTo},$
 $\lambda(e_1) = \text{command}, \quad \lambda(e_2) = \text{command}, \quad \lambda(e_3) = \text{execute}, \quad \lambda(e_4) = \text{partOf},$
 $\lambda(e_5) = \text{location}, \quad \lambda(e_6) = \text{locatedIn}, \quad \lambda(e_7) = \text{locatedIn}, \quad \lambda(e_8) = \text{adjacentTo},$
 $\lambda(e_9) = \text{adjacentTo}$
- $\sigma(v_1, \text{name}) = \text{"毛泽东"}, \quad \sigma(v_2, \text{name}) = \text{"朱德"}, \quad \sigma(v_3, \text{name}) = \text{"红军"},$
 $\sigma(v_4, \text{name}) = \text{"四渡赤水"}, \quad \sigma(v_4, \text{startDate}) = \text{"1935-01-19"},$
 $\sigma(v_4, \text{endDate}) = \text{"1935-03-22"}, \quad \sigma(v_5, \text{name}) = \text{"长征"}, \quad \sigma(v_6, \text{name}) = \text{"赤水河"},$
 $\sigma(v_6, \text{length}) = 436.5, \quad \sigma(v_7, \text{name}) = \text{"贵州省"}, \quad \sigma(v_8, \text{name}) = \text{"四川省"},$
 $\sigma(e_1, \text{role}) = \text{"战略指挥"}, \quad \sigma(e_2, \text{role}) = \text{"军事领导"}$

下面介绍属性图中的基本要素及其在 Cypher 语言中的表示方法。

1. 节点

节点 (node) 用于表示实体。例如，用 Cypher 语言的 CREATE 语句创建表示“四渡赤水”实体的节点：

```
CREATE (maoZedong:Person {name: "毛泽东"})
```

Cypher 语言的语法借鉴了“ASCII 艺术”(ASCII art) 的形式，即用字符来表示节点和边的形状。这里，CREATE 关键字后面的圆括号“()”代表一个节点，里面的内容指明节点的各类信息：maoZedong 为节点的标识 ID，Person 为节点的标签，花括号中给出属性键值对列表，如“name: “毛泽东””。

再如，创建表示“四渡赤水”实体的节点：

```
CREATE (crossing:Battle {name: "四渡赤水", startDate: "1935-01-19", endDate: "1935-03-22"})
```

2. 边

边 (edge) 用于表示实体之间的关系。例如，用 Cypher 语言的 CREATE 语句创建表示“毛泽东”与“四渡赤水”两个实体之间关系的边：

```
CREATE (maoZedong)-[:COMMAND {role: "战略指挥"}]->(crossing)
```

同样，借鉴“ASCII 艺术”，Cypher 语言使用字符拼出的箭头“-->”表示边，箭头的左右两端分别是源节点 (source node) 和目标节点 (target node) 的引用，这里的 maoZedong 和 crossing 为节点 ID，且这两个节点均已在之前创建了。箭头上的方括号中“COMMAND”表示边的类型，这条边不需要指明 ID，所以冒号“:”前面没有内容；后面花括号中是边上属性的列表，此例是“role: “战略指挥””。

3. 创建属性图

下面使用 Cypher 语言，创建图 5.4 中的“四渡赤水”知识图谱的属性图数据模型，完整代码如下：

```
// 创建节点
CREATE (maoZedong:Person {name: "毛泽东"})
CREATE (zhuDe:Person {name: "朱德"})
CREATE (redArmy:Organization {name: "红军"})
CREATE (crossing:Battle {name: "四渡赤水", startDate: "1935-01-19", endDate: "1935-03-22"})
CREATE (longMarch:Campaign {name: "长征"})
CREATE (river:Basin {name: "赤水河"})
CREATE (guizhou:Province {name: "贵州省"})
CREATE (sichuan:Province {name: "四川省"})

// 创建边
CREATE (maoZedong)-[:COMMAND {role: "战略指挥"}]->(crossing)
CREATE (zhuDe)-[:COMMAND {role: "军事领导"}]->(crossing)
CREATE (redArmy)-[:EXECUTE]->(longMarch)
CREATE (crossing)-[:PART_OF]->(longMarch)
CREATE (crossing)-[:LOCATION]->(river)
CREATE (river)-[:LOCATED_IN]->(guizhou)
CREATE (river)-[:LOCATED_IN]->(sichuan)
CREATE (guizhou)-[:ADJACENT_TO]->(sichuan)
CREATE (sichuan)-[:ADJACENT_TO]->(guizhou)
```

在 Cypher 语言中，一般遵守的命名规范是：

- 节点和边的 ID：小驼峰式命名法（lower camel case），即第一个单词首字母小写，后续单词首字母大写，如：redArmy、longMarch；
- 节点的标签：驼峰式命名法（camel case），即每个单词首字母大写，如：Person、Battel；
- 边的类型：使用全大写字母加下划线分隔符来命名，如：PART_OF、LOCATED_IN；
- 属性的键：小驼峰式命名法，如：name、startDate。

5.2 知识查询语言

如果说，知识数据模型是关于知识数据的结构组织方面，则在知识数据上的操作需要使用知识查询语言进行。RDF 图上的标准查询语言是 SPARQL；属性图上的主流查询语言包括 Cypher 和 Gremlin。

5.2.1 SPARQL 语言

国际万维网联盟（W3C）为 RDF 图上的查询和更新等操作专门制定了标准的数据查询

语言 SPARQL。SPARQL 是 SPARQL Protocol and RDF Query Language 的缩写，它提供了一种类似关系数据库 SQL 语言的语法来表达对 RDF 图数据的查询。SPARQL 允许用户编写查询来精确匹配复杂的图模式，从而检索和更新数据。作为 W3C 的官方推荐标准，SPARQL 的设计目的是提高数据的互操作性，支持跨多个数据源的查询，这使得它成为了开放链接数据和语义网应用的核心工具。

1. SPARQL 查询的语法结构

SPARQL 语言的查询语法结构由图 5.5 给出，包括：前缀声明、查询结果、定义数据源、查询模式和查询修饰符等，其中标明“(可选)”的部分表示该语法结构不是必须出现在一个查询中。下面通过例子来展示这些语法结构的基本功能。这里我们使用 5.1.1 节最后得到的完整 RDF 图作为查询数据源。



图 5.5 SPARQL 语言的查询语法结构

2. 三元组模式查询

查询 1：查询前 10 条三元组。

```
SELECT *
WHERE {
    ?s ?p ?o .
}
LIMIT 10
```

输出：

s	p	o
1<http://example.org/maoZedong>	<http://www.w3.org/2000/01/rdf-schema#label>	毛泽东
2<http://example.org/maoZedong>	<http://example.org/command>	<http://example.org/crossing>
3<http://example.org/maoZedong>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://example.org/Person>
4<http://example.org/zhuDe>	<http://www.w3.org/2000/01/rdf-schema#label>	朱德
5<http://example.org/zhuDe>	<http://example.org/command>	<http://example.org/crossing>
6<http://example.org/zhuDe>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://example.org/Commander>
7<http://example.org/redArmy>	<http://www.w3.org/2000/01/rdf-schema#label>	红军
8<http://example.org/redArmy>	<http://example.org/execute>	<http://example.org/longMarch>
9<http://example.org/redArmy>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://example.org/Organization>
10<http://example.org/crossing>	<http://www.w3.org/2000/01/rdf-schema#label>	四渡赤水

图 5.6 查询 1 的输出结果

讲解：这是一个最基本的三元组模式查询。WHERE 关键字后面的左右花括号中是查询模式，本例是最基本的一条三元组模式（triple pattern）：“?s?p?o.”，与 RDF 图的 Turtle 格式一样，英文句点“.”表示一条三元组模式的结尾；问号“?”开头的名称表示变量，这里“?s”、“?p”和“?o”均为变量，表示三元组的主语、谓语和宾语均是未知的，需要进行匹配填充。SELECT 关键字列出了要返回的结果变量，星号“*”表示要返回查询模式中的所有变量的匹配结果值。LIMIT 是一种查询修饰符，用于限定查询返回结果的数量，此处 10 表示只返回查询结果的前 10 行。查询 1 的输出结果如图 5.6 所示。

3. 基本图模式查询

查询 2：查询毛泽东指挥的战役的发生地点。

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ex: <http://example.org/>
SELECT ?river
WHERE {
    ?person rdfs:label "毛泽东" .
    ?person ex:command ?battle .
    ?battle ex:location ?place .
    ?place rdfs:label ?river .
}
```

输出：

river
1赤水河

图 5.7 查询 2 的输出结果

讲解：这是由多个三元组模式组成的一个基本图模式（basic graph pattern）查询，简称

为 BGP 查询。开头的 PREFIX 关键字定义前缀声明，这里定义了后面要用到的 rdfs 和 ex 两个前缀；在这个 BGP 查询中，第一个和第二个三元组模式具有共同的主语(变量?person)，第二个三元组模式的宾语是第三个三元组模式的主语（变量?battle），第三个三元组模式的宾语是第四个三元组模式的主语（变量?place）；这样，三元组模式之间通过共享主语以及主语和宾语的连接形成了一个查询模式图结构，如图 5.8 所示。实际上，BGP 查询就是一个带有变量的查询图，查询过程是在数据图中寻找与查询图映射匹配的所有子图，等价于图论中的子图同构（subgraph isomorphism）问题。查询 2 的输出结果如图 5.7 所示。

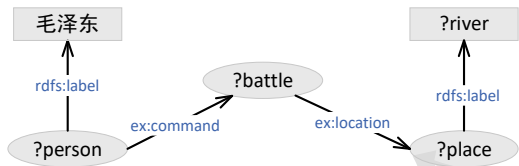


图 5.8 查询模式图结构

4. 使用条件过滤

查询 3：查询“四渡赤水”战役发生地河流及其长度，要求河流长度大于 436。

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ex: <http://example.org/>
PREFIX geo: <http://geonames.org/>
SELECT ?river ?length
WHERE {
    ?battle rdfs:label "四渡赤水" .
    ?battle ex:location ?place .
    ?place geo:length ?blank .
    ?blank rdf:value ?length .
    FILTER (?length >= 436)
    ?place rdfs:label ?river .
}
```

输出：

river	length
1赤水河	"436.5"^^<http://www.w3.org/2001/XMLSchema#decimal>

图 5.9 查询 3 的输出结果

讲解：关键字 FILTER 后面的圆括号中是条件表达式，用于指明过滤条件，对变量匹配结果进行按条件筛选，只保留满足条件的匹配。这里“?length>= 436”表示河流长度要大于等于 436，由于变量?length 的匹配值为 436.5，因而输出了结果“赤水河”。可以看到，在输出结果中显示 436.5 为 xsd:decimal 类型，这是 RDF 图中小数的默认类型。如果将过滤条件改为“?length>= 437”则没有输出结果。查询 3 的输出结果如图 5.9 所示。

5. 匹配可选模式

查询 4：查询“四渡赤水”战役的指挥官。

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ex: <http://example.org/>
SELECT ?name ?type
WHERE {
  ?battle rdfs:label "四渡赤水" .
  ?person ex:command ?battle .
  ?person rdfs:label ?name .
  ?person a ?type .
  OPTIONAL { ?person a ex:Commander } .
}
```

输出：

name	type
1毛泽东	<http://example.org/Person>
2朱德	<http://example.org/Commander>

图 5.10 查询 4 的输出结果

讲解：关键字 **OPTIONAL** 后面花括号中的三元组模式是可选的匹配条件，也就是说这些条件是可有可无的。“a”是 Turtle 格式的一个关键字，代表 **rdf:type**，即类型。这里 **OPTIONAL** 可选三元组模式指出 **?person** 的类型可以是 **ex:Commander** 也可以不是，甚至没有 **rdf:type** 属性都可以匹配为输出结果。**OPTIONAL** 可选匹配模式的使用增加了 SPARQL BGP 查询的灵活性。查询 4 的输出结果如图 5.10 所示。

6. 模式并集查询

查询 5：查询属于类型 **ex:Battle** 或类型 **ex:Campaign** 的实体。

```
PREFIX ex: <http://example.org/>
SELECT ?s1 ?s2
WHERE {
  { ?s1 a ex:Battle } UNION { ?s2 a ex:Campaign }
}
```

输出：

s1	s2
1<http://example.org/crossing>	
2	<http://example.org/longMarch>

图 5.11 查询 5 的输出结果

讲解：关键字 **UNION** 用于将两个 BGP 查询的结果以并集的方式进行合并，即只要能匹配上两个 BGP 中的至少一个就可以。该查询要求查找属于类型 **ex:Battle** ({ ?s1 a

ex:Battle }) 或类型 ex:Campaign 的实体 ({ ?s2 a ex:Campaign }), 使用 UNION 关键字将这两个 BGP 查询合并起来, 就可以返回所需结果。查询 5 的输出结果如图 5.11 所示。

7. 命名图

我们知道, RDF 图是三元组的集合。在 RDF 数据库中, 以 RDF 数据集 (RDF dataset) 的形式来管理多个 RDF 图。在一个 RDF 数据集中, 有一个默认图 (default graph) 以及零个或多个命名图 (named graph)。使用关键字 GRAPH 可以在 SPARQL 查询中指定当前用于执行匹配的命名图。

将 5.1.1 节中的完整 RDF 图默认图。将该 RDF 图拆分为前后两部分作为命名图 graph1 和 graph2 如下:

命名图 graph1, 包含默认图中的实例三元组。

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ex: <http://example.org/> .
@prefix geo: <http://geonames.org/> .
ex:maoZedong rdfs:label "毛泽东" .
ex:zhuDe rdfs:label "朱德" .
ex:redArmy rdfs:label "红军" .
ex:crossing rdfs:label "四渡赤水" ;
    ex:startDate "1935-01-19"^^xsd:date ;
    ex:endDate "1935-03-22"^^xsd:date .
ex:longMarch rdfs:label "长征" .
geo:river rdfs:label "赤水河" .
geo:guizhou rdfs:label "贵州省" .
geo:sichuan rdfs:label "四川省" .
ex:maoZedong ex:command ex:crossing .
ex:zhuDe ex:command ex:crossing .
ex:redArmy ex:execute ex:longMarch .
ex:crossing ex:partOf ex:longMarch .
ex:crossing ex:location geo:river .
geo:river geo:locatedIn geo:guizhou .
geo:river geo:locatedIn geo:sichuan .
geo:guizhou geo:adjacentTo geo:sichuan .
geo:sichuan geo:adjacentTo geo:guizhou .
geo:river geo:length [ rdf:value "436.5"^^xsd:decimal ;
    geo:unit geo:kilometers ] .
```

命名图 graph2, 包含默认图中的模式三元组。

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ex: <http://example.org/> .
```

```

ex:maoZedong rdf:type ex:Person .
ex:zhuDe rdf:type ex:Commander .
ex:Commander rdfs:subClassOf ex:Person .
ex:redArmy rdf:type ex:Organization .
ex:crossing rdf:type ex:Battle .
ex:longMarch rdf:type ex:Campaign .
geo:river rdf:type geo:Basin .
geo:guizhou rdf:type geo:Province .
geo:sichuan rdf:type geo:Province .
ex:startDate rdfs:domain ex:Battle ; rdfs:range xsd:date .
ex:endDate rdfs:domain ex:Battle ; rdfs:range xsd:date .
geo:length rdfs:domain geo:Basin ; rdfs:range xsd:decimal .
ex:command rdfs:domain ex:Person ; rdfs:range ex:Battle .
ex:execute rdfs:domain ex:Organization ; rdfs:range ex:Campaign .
ex:partOf rdfs:domain ex:Battle ; rdfs:range ex:Campaign .
ex:location rdfs:domain ex:Battle ; rdfs:range geo:Area .
geo:Basin rdfs:subClassOf geo:Area .
ex:locatedIn rdfs:domain geo:Area ; rdfs:range geo:Province .
ex:adjacentTo rdfs:domain geo:Province ; rdfs:range geo:Province .

```

查询 6：查询 ex:crossing 实体在命名图 ex:graph1 和 ex:graph2 中的谓词和宾语。

```

PREFIX ex: <http://example.org/>
SELECT *
FROM NAMED ex:graph1
FROM NAMED ex:graph2
WHERE {
    GRAPH ?g { ex:crossing ?p ?o }
}

```

输出：

p	o	g
1<http://www.w3.org/2000/01/rdf-schema#label>	四渡赤水	<http://example.org/graph1>
2<http://example.org/startDate>	"1935-01-19"^^<http://www.w3.org/2001/XMLSchema#date>	<http://example.org/graph1>
3<http://example.org/endDate>	"1935-03-22"^^<http://www.w3.org/2001/XMLSchema#date>	<http://example.org/graph1>
4<http://example.org/partOf>	<http://example.org/longMarch>	<http://example.org/graph1>
5<http://example.org/location>	<http://geonames.org/river>	<http://example.org/graph1>
6<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://example.org/Battle>	<http://example.org/graph2>

图 5.12 查询 6 的输出结果

讲解：关键字 FROM NAMED 指明了两个命名图的 IRI 名称，分别为 ex:graph1 和 ex:graph2。在 BGP 查询中，关键字 GRAPH 指定在哪个命名图上进行匹配。这里“GRAPH ?g”表示在每个命名图（即 ex:graph1 和 ex:graph2）中进行匹配，变量?g 绑定到匹配结果所在的命名图 IRI。在查询结果中可见，前 5 行结果来自命名图 ex:graph1，第 6 行结果来自命名图 ex:graph2。查询 6 的输出结果如图 5.12 所示。

查询 7：查询 `ex:crossing` 实体在默认图和命名图 `ex:graph2` 中的谓语和宾语。

```
PREFIX ex: <http://example.org/>
SELECT *
WHERE {
  { ex:crossing ?p ?o }
  UNION
  { GRAPH ex:graph2 { ex:crossing ?p ?o } }
}
```

输出：

p	o
1<http://www.w3.org/2000/01/rdf-schema#label>	四渡赤水
2<http://example.org/startDate>	"1935-01-19"^^<http://www.w3.org/2001/XMLSchema#date>
3<http://example.org/endDate>	"1935-03-22"^^<http://www.w3.org/2001/XMLSchema#date>
4<http://example.org/partOf>	<http://example.org/longMarch>
5<http://example.org/location>	<http://geonames.org/river>
6<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://example.org/Battle>
7<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://example.org/Battle>

图 5.13 查询 7 的输出结果

讲解：BGP 查询 “`{ ex:crossing ?p ?o }`” 在默认图上进行匹配，其查询结果为输出的第 1-6 行；关键字 `GRAPH` 指定在命名图 `ex:graph2` 中进行 BGP 查询 “`{ ex:crossing ?p ?o }`”，其查询结果为输出的第 7 行。查询 7 的输出结果如图 5.13 所示。

8. 内置函数

SPARQL 提供了一系列内置函数，这些函数可以在查询中使用，以执行各种操作，如数值计算、字符串处理、日期和时间处理等。这些函数增强了 SPARQL 的功能性，使其能够处理复杂的查询和数据转换需求。表 5.3 按照类别给出了常用的 SPARQL 内置函数及举例。需要说明的是，运算符（如算术运算符、逻辑与比较运算符）在 SPARQL 中也作为函数看待。

表 5.3 SPARQL 内置函数

类别	函数名称	举例
逻辑 比较	!, &&, , !=, <, <=, >, >=, IN, NOT IN	?date >= "1935-01-01" && ?date <= "1935-03-31"
条件	EXISTS, NOT EXISTS, IF, COALESCE	NOT EXISTS { ?x ex:part of ?y }
数学	+, -, *, /, abs, round, ceil, floor, RAND	round(?length * 2) > 436

字符串	STRLEN, SUBSTR, UCASE, LCASE, STRSTARTS, CONCAT, STREND, CONTAINS, STRBEFORE, STRAFTER	STRLEN(?province) = 3
日期时间	now, year, month, day, hours, minutes, seconds, timezone, tz	month(now()) < 4
测试	isURI, isBlank, isLiteral, isNumeric, bound	isURI(?person) !bound(?person)
构造	URI, BNODE, STRDT, STRLANG, UUID, STRUUID	STRLANG(?text, "en") = "hello"@en
访问	str, lang, dataType	lang(?title) = "en"
哈希	MD5, SHA1, SHA256, SHA512	BIND(SHA256(?email) AS ?hash)
其他	sameTerm, langMatches, regex, REPLACE	regex(?ssn, "\\d{3}-\\d{2}-\\d{4}")

查询 8：查询名称含有“赤水”且开始日期在 1935 年的战役，输出战役实体的 IRI、战役名称、战役年份（以“年”结尾）。

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ex: <http://example.org/>
SELECT ?battle ?label (CONCAT(str(year(?date)), "年") AS ?year)
WHERE {
  ?battle ex:startDate ?date FILTER (year(?date) = 1935)
  ?battle rdfs:label ?label FILTER (CONTAINS(?label, "赤水"))
}
```

输出：

battle	label	year
1 < http://example.org/crossing >	四渡赤水	1935年

图 5.14 查询 8 的输出结果

讲解：这里使用了日期函数 `year` 获取变量 `?date` 中年的数字并用“=”比较是否为 1935；使用了字符串函数 `CONTAINS` 来判断变量 `?label` 中字符串是否含有“赤水”两个字；使用字符串函数 `CONCAT` 将年份数字转化的字符串“`str(year(?date))`”与“年”字进行拼接，并使用 `AS` 关键字将该拼接字符串结果以 `year` 为变量名输出。查询 8 的输出结果如图 5.14 所示。

9. 联邦查询

SPARQL 提供联邦查询功能，使得用户能够在一个单一的 SPARQL 查询中访问分布在多个 SPARQL 终端（endpoint）上的数据。这项功能是通过 `SERVICE` 关键字实现的，它允许查询从远程数据源获取数据，而这些数据源支持 SPARQL 协议。联邦查询对于整合多个

数据源中的信息、执行跨库分析以及访问分布式数据集是非常有用的。

SERVICE 关键字后面跟着一个 IRI 指明要进行查询的 SPARQL 终端，查询引擎会向该 SPARQL 终端发送指定子查询。这个远程终端执行子查询并返回结果给原始查询引擎，然后这些结果可以与本地或其他远程数据源的数据一起进一步处理和整合。使用 SERVICE 关键字不需要远程数据源的先验整合或同步，这种联邦查询方式为知识数据的实时查询提供了灵活性。

如图 5.15 所示，这个 SPARQL 查询中的 A 部分是在当前三元组库或端点进行，SERVICE 关键字指明的 B 部分和 C 部分分别在 SPARQL 端点 1 和端点 2 上进行，这两个是通过 Web 远程访问的端点。

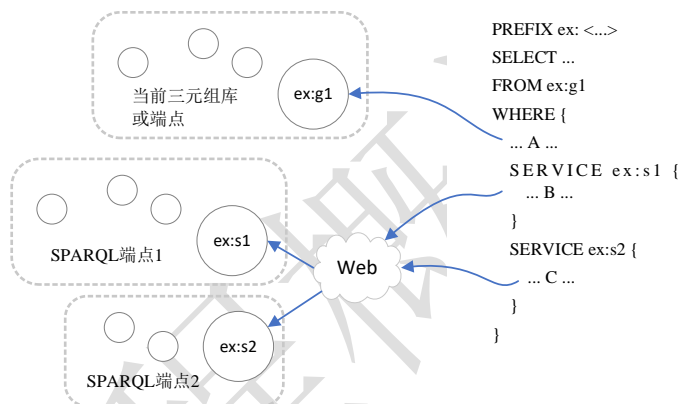


图 5.15 联邦查询示意图

SPARQL 为此专门设计了一个进行远程端点操作的网络协议，称为 SPARQL 协议 (SPARQL Protocol)，SPARQL 名称中的 P 就是协议一词的英文首字母。SPARQL 协议基于 HTTP 协议，它允许客户端通过网络向支持 SPARQL 的数据源发送查询和更新请求。这个协议支持使用 GET 和 POST HTTP 方法提交 SPARQL 查询语句，使得可以远程访问和操作 RDF 数据。返回的结果可以是 XML、JSON、CSV 等多种格式，便于不同应用程序的集成。此外，SPARQL 协议还规定了错误处理，包括如何在查询不正确或服务端遇到问题时，返回合适的 HTTP 状态码。SPARQL 协议为跨平台数据交互和集成提供了途径。

下面的查询要在 WikiData 和 DBpedia 这两个 RDF 数据集的 SPARQL 端点上进行。WikiData 是一个开放的 RDF 知识库，是支持维基百科后端的结构化数据集。它由社区成员协作编辑，用于收集与存储世界上所有知识的结构化数据，如人物、地点、事物、事件和概念等。在 WikiData 中，实体的唯一标识符以 Q 作为前缀，关系或属性的编号以 P 为前缀。

DBpedia 是从维基百科的各种语言版本中自动提取结构化信息而构建的 RDF 知识库，可以认为 DBpedia 是将维基百科中的结构化信息转换为 RDF 知识图谱格式，它是语义网和链接数据应用的一个核心资源。DBpedia 中具有与其他知识库的链接，包括 WikiData，这进一步扩展了其数据的丰富性和联通性。

下面的 SPARQL 查询就是在 WikiData 的端点中执行联邦查询，访问 DBpedia 端点，即一部分数据来自 WikiData，一部分数据来自 DBpedia。

查询 9：在 WikiData 端点上查询“赤水河”（实体标识 Q1074840）所属省份名称，以及在 DBpedia 端点上查询与 WikiData 中“赤水河”实体等价实体（owl:sameAs）的描述信息（rdfs:comment）。

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbp: <http://dbpedia.org/property/>
SELECT * WHERE {
  wd:Q1074840 rdfs:label ?river_name ;
              wdt:P131 ?province FILTER (lang(?river_name) = "zh")
  ?province rdfs:label ?prov_lab FILTER (lang(?prov_lab) = "zh")
  SERVICE <http://dbpedia.org/sparql> {
    ?dbp river owl:sameAs wd:Q1074840 ;
          rdfs:comment ?river_des .
    dbr:Battle of Chishui River dbp:place ?dbp_river .
    FILTER (lang(?river_des) = "zh")
  }
}
```

输出：


river_name	province	prov_lab	dbp_river	river_des
赤水河	 wd:Q47097	贵州省	http://dbpedia.org/resource/Chishui_River	赤水河是中国长江的一级支流，发源于云南省镇雄县芒部镇境内，流经贵州省和四川省泸州市，在四川和贵州交界的茅台镇后转为西北向，流经赤水市，向北于四川省泸州市合江县注入长江。干流全长436.5公里，流域面积2.044万平方公里。习酒镇二郎滩以上为上游，复兴场为中、下游分界。主要支流有二道河、桐梓河、古蔺河、颛水河等。平均比降1.5‰，河口流量309立方米/秒。含沙量高达0.93千克/立方米，水色赤红。*上游：海拔1000～1600米，属云贵高原。喀斯特地貌发育，河谷深切，平均比降2.2‰。水流急湍多滩。吴公滩长10公里，落差200米；*中游：流经四川盆地边缘，两岸海拔500～1000米，河谷渐宽，两岸出现台地，有暗河汇入；*下游：流经四川盆地红色丘陵区，海拔200～500米，河面开阔，平均比降0.4‰。赤水河处于长江上游珍稀特有鱼类国家级自然保护区的核心区，是长江上游支流中唯一没有闸坝阻隔、保持自然流态的河流，河域

图 5.16 查询 9 的输出结果

讲解：该查询是在 WikiData 端点上执行的，wd:Q1074840 是赤水河实体的标识。BGP 中的前三行在 WikiData 端点的数据集上进行匹配，变量?river_name 匹配名称“赤水河”，关系 wdt:P131 表示所属省份，变量?province 和?prov_lab 分别表示省份实体和其名称。SERVICE 关键字指定其后的查询在 DBpedia 端点上执行，包括变量?dbp_river 匹配与 wd:Q1074840 相同等价关系 owl:sameAs 的“赤水河”实体，变量?river_des 是该实体的文本描述，同时采用“四渡赤水”实体 dbr: Battle_of_Chishui_River 和关系“发生地点 dbp:place

对?dbp_river 匹配实体进行约束。同时采用 FILTER 约束名称语言为中文“zh”。查询 9 的输出结果如图 5.16 所示。

10. 分组与聚合

对于更复杂的数据查询和分析任务往往需要进行数据汇总工作，要用到 SPARQL 语言提供的分组、聚合与筛选功能。GROUP BY 子句用于将查询结果根据一个或多个变量进行分组。可以使用聚合函数对每组数据进行汇总，SPARQL 支持多种聚合函数，包括 COUNT()、SUM()、MIN()、MAX()和 AVG()等。

- COUNT(): 计算结果集的元素数量;
- SUM(): 计算数值总和;
- MIN()和 MAX(): 查找数值的最小值和最大值;
- AVG(): 计算数值的平均值。

HAVING 子句用于对 GROUP BY 分组后的数据进行条件过滤，排除掉不满足条件的组。

查询 10: 查找世界每个国家的人口最多的城市，要求城市人口数量超过两千万，输出国家 ID 和名称、城市 ID 和名称、人口数量。

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wikibase: <http://wikiba.se/ontology#>
PREFIX bd: <http://www.bigdata.com/rdf#>
SELECT DISTINCT ?country ?countryLabel ?city ?cityLabel ?maxPopulation
WHERE {
  ?city wdt:P31/wdt:P279* wd:Q515;
        wdt:P17 ?country;
        wdt:P1082 ?maxPopulation.

  {
    SELECT ?country (MAX(?population) AS ?maxPopulation)
    WHERE {
      ?city wdt:P31/wdt:P279* wd:Q515;
            wdt:P17 ?country;
            wdt:P1082 ?population.
    }
    GROUP BY ?country
    HAVING (MAX(?population) > 20000000)
  }
  SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE]". }
}
ORDER BY DESC (?maxPopulation)
```

输出:

country	countryLabel	city	cityLabel	maxPopulation
Q148	中华人民共和国	Q11725	重庆市	32054159
Q668	印度	Q1353	德里	26495000

图 5.17 查询 10 的输出结果

讲解：该查询从 WikiData 数据集中查找每个国家人口最多的城市，且这些城市的人口需要超过 2000 万。这个查询首先定义了一些基本的前缀，以简化 URI 的使用。使用的属性路径 “wdt:P31/wdt:P279*” 导航到变量?city 所属的类别 wd:Q515（即城市），wdt:P17 用于标识城市所属的国家，而 wdt:P1082 则用来获取城市的人口数量。此查询结构分为两个主要部分：

- 子查询：用于为每个国家选择人口最多的城市。它通过 GROUP BY ?country 将城市按国家分组，并使用 HAVING (MAX(?population) > 20000000) 来确保所选城市的最大人口数超过 2000 万。这里，聚合函数 MAX(?population) 计算每个国家所有城市中的最大人口数。
- 主查询：在子查询的基础上，主查询再次筛选这些城市，以确保所报告的城市确实是人口最多的。它通过与子查询中得到的 ?maxPopulation 进行匹配，来确保选择的是正确的城市。

最后，使用 WikiData 所提供的 “SERVICE wikibase:label” 服务，自动获取实体的标签（名称），使得输出结果不仅包括国家和城市 ID 还包括它们的名称。查询结果按城市人口数量降序排列，以便最先显示人口最多的城市。查询 10 的输出结果如图 5.16 所示。

11. 属性路径

在 SPARQL 中，属性路径（Property Paths）允许用户定义在 RDF 图中从一个节点到另一个节点的路径。这些路径可以通过各种操作符来指定多种类型的遍历方式。当想要查询与给定节点通过多个属性（或关系）间接连接的数据时就需要使用属性路径。属性路径使用一系列操作符来定义这些路径，这些操作符如表 5.4 所示。

表 5.4 SPARQL 属性路径操作符

操作符	功能	举例
/	顺序连接，表示一个属性后跟另一个属性	wdt:P802 / wdt:P106
	选择，表示一个属性或另一个	wdt:P22 wdt:P25
*	零次或多次，表示路径可以重复零次或多次	wdt:P802* / wdt:P106
+	一次或多次，表示路径至少重复一次	(wdt:P22 wdt:P25)+
?	零次或一次，表示路径可以不出现或出现一次	wdt:P802 ? wdt:P106?
^	逆向，表示反向的关系路径	(^wdt:P40)*

查询 11：查找孔子的所有学生，包括孔子直接的学生以及学生的学生、学生的学生的学生……，输出这些学生的 ID 和名称。

```
SELECT ?student ?studentLabel
WHERE {
  VALUES ?teacher {wd:Q4604}      # 指定孔子作为教师
  ?teacher wdt:P802+ ?student.    # 查找所有学生及其后代
  SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE]". }
```

输出：

student	studentLabel
Q698867	闵损
Q836861	子路
Q837573	宰予
Q840352	冉求

图 5.18 查询 11 的输出结果

讲解：整个查询的目的是找出孔子（孔子的 Wikidata ID 为 wd:Q4604）的所有直接和间接学生。这不仅包括孔子直接教过的学生，还包括那些学生的学生，依此类推。使用属性路径中的“+”操作符允许查询递归查找所有这些级别的学生。Wikidata 属性 wdt:P802 表示“学生”关系；属性路径操作符“+”代表一次或多次匹配，意味着从?teacher 起，通过 P802 属性可以递归地找到所有直接和间接的学生。这包括孔子的直接学生、学生的学生等，直到没有更多的学生为止。VALUES 关键字在 SPARQL 查询中用于指定一个或多个变量的固定值，这些值可以直接用于查询的其他部分。在本例中，设定了变量 ?teacher 的值为 wd:Q4604，这是 Wikidata 上孔子的实体 ID。查询 11 的输出结果如图 5.18 所示。

查询 12：查找孔子的所有后代（直接和间接后代），输出孔子后代的实体 ID 和名称。

```
SELECT ?descendant ?descendantLabel
WHERE {
  ?descendant (wdt:P22|wdt:P25)+ wd:Q4604.
  SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE]". }
```

输出：

descendant	descendantLabel
Q wd:Q7240164	孔鲤
Q wd:Q1147803	孔伋
Q wd:Q22812620	孔白
Q wd:Q10944385	孔求
Q wd:Q22812621	孔箕

图 5.19 查询 12 的输出结果

讲解：在此查询中的属性路径使用了选择操作符“|”，它表示“或”关系，使得路径可以通过多种属性进行选择匹配。属性路径“(wdt:P22|wdt:P25)+”的意义是：查找所有通过“父亲”或“母亲”属性能够连续追溯到孔子（wd:Q4604）的实体：

- wdt:P22 代表 Wikidata 中的“父亲”属性；
- wdt:P25 代表 Wikidata 中的“母亲”属性；
- + 表示一次或多次匹配，意在查找不仅是孔子的直接子女，还包括孔子的孙子辈、曾孙辈等所有通过父母关系直接或间接连接到孔子的后代。

查询 12 的输出结果如图 5.19 所示。

12. 处理查询结果

SPARQL 具有多种用于处理查询结果的特性，使得用户能够更有效地控制和优化输出数据。特别是在处理像 Wikidata 这样的大型数据集时，ORDER BY、LIMIT 和 OFFSET 子句的使用尤为重要。

- ORDER BY 子句用于对查询结果进行排序。它可以根据一个或多个条件对结果进行升序（ASC）或降序（DESC）排序。排序可以基于数字、字符串、日期等多种数据类型。
- LIMIT 子句用来限制 SPARQL 查询返回的结果数量。这在查询可能返回大量数据时非常有用，可以帮助减轻服务器的负担，并提高查询响应时间。
- OFFSET 子句用于指定查询结果开始返回的偏移量。这常与 LIMIT 结合使用，实现分页功能。

这三个子句经常结合使用，以更精确地控制查询结果的输出。例如，如果需要对大量数据进行分页处理，可以按照特定的顺序显示每页固定数量的结果。

查询 13：从 Wikidata 查询出名人的名字和出生日期，并且有以下需求：

- 结果需要按出生日期降序排列，即从最年轻的物理学家开始；

- 仅需要查看 10 条结果；
- 跳过最前面的 10 条结果，查看之后的结果。

```
SELECT ?person ?personLabel ?birthDate
WHERE {
    ?person wdt:P106 wd:Q169470;    # P106 代表职业, Q169470 代表“物理学家”
        wdt:P569 ?birthDate.    # P569 代表“出生日期”
SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }
}
ORDER BY DESC (?birthDate)
LIMIT 10
OFFSET 10
```

输出：

person	personLabel	birthDate
Q wd:Q519623	André Henri Mirau	1 January 2000
Q wd:Q523264	André Voros	1 January 2000
Q wd:Q528491	Deborah Berebichez	1 January 2000
Q wd:Q1358957	Ernst Kircher	1 January 2000
Q wd:Q1599157	Heinrich Wahl	1 January 2000

图 5.20 查询 13 的输出结果

讲解：这个查询从 Wikidata 数据集中查找物理学家的名字和出生日期，并按出生日期升序排列，跳过前十条结果，显示接下来的十条结果。**ORDER BY DESE(?birthDate)**表示根据出生日期降序排序结果，即越年轻的物理学家越靠前；**LIMIT 10** 表示返回最多 10 条结果；**OFFSET 10** 表示从结果的第 11 条开始返回。查询 13 的输出结果如图 5.20 所示。

13. DESCRIBE 与 ASK 查询

在 SPARQL 中，除了常见的 SELECT 查询用于进行知识图谱的模式匹配之外，还有其他的查询类型。**DESCRIBE** 查询用于获取关于一个或多个资源的全部或部分 **RDF** 图。这种查询类型返回一个或多个资源的所有属性及其相应的值，这有助于快速了解一个实体的所有可用数据。假设想从 Wikidata 获取关于孔子的全部信息，可以使用如下的 **DESCRIBE** 查询：

查询 14：从 Wikidata 获取关于孔子的全部信息。

```
DESCRIBE wd:Q4604
```

输出：

Table

?

2104条, 15毫秒





subject	predicate	object
 wd:Q4604	rdfs:label	Confucius
 wd:Q4604	rdfs:label	Kūng-cū
 wd:Q4604	rdfs:label	Конфуци
 wd:Q4604	rdfs:label	Confucio

图 5.21 查询 14 的输出结果

讲解：使用 DESCRIBE 查询可以获取关于指定实体的所有可用信息。这种查询对于了解某实体的属性和与之关联的其他实体非常有用。这个查询将返回关于孔子的所有 RDF 数据，包括生卒日期、职业、影响以及与之关联的实体等信息。查询 14 的输出结果如图 5.21 所示。

ASK 查询用来验证某个特定的事实是否为真，例如，我们可以检查孔子是否被标记为哲学家。

查询 15：在 Wikidata 中检查孔子是否被标记为哲学家。

```
ASK {
  wd:Q4604 wdt:P106 wd:Q4964182.
}
```

输出：true（真）

讲解：在这个查询中，wdt:P106 代表职业属性，wd:Q4964182 是哲学家的实体 ID。如果孔子的职业包括哲学家，这个查询将返回 true，否则返回 false。

14. 构建数据

SPARQL 还提供从现有的 RDF 数据中创建新的 RDF 图的机制，即使用 CONSTRUCT 语句。与 SELECT 查询返回变量的值不同，CONSTRUCT 查询生成并返回一个或多个三元组，从而实现数据的转换和重构。

CONSTRUCT 语句的基本语法结构如下：

```
CONSTRUCT {
  三元组模式
}
WHERE {
  查询模式
}
```

在 CONSTRUCT 部分，定义希望生成的新 RDF 三元组结构；在 WHERE 部分，指定

用于匹配数据的模式。CONSTRUCT 语句的执行流程是：首先根据 WHERE 块的条件从知识图谱中匹配数据，然后将匹配结果填充到 CONSTRUCT 块中，生成新的 RDF 图。

查询 16：构建 RDF 图，展示孔子与他的弟子之间的师生关系。

```
CONSTRUCT {
  ?teacher <http://example.org/hasStudent> ?student .
  ?student <http://example.org/hasTeacher> ?teacher .
  ?teacher rdfs:label ?teacherLabel .
  ?student rdfs:label ?studentLabel .
}
WHERE {
  # 查询孔子的学生
  ?student wdt:P1066 ?teacher .
  BIND(wd:Q4604 AS ?teacher) .

  # 查询学生的标签
  ?student rdfs:label ?studentLabel .
  FILTER(LANG(?studentLabel) = "zh") # 只选取中文标签

  # 查询教师的标签
  ?teacher rdfs:label ?teacherLabel .
  FILTER(LANG(?teacherLabel) = "zh") # 只选取中文标签
}
```

输出：

subject	predicate	object
Q wd:Q4604	<http://example.org/hasStudent>	Q wd:Q10911249
Q wd:Q10911249	<http://example.org/hasTeacher>	Q wd:Q4604
Q wd:Q4604	rdfs:label	孔子
Q wd:Q10911249	rdfs:label	原宪
Q wd:Q4604	<http://example.org/hasStudent>	Q wd:Q840352
Q wd:Q840352	<http://example.org/hasTeacher>	Q wd:Q4604
Q wd:Q840352	rdfs:label	冉求

图 5.22 查询 16 的输出结果

讲解：在 WHERE 部分，匹配孔子的所有弟子，并获取他们的中文名称；在 CONSTRUCT 部分，构建了孔子与其每个弟子之间的双向师生关系三元组，用 [<http://example.org/hasStudent>](#) 表示教师有学生关系，用 [<http://example.org/hasTeacher>](#) 表示学生有教师关系。通过该 CONSTRUCT 查询，我们成功构造了一个新的知识图谱，表示孔子作为教师和学生的师生关系。查询 16 的输出结果如图 5.22 所示。

15. 数据更新

SPARQL 不仅是一种查询语言，也是一种更新语言，支持对知识图谱进行增删改操作。SPARQL Update 是 SPARQL 1.1 规范的一部分，提供了一组用于操作 RDF 图的语句。这些语句包括 INSERT、DELETE、LOAD、CLEAR、CREATE、DROP、COPY、MOVE 和 ADD。通过 SPARQL Update，用户可以动态地修改知识图谱的内容，使其能够适应不断变化的知识需求。

SPARQL Update 的核心语法包括以下几种主要操作：

- INSERT：向知识图谱中添加新的三元组。
- DELETE：从知识图谱中删除指定的三元组。
- DELETE/INSERT：先删除旧的三元组，再插入新的三元组（用于更新数据）。
- LOAD：将一个 RDF 图装载到 RDF 数据库。
- CLEAR：清空一个 RDF 图。
- CREATE：在 RDF 数据库中创建一个新的 RDF 图。
- DROP：将一个 RDF 图从 RDF 数据库中删除。
- COPY：将一个 RDF 图的内容复制到另一个 RDF 图。
- MOVE：将一个 RDF 图的内容移动到另一个 RDF 图。
- ADD：向一个 RDF 图中添加内容。

每个操作都可以作用于默认图或命名图。其基本语法格式如下：

```
# 插入数据
INSERT DATA {
  GRAPH <graph-uri> {
    # 新增的三元组
    <subject> <predicate> <object> .
  }
}

# 删除数据
DELETE DATA {
  GRAPH <graph-uri> {
    # 要删除的三元组
    <subject> <predicate> <object> .
  }
}

# 更新数据 (DELETE/INSERT)
```



```
DELETE {
  # 要删除的旧三元组
  <subject> <predicate> <old_object> .
}
INSERT {
  # 要新增的三元组
  <subject> <predicate> <new_object> .
}
WHERE {
  # 匹配条件 (可选)
  <subject> <predicate> <old_object> .
}
```

本节我们还是使用 5.1.1 节得到的“四渡赤水”RDF 图作为知识图谱数据，通过例子展示 SPARQL Update 语言的基本用法。

查询 17：新增数据。

我们发现赤水河的长度单位 `geo:kilometers` 缺少标签信息，需要补充单位的名称为“公里”。

```
INSERT DATA {
  geo:kilometers rdfs:label "公里" .
}
```

讲解：该 INSERT DATA 语句执行后，RDF 图中会增加一条三元组 “`geo:kilometers rdfs:label "公里"`”。

查询 18：删除数据。

假设我们认为知识图谱中三元组 “`geo:river geo:locatedIn geo:Sichuan`” 需要删除。

```
DELETE DATA {
  geo:river geo:locatedIn geo:sichuan .
}
```

讲解：该 DELETE DATA 语句执行后，三元组 “`geo:river geo:locatedIn geo:Sichuan`” 会从 RDF 图中被删除掉。

查询 19：修改数据。

赤水河的长度目前记录为 436.5，但假设实际长度为 440.0 公里，我们需要修正这个错误。

```
DELETE {
  geo:river geo:length ?x .
  ?x rdf:value "436.5"^^xsd:decimal .
}
INSERT {
  geo:river geo:length ?x .
}
```

```

    ?x rdf:value "440.0"^^xsd:decimal .
}
WHERE {
    geo:river geo:length ?x .
    ?x rdf:value "436.5"^^xsd:decimal .
}

```

讲解：修改数据需要使用 DELETE/INSERT/WHERE 这个语句结构，首先由 WHERE 匹配要更新的三元组。由于 RDF 图中含有空节点，而空节点与空节点均为匿名实体，是不能进行匹配的，因而在 WHERE 中要引入变量 ?x 进行匹配，不能使用空节点来匹配。DELETE 中与 WHERE 中是相同的匹配模式，表示将 RDF 图中的 “geo:river geo:length [rdf:value "436.5"^^xsd:decimal]” 删除，INSERT 中的匹配模式表示将 “geo:river geo:length [rdf:value "440.0"^^xsd:decimal]” 插入 RDF 图中，从而实现 RDF 数据的修改更新。

SPARQL Update 语言的其他语句较为简单，不再赘述。读者可以查询相应文档，自行学习。

5.2.2 Cypher 语言

Cypher 是面向属性图数据库的声明式查询语言，最初由图数据库 Neo4j 提出并实现。与 SPARQL 类似，Cypher 允许用户通过简洁的语法来“声明”想要查询的数据，而不必显式地指定完整的查询执行路径或算法，从而减轻了进行图查询的编程负担。随着知识图谱在工业界与学术界的广泛应用，Cypher 已逐渐成为属性图数据库的事实标准语言之一。本节将以图 5.4（“四渡赤水”示例知识图谱的属性图数据模型）为例，介绍 Cypher 语言在知识查询方面的主要功能与用法。下文的示例默认均基于属性图数据库系统环境（如 Neo4j）。

1. Cypher 查询的语法结构

Cypher 查询的基本语法结构为：

```
<关键字> <图模式> <条件> <返回结果> <其他操作>
```

其中的这些部分是指：

- 关键字：指定查询的操作类型，例如 MATCH、CREATE、DELETE 等。
- 图模式：描述图中的节点和关系的结构。
- 条件：使用 WHERE 子句定义过滤条件。
- 返回：使用 RETURN 或其他关键字指定查询结果。
- 其他操作：包括排序、分页、聚合等。

Cypher 查询的基本执行过程是：在属性图中按照 MATCH 描述的模式进行匹配，把

匹配结果绑定到变量上，然后通过 RETURN 子句输出查询结果。必要时可在 MATCH 中用 WHERE 进行筛选、在 RETURN 中进行聚合或排序等。

2. Cypher 的 ASCII 艺术语法风格

Cypher 语言的一大特点是其具备的 ASCII 艺术（ASCII art）语法风格。这种语法风格通过简单的文本符号（如圆括号、方括号和箭头）来直观地表示图模型中的节点和关系，使查询语句在结构上清晰易读，非常适合知识图谱上数据的直观表达和操作。

Cypher 语言使用以下符号来表示属性图模型的核心组件：

(1) 节点（Node）：使用圆括号 () 表示。

- 示例：(n) 表示一个节点变量 n。
- 如果节点有标签，可以在括号内添加标签：(n:Person) 表示一个标签为 Person 的节点。

(2) 关系（Relationship）：使用中括号 [] 表示，并通过箭头 --> 或 <-- 指定方向。

- 示例：()-[r]->>() 表示一个从左侧节点到右侧节点的有向关系，变量为 r。
- 指定关系类型：()-[:location]->>() 表示类型为 location 的关系。

(3) 属性（Properties）：使用花括号 {} 表示属性和值。

- 示例：(n {name: "四渡赤水"}) 表示一个带有 name 属性的节点。

这种符号化的表达方式使 Cypher 更接近图的视觉化表达，使查询代码本身像一幅“文本图”，便于用户理解复杂的图结构。图 5.23 给出了 Cypher 中一个 MATCH 基本匹配查询的结构示意。对应于图 5.4 中的节点 v_4 和 v_6 以及两者之间的边 e_5 ，可以看出，MATCH 查询中用于匹配节点和边的 ASCII 艺术语法。

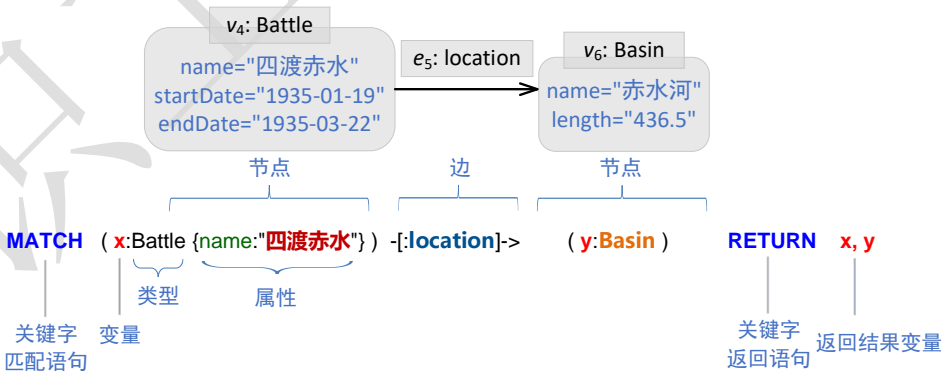


图 5.23 Cypher 语言的 ASCII 艺术语法风格

3. 基本匹配查询

与 SPARQL 的三元组模式查询类似，Cypher 的最基本查询形式就是通过 MATCH 子句来描述一条或多条图模式，然后将匹配成功的结果返回。

下面对基本匹配查询就行举例讲解，首先是匹配一个节点的查询。

查询 20：查询所有战役（Battle）实体的名称。

```
MATCH (b:Battle)
RETURN b.name AS battleName
```

输出：

battleName
"四渡赤水"

图 5.24 查询 20 的输出结果

讲解：

- MATCH 子句中，圆括号“()”表示节点；“b”为该节点的标识符（变量）；“:Battle”指定节点具有标签（label）“Battle”。
- RETURN 子句中，使用“b.name AS battleName”把节点 b 的属性 name 作为输出列命名为“battleName”。
- 我们的图中只有一个战役节点“四渡赤水”（即 v_4 ），查询结果返回一行；查询结果如图 5.24 所示。

上面的查询结果为节点属性，以表格形式返回，下面的查询将匹配一条边，并以图的形式返回结果。

查询 21：查询所有战役（Battle）与其发生的流域（Basin）。

```
MATCH (b:Battle)-[:LOCATION]->(r:Basin)
RETURN b, r
```

输出：

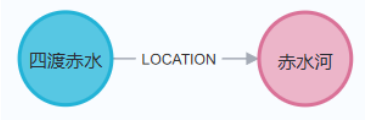


图 5.25 查询 21 的输出结果

讲解：

- (b:Battle) 表示具有标签 “Battle” 的节点。

- (r:Basin) 表示具有标签 “Basin” 的节点。
- -[:LOCATION]-> 表示从 “Battle” 节点到 “Basin” 节点的 “LOCATION” 关系。
- RETURN b,r 将匹配到的所有 “战役” 节点和对应的 “流域” 节点一起返回；查询结果如图 5.25 所示。

查询 22: 查询人物 (Person) 指挥的战役 (Battle), 战役所属的战争 (Campaign) 以及战争的参战部队 (Organization)。

```
MATCH (p:Person)-[:COMMAND]-(b:Battle)-[:PART_OF]->(c:Campaign)<-[:EXECUTE]-(o)
RETURN p, b, c, o
```

输出:

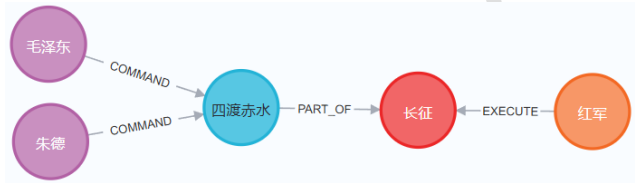


图 5.26 查询 22 的输出结果

讲解：这个例子给出了一个较为复杂的图模式匹配查询，共涉及到 COMMAND、PART_OF 与 EXECUTE 三种关系类型；通过 ASCII 艺术风格的语法实现了节点与边组织的路径导航，既有正向边（如-[:PART_OF]->），又有反向边（如<-[:EXECUTE]-），还有无向边（如-[:COMMAND]-）；体现出 Cypher 语言对图匹配查询的支持。查询结果如图 5.26 所示。

4. 使用条件过滤

在 Cypher 中可使用 WHERE 子句对 MATCH 的匹配结果进行条件过滤，作用类似于 SPARQL 中的 FILTER。

查询 23: 查询在 1935 年开始的战役名称及开始日期。

```
MATCH (b:Battle)
WHERE b.startDate >= "1935-01-01" AND b.startDate < "1936-01-01"
RETURN b.name AS battleName, b.startDate AS start
```

输出:

battleName	start
"四渡赤水"	"1935-01-19"

图 5.27 查询 23 的输出结果

讲解：

- 使用 WHERE 子句对 b.startDate 进行条件判断：大于等于 1935-01-01 且小于 1936-01-01（即在 1935 年开始）。
- Cypher 的逻辑运算符为 AND、OR、NOT，比较运算符包括 <、<=、>、>=、=、<> 等，与 SPARQL 基本对应。
- 查询结果如图 5.27 所示。

4. 使用可选匹配

与 SPARQL 中的 OPTIONAL 类似，Cypher 提供了 OPTIONAL MATCH 子句，用于在匹配模式时允许部分缺失。如果可选匹配的部分无法匹配成功，Cypher 会返回 null（空值）作为对应变量的值，而不会直接丢弃该行数据。

查询 24：查询指挥“四渡赤水”战役的将领及其角色。

这里我们还要求可选地匹配将领到其他实体的 EXECUTE 关系，并将结果按将领姓名降序排序输出。

```
MATCH (b:Battle {name: "四渡赤水"})
MATCH (p:Person)-[c:COMMAND]->(b)
OPTIONAL MATCH (p)-[e:EXECUTE]->()
RETURN p.name AS commander, c.role AS role, e
ORDER BY commander DESC
```

输出：

commander	role	e
"毛泽东"	"战略指挥"	null
"朱德"	"军事领导"	null

图 5.28 查询 24 的输出结果

讲解：

- OPTIONAL MATCH (p)-[e:EXECUTE]->() 是一个可选匹配操作，用于查找人物 p 是否与其他某节点存在 EXECUTE 关系。如果 p 没有具有 EXECUTE 关系的节点，那么这一可选匹配并不会过滤掉该人物，而只是将 e 置为空值 null。
- ORDER BY commander DESC 表示按将领姓名 commander 的值降序排序输出。
- 查询结果如图 5.28 所示，可见两个匹配的人物节点均没有 EXECUTE 关系，e 这

列的值为 `null`。

- 如果将 `OPTIONAL` 关键字去掉，再次执行该查询，将没有任何结果输出。

5. 模式并集查询

使用 `UNION` 关键字可以将多个 `MATCH` 的匹配结果按并集操作合在一起。

查询 25：查询所有类型为 `Battle` 或 `Campaign` 的实体并返回其名称。

```
MATCH (b:Battle)
RETURN b.name AS entityName
UNION
MATCH (c:Campaign)
RETURN c.name AS entityName
```

输出：

entityName
"四渡赤水"
"长征"

图 5.29 查询 25 的输出结果

讲解：这里 `UNION` 将两个 `MATCH` 查询的结果求并集，再放到同一输出结果中。对比 SPARQL 语言的查询 5。查询结果如图 5.29 所示。

6. 使用函数

Cypher 语言内置了多种函数，可简化对属性图中数据的处理与计算。常见函数涵盖字符串处理、数值计算、日期时间、列表集合、路径与关系等多个方面。

(1) 字符串函数：

- `toLowerCase() / toUpper()`：将文本转换为小写/大写。
- `substring(text, start, length)`：从 `text` 的指定位置 `start` 起，截取长度为 `length` 的子字符串。

(2) 数值函数：

- `ceil(expression) / floor(expression)`：分别将表达式结果向上/向下取整。
- `sqrt(expression)`：计算表达式结果的平方根。

(3) 日期时间函数：

- `date()`：创建日期对象或转换字符串为日期。

- `time()`: 获得当前时间, 不含日期。

(4) 列表与集合函数:

- `size(list)`: 返回列表项的数量。
- `head(list)` / `tail(list)` / `last(list)`: 分别返回列表的第一个、除第一个以外所有元素、以及最后一个元素。
- `apoc.coll.*`: Neo4j 的 APOC 库提供了大量的集合函数, 如去重、排序、合并、交集等。

(5) 路径与关系函数:

- `length(path)`: 返回给定路径中关系的数量。
- `relationships(path)`: 返回路径中的所有关系列表

下面给出使用字符串函数和日期时间函数的一个查询示例。

查询 26: 查询“四渡赤水”战役的持续天数。

```
MATCH (b:Battle { name: "四渡赤水" })
RETURN
    b.name AS originalName,
    substring(b.name, 0, 2) AS shortName,
    duration.inDays(date(b.startDate), date(b.endDate)).days AS daysElapsed
```

输出:

originalName	shortName	daysElapsed
"四渡赤水"	"四渡"	62

图 5.30 查询 26 的输出结果

讲解: 输出结果第 1 列 `originalName` 为战役名称 (即 `b.name`); 第 2 列 `shortName` 使用字符串函数 `substring` 对名称的前 2 个字符做截取; 第 3 列 `daysElapsed` 是利用日期时间函数 `duration.inDays` 计算该战役从开始到结束的相隔天数, 其中使用 `date` 函数将战役开始日期 (`b.startDate`) 和结束日期 (`b.endDate`) 由字符串类型转化为日期类型。查询结果如图 5.30 所示。

7. 聚合与分组

Cypher 中可以使用聚合函数 (包括 `COUNT`、`SUM`、`AVG`、`MIN`、`MAX` 等) 对匹配结果进行分组统计。与 SPARQL 不同的是, Cypher 并未使用 `GROUP BY` 子句, 而是在 `MATCH`、`RETURN` 中结合 `WITH` 子句来实现“分组”与“聚合”的功能。

查询 27：统计每个省份（Province）包含的战役（Battle）数量。

```
MATCH (b:Battle)-[:LOCATION]->(r:Basin)
MATCH (r)-[:LOCATED_IN]->(p:Province)
WITH p, COUNT(b) AS battleCount
RETURN p.name AS province, battleCount
ORDER BY battleCount DESC
```

输出：

province	battleCount
"贵州省"	1
"四川省"	1

图 5.31 查询 27 的输出结果

讲解：首先通过两条 MATCH 语句将战役节点与省份节点匹配出来；使用 WITH p, COUNT(b) AS battleCount 将同一省份节点 p 分组，并统计其对应的战役数量；在 RETURN 子句中输出省份名称与战役数量，并按数量从大到小排序。查询结果如图 5.31 所示。

8. 更新数据

Cypher 同样支持对属性图中的节点、关系及属性进行插入、删除和更新操作，其最常用命令包括 CREATE、MERGE、SET、DELETE 等。

- CREATE：在图中新建节点或关系。
- MERGE：在图中查找是否已有对应节点或关系，若不存在则新建。
- SET：对匹配到的节点或关系设置属性值或添加标签。
- DELETE：删除匹配到的节点或关系（注意若删除节点，需先删除其所有关系）。

查询 28：更新赤水河的长度数值。

```
MATCH (r:Basin {name: "赤水河"})
SET r.length = 440.0
RETURN r.name AS basinName, r.length AS newLength
```

讲解：MATCH 匹配到名称为“赤水河”的流域节点 r；SET r.length = 440.0 将其 length 属性值更新为 440.0；RETURN 显示更新结果。

9. 处理查询结果

Cypher 中可使用 ORDER BY、LIMIT、SKIP 等关键字对查询结果进一步处理和分页。

- **ORDER BY:** 指定按某些条件进行排序，可指定升序 ASC 或降序 DESC。
- **LIMIT:** 限制返回结果数量。
- **SKIP:** 跳过指定数量的结果，可与 LIMIT 一同用于查询结果数据的分页。

查询 29：按开始日期降序列出所有战役名及开始日期，只返回第 2 页的 5 条结果。

```
MATCH (b:Battle)
RETURN b.name AS battleName, b.startDate AS start
ORDER BY b.startDate DESC
SKIP 5
LIMIT 5
```

讲解：ORDER BY b.startDate DESC 表示根据战役开始日期降序排列；SKIP 5 跳过前面 5 条结果；LIMIT 5 仅返回后续的 5 条结果；SKIP 和 LIMIT 结合使用可以实现数据分页功能。

10. 路径查询

在 Cypher 中，使用路径查询能够帮助发现节点之间的连接方式与结构关系。一个“路径 (Path)”是由一系列相互连接的节点和边组成的遍历轨迹。例如，一个路径可以是：

```
(p1) -[:RELATION1]-> (p2) -[:RELATION2]-> (p3)
```

其中 (p1)、(p2)、(p3) 为节点，[:RELATION1]和[:RELATION2]表示它们之间的关系或边。

实际上，在前面的 Cypher 图模式匹配查询中已经涉及到了路径查询（如查询 22）。但是这些查询均为固定长度的路径，Cypher 提供了修饰符语法来支持可变长度路径的匹配。在 Cypher 中，这种语法结构称为“量化关系”（quantified relationship），就是在关系后面加上量词修饰，例如，“-[:R]->{3,5}”表示关系 R 要连续匹配至少 3 次至多 5 次。表 5.5 给出了 Cypher 中路径查询的量词修饰符语法及其含义。

表 5.5 路径查询的量词修饰符

修饰符语法	含义
{m, n}	匹配 m 到 n 次
{1,} 或 +	匹配 1 到多次
{0,} 或 *	匹配 0 到多次
{n, n} 或 {n}	匹配 n 次
{m,}	匹配 m 到多次
{, n} 或 {0, n}	匹配 0 到 n 次

查询 30：查询从“四渡赤水”战役节点出发至少经 3 条有向边才能到达的路径。

```
MATCH path = (crossing:Battle)-[:>]{3,}(other)
WHERE crossing.name = "四渡赤水"
RETURN path
```

输出：

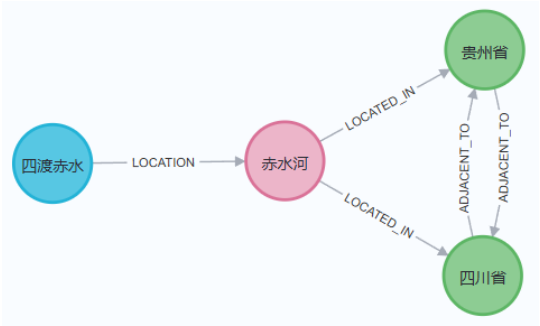


图 5.32 查询 30 的输出结果

讲解：“(crossing:Battle)-[]->{3,}(other)”表示从 crossing 节点出发经过至少 3 条边能到达的节点构成的路径，path 是代表该路径的变量。通过 WHERE 指明 crossing 节点为“四渡赤水”战役。其中，量词修饰符“{3,}”表示匹配 3 条至多条边。表 5.6 给出了该查询表格形式输出结果，其中显示的是符合查询条件的路径变量 path 的值。可以看到，结果中一共有 4 条匹配成功的路径，第 1、3 行中的路径长度为 3，第 2、4 行的路径长度为 4，且有重复的节点出现，但没有重复的边。这是因为 Cypher 默认的路径匹配语义是“无重复边”（no-repeated edge），即匹配的结果路径中不能出现重复的边，但可以出现重复的点。该查询的图形结果如图 5.32 所示。

表 5.6 查询 30 的表格形式输出结果

path
(:Battle {endDate: "1935-03-22",name: "四渡赤水",startDate: "1935-01-19"})-[:LOCATION]->(:Basin {name: "赤水河",length: 440.0})-[:LOCATED_IN]->(:Province {name: "贵州省"})-[:ADJACENT_TO]->(:Province {name: "四川省"})
(:Battle {endDate: "1935-03-22",name: "四渡赤水",startDate: "1935-01-19"})-[:LOCATION]->(:Basin {name: "赤水河",length: 440.0})-[:LOCATED_IN]->(:Province {name: "贵州省"})-[:ADJACENT_TO]->(:Province {name: "四川省"})-[:ADJACENT_TO]->(:Province {name: "贵州省"})
(:Battle {endDate: "1935-03-22",name: "四渡赤水",startDate: "1935-01-19"})-[:LOCATION]->(:Basin {name: "赤水河",length: 440.0})-[:LOCATED_IN]->(:Province {name: "四川省"})-[:ADJACENT_TO]->(:Province {name: "贵州省"})
(:Battle {endDate: "1935-03-22",name: "四渡赤水",startDate: "1935-01-19"})-[:LOCATION]->(:Basin {name: "赤水河",length: 440.0})-[:LOCATED_IN]->(:Province {name: "四川省"})-[:ADJACENT_TO]->(:Province {name: "贵州省"})-[:ADJACENT_TO]->(:Province {name: "四川省"})

查询 31：查询从“四渡赤水”战役节点出发至少经 1 条有向边能够到达的节点。

```
MATCH (start:Battle {name:"四渡赤水"})-[]->+(end)
RETURN start, end
```

输出：



图 5.33 查询 31 的输出结果

讲解：“-[]->+”表示从起始节点出发经过任意有向边 1 次至多次。由于对匹配次数放宽了要求，“长征”节点也被匹配到了结果中。查询结果如图 5.33 所示。

查询 32：查询从“四渡赤水”战役节点出发至少经 3 条有向边能够到达的最短路径。

```
MATCH path = SHORTEST 1 (crossing:Battle)-[]->{3,}(other)
WHERE crossing.name = "四渡赤水"
RETURN path
```

输出：

表 5.7 查询 32 的表格形式输出结果

path
(:Battle {endDate: "1935-03-22",name: "四渡赤水",startDate: "1935-01-19"})-[:LOCATION]->(:Basin {name: "赤水河",length: 440.0})-[:LOCATED_IN]->(:Province {name: "贵州省"})-[:ADJACENT_TO]->(:Province {name: "四川省"})
(:Battle {endDate: "1935-03-22",name: "四渡赤水",startDate: "1935-01-19"})-[:LOCATION]->(:Basin {name: "赤水河",length: 440.0})-[:LOCATED_IN]->(:Province {name: "四川省"})-[:ADJACENT_TO]->(:Province {name: "贵州省"})

讲解：该查询是在查询 30 的基础上增加了关键字“SHORTEST 1”表示结果中返回两个节点之间的最短路径，如果这样的最短路径有多条，则只返回其中的 1 条。从表 5.7 中可以看出，查询结果比表 5.6 中的少了两行，因为那两行结果并不是起点和终点之间的最短路径。

11. Cypher 与 GQL 语言

Cypher 语言最早由 Neo4j 公司于 2011 年左右提出，此后 Neo4j 发起了开源项目 openCypher，旨在对 Cypher 进行标准化工作；但与 W3C 制定的 RDF 图标准查询语言 SPARQL 相比，属性图查询语言的标准化起步较晚。Cypher 虽然是属性图的主流查询语言，仍存在着多种其他的声明式查询语言，包括：SQL/PGQ、PGQL、G-CORE 和 GSQL 等，以及过程式查询语言 Gremlin，用于属性图查询。

为此，2019 年开始，国际标准化组织（ISO）发起制定统一的属性图查询语言国际标

准，即 GQL（Graph Query Language）；2024 年 4 月，GQL 国际标准（ISO/IEC 39075）正式发布，是继 SQL 之后，ISO 标准化组织设计的第二个数据查询语言标准。GQL 旨在统一属性图数据查询语言的语法与语义，支持属性图数据的创建、修改、查询、分析和管理，解决跨属性图数据库系统的互操作性问题。GQL 在设计上充分吸收 Cypher 的语言实践，二者在基础语法上高度兼容，GQL 通过标准化实现更广泛的应用场景。Cypher 与 GQL 基本实现了向上兼容，大部分 Cypher 查询可直接迁移至 GQL 执行环境，同时，Neo4j 宣布支持 GQL 标准，未来 Cypher 将作为 GQL 的子集或方言存在。

5.2.3 Gremlin 语言

Gremlin 是 Apache TinkerPop 提供的面向属性图数据库与图计算框架的查询语言。Gremlin 的核心理念是通过“图遍历”（graph traversal）来进行图数据的导航与操作。Gremlin 的设计定位为过程式语言，用户通过定义一系列遍历步骤（traversal step）来描述操作流程，从而实现对属性图数据库的查询、分析和更新。

与 SPARQL 和 Cypher 这类声明式语言不同，Gremlin 强调操作路径的显式定义，类似于编程语言中的函数式编程。Gremlin 的定位是图遍历语言，其执行机制好比一个人置身于图数据中，沿着有向边从一个节点到另一个节点进行游走。其优点在于灵活性和可扩展性，适合复杂的路径导航和多重条件的动态查询；但同时也要求用户对图的结构和遍历路径有较清晰的认知。

1. Gremlin 查询的语法结构

Gremlin 查询是一系列遍历步骤（Step）的组合，每个步骤从图中某个起点开始，执行特定的操作，并将结果作为输入传递给下一个步骤。其基本语法结构如下：

```
g.<step>.<step>.<step>...
```

其中，g 是图的遍历源（graph traversal source），<step> 是图遍历的具体操作。遍历步骤分为以下几类：

- 起点步骤（source step）：定义遍历的起始点，例如 V()（从所有节点开始遍历）或 E()（从所有关系开始遍历）。
- 过滤步骤（filter step）：根据条件筛选节点或关系，例如 has()、where()。
- 投影步骤（projection step）：提取节点或关系的属性，例如 values()、valueMap()。
- 路径操作步骤（path step）：定义和操作路径，例如 out()、in()、both()。
- 分组与聚合步骤（group and aggregate step）：对结果进行分组或统计，例如 group()、count()。

- 结果操作步骤 (result step): 输出或排序结果, 例如 `toList()`、`order()`。

Gremlin 的查询过程通常由一系列连贯的步骤组成, 形成一个管道式的操作流。

2. 基本节点与关系查询

Gremlin 的基本查询从节点或关系开始, 用户通过 `V()` 或 `E()` 步骤遍历图中的所有节点或关系。

查询 33: 查询所有节点及其标签。

```
g.V().label().toList()
```

输出:

```
Person
Province
Person
Organization
Battle
Campaign
Basin
Province
```

讲解: `g.V()`代表图中的所有节点; `label()`返回每个节点的标签; `toList()`将结果转换为列表输出。

查询 34: 查询所有 **Battle** 类型节点的名称。

```
g.V().hasLabel("Battle").values("name")
```

输出:

```
四渡赤水
```

讲解: `hasLabel("Battle")`筛选出节点标签为 **Battle** 的节点; `values("name")` 返回节点的 **name** 属性值。

查询 34: 查询所有 **COMMAND** 类型的关系。

```
g.E().hasLabel("COMMAND")
```

输出:

```
e[18][0-COMMAND->6]
e[19][2-COMMAND->6]
```

讲解: `g.E()`代表图中的所有关系; `hasLabel("COMMAND")`筛选出类型为 **COMMAND** 的关系; 注意, 输出结果中的节点与关系的编号是数据库自动赋予的。

3. 关系与路径查询

Gremlin 提供了丰富的路径操作步骤，可以方便地表达节点间的关系和路径查询。

查询 35：查询“四渡赤水”战役的发生地点。

```
g.V().has("name", "四渡赤水").out("LOCATION").values("name")
```

输出：

```
赤水河
```

讲解：has("name", "四渡赤水") 定位到 name 属性值为“四渡赤水”的节点；out("LOCATION")表示沿着 LOCATION 关系从“四渡赤水”节点向外遍历，找到目标节点；注意，out 函数的导航方向是沿当前节点的出边向外。

查询 36：查询“四渡赤水”战役的指挥官及其角色。

```
g.V().has("name", "四渡赤水").inE("COMMAND").as("e").outV().as("p")
.select("p", "e").by("name").by(values("role"))
```

输出：

```
[p:毛泽东,e:战略指挥]
[p:朱德,e:军事领导]
```

讲解：inE("COMMAND")表示遍历指向“四渡赤水”战役的 COMMAND 类型的入边；as("e")将该边标记为变量 e；outV()获取边的起始节点（即指挥官节点）；as("p")将指挥官节点标记为变量 p；select("p", "e").by("name").by(values("role"))输出指挥官的名称和对应边的角色属性。

查询 37：查询从“四渡赤水”战役出发的所有路径。

```
g.V().has("name", "四渡赤水").repeat(out()).emit().path().by("name")
```

输出：

```
[四渡赤水, 赤水河]
[四渡赤水, 长征]
[四渡赤水, 赤水河, 贵州省]
[四渡赤水, 赤水河, 四川省]
[四渡赤水, 赤水河, 贵州省, 四川省]
[四渡赤水, 赤水河, 四川省, 贵州省]
...
```

讲解：

- repeat(out())定义了一个递归遍历，即从当前节点出发，沿所有出边（out()）重复遍历；这个步骤会无限制地沿出边递归，直到没有可用的出边为止；对于存在环的图数据，会产生无限循环输出。

- `emit()`表示每次递归都会立即输出当前路径的结果；由于 `emit()`会在每次递归时输出结果，因此即便路径还未到达终点，也会将中间路径输出。
- `path()`收集从起点到当前节点的完整路径；每一条路径都是一个列表，包含路径中所有节点的值。
- `by("name")`指定 `path()`中的每个节点提取 `name` 属性值，输出的路径中只显示节点的 `name` 属性。

在“四渡赤水”知识图谱中，赤水河的流域节点通过 `LOCATED_IN` 关系同时指向贵州省和四川省两个节点，而两个省又通过相邻关系（`ADJACENT_TO`）互相连接，形成一个循环结构。于是，查询在遍历到贵州省或四川省后，就会不断在这两个节点之间来回跳转，形成无限循环的路径。可以使用 `times()`函数限制递归的最大深度，从而避免无限循环路径和重复输出。例如，下面查询使用 `times(3)`将上述查询的递归深度限制为不超过 3，输出结果只会包含深度不超过 3 的路径。

```
g.V().has("name", "四渡赤水").repeat(out()).emit().times(3).path().by("name")
```

4. 条件过滤

Gremlin 提供了多种条件过滤功能，可通过 `has()`和 `where()`等步骤实现对节点和关系的筛选。

查询 38：查询长度大于 400 的河流名称及其长度。

```
g.V().hasLabel("Basin").has("length", gt(400)).valueMap("name", "length")
```

输出：

```
[name:[赤水河], length:[436.5]]
```

讲解：`has("length", gt(400))`筛选出 `length`（长度属性）大于 400 的节点；其中 `gt(400)`函数是比较符运算符，表示“greater than”（大于），即如果某个 `Basin` 节点的 `length` 属性值小于等于 400，则会被过滤掉；`valueMap()`会返回一个包含节点中指定属性的键值对（`Map`），`valueMap("name", "length")`提取筛选后的 `Basin` 节点的 `name` 和 `length` 属性。

5. 聚合与分组

Gremlin 支持通过 `group()`、`count()` 等步骤实现分组和聚合操作。

查询 39：统计每个省份包含的战役数量。

```
g.V().hasLabel("Province").group().by("name")
    .by(in("LOCATED_IN").inE("LOCATION").count())
```

输出：

[贵州省:1, 四川省:1]

讲解:

- `group()`将其后续的遍历结果进行分组操;`group()`会生成一个 `Map` 结构(键值对形式), 其中键(key)和值(value)分别通过后面的`.by(...)`指定。
- `.by("name")`指定分组时以节点的“name”属性作为分组键。
- `.by(in("LOCATED_IN").inE("LOCATION").count())`指明对省份节点如何生成分组值, 即通过遍历计算每个省份对应的战役数量; 先沿着 `in("LOCATED_IN")`入边逆向遍历到流域节点, 再通过 `inE("LOCATION")`获取流域节点的 `LOCATION` 入边, 最后通过调用聚合函数`.count()`获得这些 `LOCATION` 边的数量; 每条 `LOCATION` 边都代表了战役节点与流域节点的连接关系。这样就间接统计了一个省份下关联到同一流域的所有战役数量。

6. 更新数据

Gremlin 可以对图数据进行动态更新, 包括插入、修改和删除节点或关系。

查询 40: 为赤水河添加长度单位属性值“公里”。

```
g.V().has("name", "赤水河").property("lenUnit", "公里").iterate()
// 显示赤水河节点的所有属性
g.V().has("name", "赤水河").valueMap(true)
```

输出:

```
[id:12,label:Basin,name:[赤水河],length:[436.5],lenUnit:[公里]]
```

讲解: `property` 函数为筛选出的节点添加属性 `lenUnit`, 并将其值设置为“公里”; `iterate()`的作用是触发查询执行, 用于执行无返回值的操作。接着, 通过显示赤水河节点的所有属性来查看添加 `lenUnit` 属性值操作是否成功; `valueMap(true)`用于显示赤水河节点的所有属性及其值, 参数 `true` 表示包括元数据 (如节点的 `id` 和 `label`)。

5.3 知识图谱数据库

知识图谱数据库是存储与管理大规模知识图谱数据的基础软件系统。与其他数据库管理系统类似, 知识图谱数据库也是结构复杂的系统软件, 被设计专门用于知识图谱数据的高效管理。本节介绍知识图谱数据库的核心组成部分, 包括存储管理器、查询处理引擎和数据访问接口。存储管理器负责数据的持久化存储, 通过优化的数据结构和索引机制, 支持对大规模知识图谱数据的高效存取。查询处理引擎则处理用户的查询请求, 生成查询执行计划, 高效率执行查询算法, 以提供精确的查询结果。数据访问接口为开发者提供与知识图谱数据库交互的手段, 通过这些接口, 用户可以方便地与数据库进行各类交互。知识图谱数据库是实现知识存储与查询的重要基础设施。

5.3.1 存储管理器

1. 知识图谱的非原生存储管理

(1) 基于关系数据库的 RDF 图存储管理

关系数据库经过数十年的发展，拥有从理论到实践的完整成熟体系，是现今进行数据管理的主流产品。基于关系数据库的 RDF 图存储管理是使用关系数据库管理系统来存储和管理 RDF 图数据的方法。这种存储管理方式是将 RDF 三元组按照某种规则映射到关系表中，根据存储 RDF 数据的关系表的不同设计，该方法主要可分为以下几种。

- 三元组表

三元组表（triple table）是最简单的存储方式，将 RDF 图中的每一个三元组存储为表中的一行，表结构固定为三列：主语（subject）、谓语（predicate）和宾语（object），即关系表的模式为：

三元组表（主语，谓语，宾语）

这种存储方式简单直接，但其缺点也很明显：大规模 RDF 图会使三元组表行数过多，查询复杂时会导致大量的三元组表自连接操作，会导致性能瓶颈。图 5.34 给出了“四渡赤水”RDF 图的三元组表存储示例。

主语	谓语	宾语
ex:maoZedong	rdfs:label	"毛泽东"
ex:zhuDe	rdfs:label	"朱德"
ex:redArmy	rdfs:label	"红军"
...
ex:maoZedong	ex:command	ex:crossing
...

图 5.34 三元组表存储示例

- 属性表

属性表（property table）是将具有相似谓语集合的主语划分为不同的表，每个表存储一类主语的数据。这种方案适合对于同类主语的批量查询，但在处理复杂查询时仍需要多个表的连接操作；而且对于规模较大的真实 RDF 图，主语的类别可能有几千种，需要建立几千个表，可能会超过关系数据库的限制。图 5.35 给出了“四渡赤水”RDF 图的属性表存储示例，其中包括了 Person 表、Organization 表和 Battle 表。

Person		
主语	rdfs:label	ex:command
ex:maoZedong	"毛泽东"	ex:crossing

ex:zhuDe	"朱德"	ex:crossing
Organization		
主语	rdfs:label	ex:execute
ex:redArmy	"红军"	ex:longMarch
Battle		

主语	rdfs:label	ex:startDate	ex:endDate	ex:partOf	ex:location
ex:crossing	"四渡赤水"	"1935-01-19"	"1935-03-22"	ex:longMarch	geo:river

图 5.35 属性表存储示例

- 垂直划分

垂直划分（vertical partitioning）方法是为每个谓语创建一张表，每张表由主语和宾语两列组成。表的总数量即知识图谱中不同谓语的数量。这种方式能够支持快速的谓语查询；由于谓语表中的行是按照主语列进行排序的，可以快速执行以“主语-主语”作为连接条件的查询操作。然而，对于未指定谓语的查询，垂直划分方法需要扫描所有表，这会带来较大的性能开销；同时，对于一个主语的更新也将涉及多张表，产生较高的更新代价。图 5.36 给出了“四渡赤水” RDF 图的垂直划分存储示例，其中包括了 6 张谓语表。

ex:command		ex:execute		ex:partOf	
主语	宾语	主语	宾语	主语	宾语
ex:maoZedong	ex:crossing	ex:redArmy	ex:longMarch	ex:crossing	ex:longMarch
ex:zhuDe	ex:crossing				

ex:location		geo:locatedIn		geo:adjacentTo	
主语	宾语	主语	宾语	主语	宾语
ex:crossing	geo:river	geo:river	geo:guizhou	geo:guizhou	geo:sichuan
		geo:river	geo:sichuan	geo:sichuan	geo:guizhou

图 5.36 垂直划分存储示例

- 六重索引

六重索引（sextuple indexing）存储方案是三元组表的扩展，其将三元组主语、谓语和宾语的不同组合建立六张索引表，即

```

spo (主语, 谓语, 宾语)
pos (谓语, 宾语, 主语)
osp (宾语, 主语, 谓语)
sop (主语, 宾语, 谓语)
pso (谓语, 主语, 宾语)
ops (宾语, 谓语, 主语)

```

六重索引通过 6 张表的连接操作解决了三元组表的自连接问题，加速了知识图谱查询处理效率。每种三元组模式查询都可以直接使用索引表进行快速查找，表 5.8 给出了不同三元组模式查询所使用的索引表；还可以通过不同索引表之间的连接操作实现图模式匹配

查询。六重索引的缺点在于其存储空间开销与索引维护代价较大。

表 5.8 三元组模式查询及其所使用的索引表

序号	三元组模式查询	可用索引表
1	(s, p, o)	spo, pos, osp, sop, pso, ops
2	$(s, p, ?x)$	spo, pso
3	$(s, ?x, o)$	sop, osp
4	$(?x, p, o)$	pos, ops
5	$(s, ?x, ?y)$	spo, sop
6	$(?x, ?y, o)$	osp, ops
7	$(?x, p, ?y)$	pos, pso
8	$(?x, ?y, ?z)$	spo, pos, osp, sop, pso, ops

(2) 基于关系数据库的属性图存储管理

属性图的主要组成要素包括节点、边及其相关的属性键值，使用关系数据库存储属性图需要通过将节点、边分别映射为关系表，并将属性作为表中的列进行存储，以实现属性图的关系化存储。具体地，属性图的节点和边分别存储在独立的表中，节点表记录节点的 ID、类型标签和属性，边表记录边的 ID、类型标签、起点节点、终点节点和属性，从而实现对属性图的完整表示。

AgensGraph 是基于关系数据库 PostgreSQL 扩展开发的属性图数据库系统。AgensGraph 采用的就是这种节点表与边表的存储方案。图 5.37 给出了“四渡赤水”属性图在 AgensGraph 中的存储方案示意。可以看出，节点表和边表中的 properties 列是以 JSON 格式保存属性键值对的，这种形式可以适应属性的灵活性和扩展性，避免将每个不同属性作为列而产生的空值问题。

节点表 (nodes)

id	label	properties
1	Person	{name: "毛泽东"}
2	Person	{name: "朱德"}
3	Organization	{name: "红军"}
4	Campaign	{name: "长征"}
5	Battle	{name: "四渡赤水", startDate: 1935-01-19, endDate: 1935-03-22}
6	Basin	{name: "赤水河", length: 436.5}
7	Province	{name: "贵州省"}
8	Province	{name: "四川省"}

边表 (edges)

id	label	source	target	properties
1	COMMAND	maoZedong	crossing	{role: "战略指挥"}
2	COMMAND	zhuDe	crossing	{role: "军事领导"}
3	EXECUTE	redArmy	longMarch	{}

4	PART_OF	crossing	longMarch	{}
5	LOCATION	crossing	river	{}
6	LOCATED_IN	river	guizhou	{}
7	LOCATED_IN	river	sichuan	{}
8	ADJACENT_TO	guizhou	sichuan	{}
9	ADJACENT_TO	sichuan	guizhou	{}

图 5.37 “四渡赤水”属性图在 AgensGraph 中的存储方案示意

(3) 基于键值数据库的 RDF 图存储管理

利用键值数据库作为知识图谱的存储管理器也是知识图谱非原生存储管理的一种方式。例如，RDF 数据库 RDF4J 的 LMDB 就提供了一种典型的基于键值数据库的三元组存储方案。此方案将 RDF 数据中的主体、谓语、宾语经由哈希或字典编码后，存储为键值对，从而有效利用键值数据库在高并发随机读写、分布式扩展等方面的优势。

键值数据库通常以大规模的哈希表为核心，通过(键→值)的映射实现数据的持久化存储。对于 RDF 三元组(s, p, o)，可以将其编码为若干键值条目。例如，将(s, p, o)中每个部分都映射为整数 ID，然后再根据访问需求，组合成以下不同形式的键值对映射：

(主体-谓语 → 宾语列表)
(谓语-宾语 → 主体列表)
(宾语-主体 → 谓语列表)

通过维护这些索引键，查询处理器在执行 SPARQL 查询时，就能够根据三元组模式快速地定位对应记录。

在装载 RDF 数据时，系统会自动将这些三元组映射为键值对。首先，所有 URI 和字面量会被赋予内部整数 ID，例如：

```
ex:maoZedong → 1001
ex:zhuDe → 1002
ex:crossing → 1003
ex:command → 2001
rdfs:label → 2002
"毛泽东" → 3001
"朱德" → 3002
"1935-01-19"^^xsd:date → 3003
"1935-03-22"^^xsd:date → 3004
```

随后，为加速各类查询，LMDB 通常会维护多种前缀/后缀组合的索引。例如，对于(主体-谓语 → 宾语)的索引，会将(1001,2002)作为键，并将 3001 放入对应值的列表中；对于(谓语-宾语 → 主体)的索引，可能将(2002,3002)作为键，值中记录 1002 等等。这样一来，当 SPARQL 查询需要匹配类似：?x rdfs:label "朱德" 时，就能够在键 (2002,3002) 的索引下直接获得主语 ID 1002，并通过字典表快速映射回 ex:zhuDe。在 LMDB 底层，这些键

值被放入对应的 B+树或哈希索引结构中，以便快速增删改查；同时，LMDB 将数据库文件直接映射到内存中，这使得 RDF 数据的查询和插入操作非常高效，尤其是在只读查询场景下。

基于键值数据库的 RDF 图存储具有结构简单、灵活可扩展和高并发读写的优势。哈希/字典编码后仅需维护相应键值索引，不必显式地创建或修改关系表结构；键值库往往可以水平扩展；键值数据库天然擅长高并发、海量小对象的读写，适合存储大规模三元组。但键值库的缺点在于，与关系数据库相比，尚缺少较成熟的成本模型及自动优化算法。

(4) 基于键值数据库的属性图存储管理

我们也可以利用键值数据库来管理属性图数据。JanusGraph 就是一个典型的基于键值数据库的属性图存储管理系统，能够基于分布式键值存储引擎构建高可扩展、高并发的属性图数据存储；其通过灵活的表结构设计和索引策略，将属性图中的节点、关系、属性分拆为一系列键值对，以满足对大批量分布式部署和复杂图查询的需求。

JanusGraph 利用底层键值数据库的扩展能力，将属性图（节点、边及其属性）按特定规则拆分成若干键值对，并分布到不同数据节点上，实现对大规模属性图数据的水平扩展存储。JanusGraph 的存储方案主要包括：

- 每个节点和边会有一个唯一的内部 ID；
- 节点属性存储为：((节点 ID + 属性键) → 属性值)；
- 节点到关联边的邻接关系表示为“某节点在某种边类型或某种方向上的所有边 ID 列表”，例如：((节点 ID + 边类型 + 边方向) → [边 ID1, 边 ID2, ...])；
- 边属性存储为：((边 ID + 属性键) → 属性值)；
- 边的连接信息记录其起点节点和终点节点，即：
(边 ID + "outVertex") → 起点节点 ID
(边 ID + "inVertex") → 终点节点 ID

对于下面插入两个节点和一条边的 Gremlin 语句：

```
g.addV("Person").property("name","毛泽东").as("m")
.addV("Battle").property("name","四渡赤水").as("c")
.addE("COMMAND").from("m").to("c").property("role","战略指挥")
```

JanusGraph 会将之拆分为以下若干条键值对进行存储：

- 节点 maoZedong (ID=1001)：

```
(1001 + "label") → "Person"
(1001 + "name") → "毛泽东"
```

- 节点 crossing (ID=1002):

```
(1002 + "label") → "Battle"  
(1002 + "name") → "四渡赤水"
```

- 边 (edgeID=2001):

```
(2001 + "label") → "COMMAND"  
(2001 + "role") → "战略指挥"  
(2001 + "outVertex") → 1001  
(2001 + "inVertex") → 1002
```

- 邻接关系:

```
(1001 + "COMMAND" + "out") → [2001]  
(1002 + "COMMAND" + "in") → [2001]
```

如执行以下查询，查找“毛泽东指挥的战役”：

```
g.V().has("Person","name","毛泽东").outE("COMMAND").has("role","战略指挥")  
.inV().valueMap()
```

JanusGraph 的大体执行流程是：

- 先根据属性索引找到“name=毛泽东”的节点 ID 为 1001；
- 再根据键值映射“(1001 + "COMMAND" + "out") → [2001]”找到边 ID 为 2001；
- 检查边属性(2001 + "role")是否为“战略指挥”，若匹配，则取(2001 + "inVertex")=1002；
- 最后将 1002 映射回 crossing 节点，输出其属性，如{name="四渡赤水"}。

在底层键值数据库中，这些操作依赖一系列快速随机读写访问（哈希表或 B+树查找），JanusGraph 的分布式架构确保即使数据规模增大，也能通过扩展后端的键值存储进行负载均衡。此外，除了基础的键值对，JanusGraph 还可以集成外部全文检索与多字段索引系统（如 ElasticSearch、Solr 等），以支持复杂图查询的高效执行。

2. 知识图谱的原生存储管理

(1) RDF 图的原生存储管理

RDF 图的原生存储方案旨在对 RDF 三元组数据做紧凑和高效的存储和索引结构组织，从而最大程度地支持 RDF 图数据的查询处理操作。Apache Jena 的 TDB2（下称“TDB2”）便是一个典型的 RDF 三元组原生存储管理器。TDB2 是 Jena 的一种内置三元组数据库，专门针对 RDF 三元组结构设计。与 Jena 体系相结合后，TDB2 可通过 Jena API 或 SPARQL Endpoint 提供高效的三元组插入、查询和推理处理，其核心思路包括：

- 字典编码：为 RDF 中主语、谓语、宾语（包括 URI、字面量、空节点）都分配整数 ID，从而以紧凑方式存储，减少空间占用并加快索引查找。例如，

```

ex:maoZedong → ID = 1001
ex:zhuDe → ID = 1002
ex:crossing → ID = 1003
rdfs:label → ID = 2001
ex:command → ID = 2002
"毛泽东"^^xsd:string → ID = 3001
"朱德"^^xsd:string → ID = 3002
"四渡赤水"^^xsd:string → ID = 3003
"1935-01-19"^^xsd:date → ID = 3004
"1935-03-22"^^xsd:date → ID = 3005

```

- 多重索引：TDB2 维护多种三元组索引（如 SPO、POS、OSP 等），保证对各种组合形式的 SPARQL 模式都能迅速定位。以 SPO 作为示例：SPO 索引经过字典编码，三元组 (ex:maoZedong, ex:command, ex:crossing) 会被存储为 (1001, 2002, 1003) 并写入该索引对应的 B+树存储文件中。

- 事务与并发：TDB2 支持 ACID 事务语义，通过多版本并发控制等机制，为高并发访问提供可靠性。

将 RDF 数据加载到 TDB2 时，其会将所有三元组解析并写入到专门的三元组索引文件和字典文件，并对空节点和字面量进行特殊处理。

当 RDF 三元组数据写入 Jena 时，会进行以下操作：

- 将 URI、字面量等进行字典编码，若已有编码则沿用；
- 将三元组 (s, p, o) 转为相应字典编码 ID，即得到 (sid, pid, oid)；
- 将 (sid, pid, oid) 插入到 SPO、POS、OSP 等索引的数据结构中，并保持各 B+树的排序/平衡；
- 提交事务后，数据落盘持久化。

当需要执行一条 SPARQL，如：

```

SELECT ?who
WHERE {
  ?who ex:command ex:crossing .
  ?who rdfs:label ?name .
}

```

- 先将“ex:command”映射为 2002、“ex:crossing”映射为 1003；
- 在 POS 索引的 B+树结构上，根据 (?s=ANY, p=2002, o=1003) 查找所有满足谓语 =ex:command、宾语=ex:crossing 的条目；
- 得到结果包含 (?s=1001, p=2002, o=1003) 和 (?s=1002, p=2002, o=1003)，?s 分别对应 ex:maoZedong 与 ex:zhuDe。

图 5.38 给出了使用 POS 索引进行三元组模式(?s=ANY, p=2002, o=1003)查找的示意。

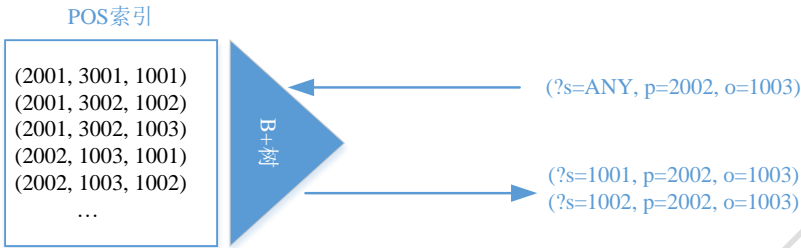


图 5.38 使用 POS 索引进行三元组模式查找

在 TDB2 内，B+树是支撑高效三元组检索的关键，配合多重索引 SPO、POS、OSP，能够管理数百万甚至上亿条 RDF 三元组，能够满足各种 SPARQL 查询模式下的快速定位需求。

(2) 属性图的原生存储管理

属性图的原生存储管理最典型的系统是 Neo4j 数据库，与基于关系或键值数据库的存储有所不同，它在底层针对属性图进行了数据结构与索引的原生优化设计。Neo4j 对节点、边，以及它们的属性都使用自定义的存储格式，以提升图遍历和查询的效率。

Neo4j 将节点、关系、属性都进行独立存储，其基本文件结构包括：

- 节点存储文件（node store）：每个节点在创建时分配一个独立的内部 ID；节点存储文件中记录节点的基本元信息，如节点 ID、指向关系记录的起始指针、指向属性记录的指针、节点标签信息等。
- 关系存储文件（relationship store）：每条关系同样有一个唯一 ID；在关系存储文件中，记录了起始节点 ID、终止节点 ID、关系类型、关系属性指针等信息。关系文件也维护指向上一条或下一条关系的指针，以便快速遍历一条链上的关系。
- 属性存储文件（property store）：无论是节点，还是关系，其属性都会存放在属性存储文件里，通过某个属性记录指针与节点存储文件和关系存储文件相连。属性通常以(属性键 ID, 类型, 属性值) 形式存储。
- 标签/索引存储文件（label/index store）：节点标签（label）也通过一个 ID 记录在相应的标签存储文件里，节点记录中会引用该标签 ID；若在某些属性上创建了索引，会给这部分属性建立 B+树或类似索引文件，以加速按属性值的查询。

通过这些原生存储方式，Neo4j 可以在底层以图结构形式更加优化地管理节点、关系以及属性数据，相比将图数据在关系表或键值对存储中拆分，原生存储方式更能直观高效地执行图遍历等图查询操作。Neo4j 实现节点和边快速定位的一个关键之处是采取了“定长记录”的存储方案，即将具有定长记录的图结构（节点和边）与具有变长记录的属性数

据分开存储。假设一个节点记录长度是 9 字节，如果要查找 ID 为 99 的节点记录所在位置（ID 从 0 开始），则可直接到节点存储文件第 891 个字节处访问（存储文件从第 0 个字节开始），这样，已知记录 ID 就可以常熟时间代价直接访问到其存储位置。

Neo4j 实现了原生属性图存储需要具备的一个重要特性，即“无索引邻接”（index-free adjacency）特性。在关系数据库或键值数据库中，如果要找出某个实体（行或记录）与其邻居实体（外键关联或引用对象）之间的关系，往往需要依赖索引（B+树、Hash 索引等）或进行额外的查表操作来定位。随着数据量的增大，在查询时可能需要多次索引查找才能拼出一条关系路径，图遍历查询代价相对较高。而在属性图数据库中，“无邻接索引”指的是每个节点都直接持有指向其相邻节点及关系的物理指针，换言之，一个节点本身就知道它跟哪些节点及关系连接。这样一来，在遍历图（如多跳查询）时，无需在每跳都进行额外的索引扫描，而是只需顺着节点内存结构所保存的指针直接跳转到关联的关系对象，重复此操作，直到抵达目标节点。

下面以“查询四渡赤水战役的指挥官是谁”为例，说明 Neo4j 属性图存储方案的使用过程。

- 当我们访问节点(b:Battle { name: "四渡赤水" })，在其底层物理存储结构中，(b)节点含有一个指针列表，指向所有与(b)相连的关系记录。
- 这些关系记录中，会包含声明“这是一条:COMMAND 关系，关联到节点(m:Person { name: "毛泽东" })”以及“另一条 :COMMAND 关系，关联到节点(z:Person { name: "朱德" })”。
- 当查询处理引擎要执行“(b)的指挥官是谁？”时，无需再查找辅助索引，而是直接访问(b)节点内存结构中存储的指针，跳转到相应关系记录，进而找到目标节点(m)与(z)。这样就完成了一步邻接遍历。

需要注意的是，“无邻接索引”并不意味着属性图存储就完全不需要其它索引。一般来说，属性图存储还是会在属性（如节点的 name 属性）上建立一套 B+树或哈希索引，以便做快速定位（如执行查询 MATCH (n:Person { name: '毛泽东' })）。首先通过索引查找属性，然后再通过无邻接索引做快速图遍历，这两者配合得当，可以取得优良的性能。

5.3.2 查询处理引擎

1. 面向 RDF 图的查询处理

面向 RDF 图的查询处理主要是对 SPARQL 语言的查询处理操作。我们以 Jena 的查询处理引擎 ARQ 为例，对面向 RDF 图的查询处理进行讲解。

在 Jena 中，ARQ 是核心的查询处理器（query engine），其主要职责如下：

- 解析 (parsing): 将 SPARQL 查询语句解析为抽象语法树 (AST)。
- 优化 (optimization): 基于查询模式以及底层索引和统计信息, 对查询执行顺序、连接顺序作一定程度的优化。
- 执行 (execution): 将拆分、重写后的查询操作映射到存储管理器 (TDB2) 的多重索引 (SPO、POS、OSP) 上, 或者利用外部函数库完成过滤。
- 结果返回: 最终将匹配结果以表格、图结构或 JSON、XML 等格式返回给上层应用。

对于一个 SPARQL 查询, ARQ 通常将其分解为一组基本图模式 (Basic Graph Pattern, BGP), 并对这些 BGP 进行联接。若有复杂的 FILTER、OPTIONAL、UNION 等子句, 则需要进行额外的算子处理与结果合并。

例如, 想要找出“四渡赤水战役指挥官实体及名字”, SPARQL 查询如下:

```
SELECT ?person ?name
WHERE {
  ?person ex:command ex:crossing .
  ?person rdfs:label ?name .
}
```

在 ARQ 中, 这条 SPARQL 查询首先会被解析为抽象语法树, 其中包含了两个三元组模式:

```
(?person, ex:command, ex:crossing)
(?person, rdfs:label, ?name)
```

随后, ARQ 会基于 TDB2 的索引统计信息, 对这两个三元组模式进行优化, 具体包括: 确定先在哪一个索引 (如 SPO、POS 或 OSP) 上进行扫描; 确定查询算子的执行算法 (如连接算子使用嵌套循环连接 (nested loop join) 还是索引连接 (index join)); 在最终执行阶段将结果合并, 以得到变量(?person, ?name)对应值的绑定; 经过一系列步骤, 将抽象语法树变换为查询执行计划树 (query plan tree); 自叶子节点向根节点执行每个节点上指定的操作, 最后根节点将返回查询执行结果。图 5.39 给出了上面 SPARQL 的查询执行计划树的一个简化示意图, 其中每个节点上的操作如下:

(1) BGP 模式 1: (?person ex:command ex:crossing)

(a) 字典映射

```
ex:command → predicateID (如 2002)
ex:crossing → objectID (如 1003)
```

(b) 索引检索

在 POS 的索引中查找所有满足 (S=?person, P=2002, O=1003) 的记录。得到匹配:

```
(personID1, 2002, 1003)
(personID2, 2002, 1003)
```

(2) BGP 模式 2: (?person rdfs:label ?name)

(a) 字典映射

```
rdfs:label → predicateID (如 2001)
```

(b) 索引检索

在 POS 的索引中查找所有满足 (S=?person, P=2001, O=?name) 的记录。得到匹配:

```
(personID1, 2001, nameID1)
(personID2, 2001, nameID1)
```

(c) Join on ?person

将 BGP 模式 1 与 BGP 模式 2 的结果集按 ?person 变量进行连接 (join)。personID1 和 personID2 在 BGP 模式 1 和 BGP 模式 2 中都出现, 得到连接结果:

```
(personID1, 2002, 1003)
(personID1, 2001, nameID1)
(personID2, 2002, 1003)
(personID2, 2001, nameID1)
```

(d) Project ?person ?name

投影输出 ?person 和 ?name 变量的值。在上一步的连接结果中, 将 (personID1, nameID1) 和 (personID2, nameID2) 进行字典编码反映射, 得到最终输出结果:

person	name
ex:maoZedong	毛泽东
ex:zhuDe	朱德

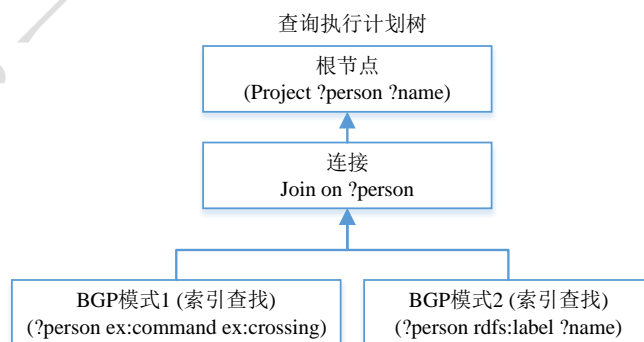


图 5.39 SPARQL 查询执行计划树示例

实际上，在 ARQ 中，查询执行计划树的生成还需要查询优化器（query optimizer）进行优化变换，其过程较为复杂，主要采取的查询优化机制包括：

- 选择性估计：根据收集的统计信息，查询处理器会估算某一谓词或对象的选择性，从而决定以何种顺序进行三元组模式的连接。一般会先用高选择性的三元组模式减少结果集，然后再连接其他模式。
- 过滤下推：对于 FILTER 运算，查询处理器会尝试将其尽量下推到最靠近数据读取的地方，从而减少中间结果量，提升性能。

在 Jena ARQ 查询处理引擎中，SPARQL 查询大多会拆分为若干 BGP，然后基于 TDB2 的多重索引进行模式匹配；若有 FILTER、OPTIONAL、UNION 等语句，执行计划还会包含相应的算子与连接逻辑；查询执行计划树会采取多种优化策略来调度各个算子的执行；最终产生符合 SPARQL 查询的结果集。

2. 面向属性图的查询处理

面向属性图的查询处理以 Neo4j 图数据库的查询处理引擎为典型代表。与 Jena ARQ 类似，Neo4j 查询处理引擎的主要流程包括：

- 解析（parsing）：将用户输入的 Cypher 语句转换为内部语法树及逻辑执行计划；
- 优化（optimization）：基于模式、索引、统计信息，对匹配顺序与算子进行一定程度的优化调度；
- 执行（execution）：调度磁盘或内存中的节点、关系存储，按顺序执行索引查找、图遍历、算子连接等操作，最终返回结果集。

例如，要查询“四渡赤水战役的指挥官及其在指挥中的角色”，可以编写以下 Cypher 语句：

```
MATCH (p:Person)-[r:COMMAND]->(b:Battle { name: "四渡赤水" })
RETURN p.name AS commander, r.role AS role
```

Neo4j 在内部会生成查询的逻辑执行计划表示，包括节点扫描（对 Battle 节点）、条件过滤（WHERE b.name="四渡赤水"）、关系扩展（:COMMAND 边）、再到目标节点（p:Person）的属性获取等。

在优化阶段，Neo4j 可能进行以下步骤：

- 确定如何先定位(b:Battle { name: "四渡赤水" })：如果为该属性建立了索引，则可通过索引快速找到满足条件的节点 ID，否则需进行标签扫描（label scan）。
- 启动从 b 节点出发的“关系遍历（expand）”操作，仅遍历[:COMMAND]类型的关系，找到与之相连的节点 p:Person。

- 对 p.name 和 r.role 做投影输出。

Neo4j 采取的查询执行策略通常是“基于模式的索引查找 + 图遍历”，充分结合了索引查找和图遍历搜索两者的优势。Neo4j 提供了“EXPLAIN”或“PROFILE”命令，能向用户展示大致的执行计划。以下为一个简化的执行计划树示例：

```
+-----+
|                               Plan                               |
+-----+
| NodeIndexSeek                                                       |
|   Label: Battle                                                     |
|   Index: :Battle(name)                                             |
|   Key: name                                                         |
|   Value: "四渡赤水"                                                |
|                                                                       |
| Expand(All)                                                         |
|   Relationship: COMMAND (outgoing)                                  |
|   Node: Person                                                      |
|                                                                       |
| Projection                                                          |
|   Expressions: p.name AS commander, r.role AS role                |
+-----+
```

- NodeIndexSeek 算子：由于我们在:Battle(name)上建立了索引，Neo4j 可以直接通过索引定位到 name="四渡赤水"的节点(b)；一旦查找到该节点 ID，就从存储层读取其属性(startDate, endDate)并验证它是:Battle 标签。
- Expand(All)算子：从(b)节点出发，针对所有[:COMMAND]出边，获取所有匹配的关系 r 以及关系另一端的节点 p；关系 r 中保存的 role 属性，也在这一阶段关联到结果中；目标节点 p 需满足标签:Person。
- Projection 算子：将 p.name 投射为 commander，r.role 投射为 role，返回。

在实际应用中，Neo4j 的查询执行计划还会包含更多细节，如多步图遍历、可选匹配、参数化过滤、分组聚合等算子。如果用户有更深层次的大规模图分析需求，可结合 Neo4j 提供的 GDS 模块（Graph Data Science）或外部数据分析平台进行集成。

5.3.3 数据访问接口

1. RDF 图的访问接口

RDF 图数据库包括本地编程 API 访问以及作为远程服务的 SPARQL 端点（SPARQL Endpoint）以及 REST 接口等多种形式。典型的 RDF 图数据库工具，如 Apache Jena 与 Eclipse RDF4J，都提供了完善的 SPARQL 协议实现和多种编程接口，使得用户能够以标准化、可扩展的方式对 RDF 图数据进行查询和更新。

(1) SPARQL 协议与 SPARQL 端点

SPARQL 协议（SPARQL Protocol）是由 W3C 标准化的一套基于 HTTP 的通信协议，规定了客户端如何与 RDF 数据源进行交互，执行 SPARQL 查询或更新指令。其主要特点包括：通过 HTTP 请求发送 SPARQL 查询与更新语句；可在 URL 或 HTTP POST 的请求体中携带 SPARQL 语句和查询参数；支持多种返回格式，如 JSON、XML、CSV/TSV 等。

SPARQL 端点是对外开放的网络服务端点，是 SPARQL 协议的具体实现对外提供的访问服务接口。客户端（应用程序或命令行工具）只需知道 SPARQL 端点的 URL，就能通过标准 HTTP 请求执行查询和更新。

SPARQL 查询可以使用 GET 或 POST 请求，其 URL 示例为：

```
http://example.org/sparql?query=... (查询)
```

SPARQL 更新使用 POST 请求，其 URL 示例为：

```
POST http://example.org/sparql?update=... (更新)
```

在 Apache Jena 中，Fuseki 是 SPARQL 协议与端点的实现；在 RDF4J 中，则用其自带 Jetty/Webapp 支持 SPARQL 协议与端点的实现。它们都具备 Web 界面，帮助用户测试查询。

假设“四渡赤水”RDF 图提供的 SPARQL 端点为 `http://localhost:3030/sidu/sparql`，则用户就能通过以下方式访问：

- 执行 SPARQL 查询

在浏览器或其他客户端中发送 GET 请求：

```
http://localhost:3030/sidu/sparql?query=
SELECT ?person ?label
WHERE {
  ?person ex:command ex:crossing;
    rdfs:label ?label .
}
```

- 执行 SPARQL 更新

对知识图谱作更新操作，插入一条“周恩来也指挥了四渡赤水”的三元组：

客户端以 HTTP POST 的方式将该插入命令发送到 SPARQL 端点：

```
http://localhost:3030/sidu/sparql?update=
PREFIX ex: <http://example.org/>
INSERT DATA {
  ex:zhouEnlai ex:command ex:crossing .
  ex:zhouEnlai rdfs:label "周恩来" .
}
```

通过这样的方式，用户利用标准协议对 RDF 图数据进行远程访问与管理，而无需深入了解 RDF 数据库的本地实现。

(2) 本地编程接口

(a) Apache Jena

Apache Jena 是一个功能齐全的开源 Java 框架，针对 RDF、SPARQL、推理等提供了多层次的 API。常见的主要接口及其用法如下：

- **Model 与 Dataset:** Model 接口是 Jena 中对 RDF 图的抽象；Dataset 接口可包含多个命名图和一个默认图，适用于多图场景。
- **TDB2 的本地访问:** TDB2Factory 可以创建或打开一个本地 TDB2 数据集 (Dataset)；在获取到 Dataset 与 Model 后，即可编程读写 RDF 三元组，执行 SPARQL 查询或更新。示例 Java 代码如下：

```
Dataset dataset = TDB2Factory.createDataset("path/to/tdb2");
try (dataset) {
    // 在 try 块内进行读写操作
    Model model = dataset.getDefaultModel();
    // ... 执行对该 model 的增删改查
}
```

- **SPARQL 查询与更新:** Jena 提供了 QueryExecutionFactory、QueryExecution 等类来执行 SPARQL，示例 Java 代码如下：

```
String sparql = "SELECT ?s ?p ?o WHERE { ?s ?p ?o } LIMIT 10";
try (QueryExecution qe = QueryExecutionFactory.create(sparql, defaultModel)) {
    ResultSet rs = qe.execSelect();
    while (rs.hasNext()) {
        QuerySolution sol = rs.next();
        System.out.println(sol.get("s") + " " + sol.get("p") + " " + sol.get("o"));
    }
}
```

SPARQL 更新同理，可使用 UpdateAction.execute()，或更灵活的 UpdateRequest 对象。

- **推理接口:** 若需要在本地进行 RDFS/OWL 推理，可以使用推理模型 InfModel 类，在加载本地规则或本体后，对推断结果进行查询。

(b) Eclipse RDF4J

另一个主要的开源 RDF 图数据库 RDF4J 同样采用的 Java 编程，支持 SPARQL 和多种推理模式，其核心组件为 Repository 与 RepositoryConnection：

- **Repository** 类似 Jena 的 Dataset，用于管理一个或多个 RDF 图；内置多种 Repository

类型，例如内存型 `MemoryStore`、本地文件型 `NativeStore`、或对接特定存储后端的类型。`RepositoryConnection` 是通过 `Repository.getConnection()` 获取的一个连接实体，可在其中执行读取或写入操作。示例 Java 代码如下：

```
Repository repo = new SailRepository(new MemoryStore());
repo.init();
try (RepositoryConnection conn = repo.getConnection()) {
    // 1) 添加三元组
    conn.add(new File("sidu.ttl"), "http://example.org/", RDFFormat.TURTLE);
    // 2) SPARQL 查询
    String queryString = "SELECT ?s ?p ?o WHERE { ?s ?p ?o } LIMIT 5";
    TupleQuery tupleQuery = conn.prepareTupleQuery(QueryLanguage.SPARQL, queryString);
    try (TupleQueryResult result = tupleQuery.evaluate()) {
        while (result.hasNext()) {
            BindingSet bs = result.next();
            // ...
        }
    }
}
```

- SPARQL 更新

RDF4J 也可执行 SPARQL 更新操作，示例 Java 代码为：

```
String updateString = "INSERT DATA { ex:zhouEnlai ex:command ex:crossing . }";
conn.prepareUpdate(QueryLanguage.SPARQL, updateString).execute();
```

- 推理配置：对推理的支持依赖配置后台的 SAIL 层（RDF4J 的存储与推理层封装），如 `RDFS SAIL`、`OWLIM SAIL` 等，能自动生成推理三元组。

(c) Python RDFLib

除 Java 编程接口外，在 Python 中可使用 `RDFLib` 库读写 RDF 以及执行 SPARQL 查询。示例 Python 代码为：

```
import rdflib
g = rdflib.Graph()
g.parse("sidu.ttl", format="turtle")

# SPARQL 查询
q = """
SELECT ?s ?p ?o
WHERE {
    ?s ?p ?o .
} LIMIT 5
"""
for row in g.query(q):
    print(row)
```

`RDFLib` 的优势在于 Python 环境易于快速开发和数据处理，适合需要脚本化或前期分

析的场景。但其性能往往不及专业的 RDF 三元组数据库。

2. 属性图的访问接口

作为属性图主流数据库的 Neo4j 在系统交互层面提供了多种访问接口，主要包括：Bolt 协议、Neo4j 驱动程序（driver）、REST API 接口以及 Neo4j 图形界面与云端部署。这些接口可分别满足不同编程语言或部署场景下的需求。下面将详细介绍各接口的主要特点和调用方式，并以“四渡赤水”知识图谱的示例数据加以说明。

(1) Bolt 协议

Bolt 是 Neo4j 官方开发的二进制网络协议，专为高效执行 Cypher 查询而设计。它也支持事务处理、可扩展认证与加密传输。当本地或远程应用程序与 Neo4j 数据库交互时，往往通过“bolt://host:7687”（host 代表本地或远程服务器地址）这种连接串，以及官方/第三方驱动来发送 Cypher 命令并获取结果。

(2) Neo4j 驱动程序

Neo4j 提供了多种驱动程序，覆盖常见的编程语言。它们都通过 Bolt 协议与数据库进行通信。主要编程语言的驱动程序示例代码如下：

(a) Java

```
import org.neo4j.driver.*;

public class Neo4jExample {
    public static void main(String[] args) {
        // 1. 创建驱动实例
        Driver driver = GraphDatabase.driver("bolt://localhost:7687",
            AuthTokens.basic("neo4j", "password"));

        // 2. 打开会话
        try (Session session = driver.session()) {
            // 3. 执行 Cypher 查询
            String cypher = ""
                MATCH (p:Person)-[r:COMMAND]->(b:Battle { name: "四渡赤水" })
                RETURN p.name AS commander, r.role AS role
                "";
            Result result = session.run(cypher);

            // 4. 处理结果
            while (result.hasNext()) {
                Record record = result.next();
                System.out.println("Commander: " + record.get("commander").asString()
                    + ", Role: " + record.get("role").asString());
            }
        } finally {
            // 关闭驱动
        }
    }
}
```

```

        // 5. 关闭驱动
        driver.close();
    }
}
}

```

(b) Python

```

from neo4j import GraphDatabase

uri = "bolt://localhost:7687"
driver = GraphDatabase.driver(uri, auth=("neo4j", "password"))

def find_commanders(tx):
    query = """
    MATCH (p:Person)-[r:COMMAND]->(b:Battle { name: "四渡赤水" })
    RETURN p.name AS commander, r.role AS role
    """
    result = tx.run(query)
    return [(record["commander"], record["role"]) for record in result]

with driver.session() as session:
    c_list = session.read_transaction(find_commanders)
    for c in c_list:
        print("Commander:", c[0], ", Role:", c[1])

driver.close()

```

(3) REST API 接口

除了 Bolt 协议，Neo4j 还提供了基于 HTTP/REST 的访问方式。用户可发送 POST 请求，包含 Cypher 查询，即可获取 JSON 或 CSV 格式的结果。下面是一个简化示例（使用 curl）：

```

curl -i -X POST \
-H "Content-Type: application/json" \
-u neo4j:password \
-d '{
  "statements": [
    {
      "statement": "MATCH (p:Person)-[r:COMMAND]->(b:Battle { name: \"四渡赤水\" })
RETURN p.name AS commander, r.role AS role"
    }
  ]
}' \
http://localhost:7474/db/neo4j/tx/commit

```

Neo4j 会返回一个 JSON 响应，其中包含数据列表，供前端或脚本解析。这种接口适合一些不想或不便使用 Bolt 驱动的场景，或快速测试/调试插件化访问。

(4) Neo4j 图形界面与云端部署

- **Neo4j Browser:** Neo4j 自带的 Web 界面，默认运行在 `http://localhost:7474` 上，允许用户直接在浏览器里输入 Cypher 语句、查看结果与可视化图形。对于“四渡赤水”知识图谱示例，可以直接在 Browser 中看到具体节点、关系及其属性，形象地展示知识图谱结构。
- **Neo4j Desktop:** 提供图形化的管理与调试环境，本地启动和管理多个 Neo4j 实例。整合了 Neo4j Browser、插件、扩展等，适合开发和学习阶段使用。
- **Neo4j Aura:** 云端托管的 Neo4j 数据库服务，无需自行运维，只需在云端地址上进行访问和编程即可。访问方式类似本地，但连接 URI 放在云端，仍可用 Bolt 协议或 Neo4j 驱动程序。

5.4 本章小结

本章围绕“知识存储与查询”展开，系统地介绍了知识数据模型、知识查询语言和知识图谱数据库三大部分内容。首先，在知识数据模型层面，分别深入探讨了 RDF 图和属性图这两种广泛采用的知识图谱数据模型。RDF 图以三元组为基础，通过一系列标准化技术，实现了语义万维网场景；属性图则通过节点、边和属性键值的形式来表示实体与关系，并在图数据库中得得到普遍应用。两种数据模型各具优势，为后续知识图谱的存储与查询的实现奠定了数据模型基础。

接着，在知识查询语言方面，本章重点阐释了用于 RDF 数据的 SPARQL 语言、用于属性图数据的 Cypher 语言和 Gremlin 语言。通过示例讲解，读者可以感受到它们在知识图谱查询与更新等操作方面各自的特性与使用方法。

随后，本章介绍了知识图谱数据库，包括存储管理器、查询处理引擎与数据访问接口三大核心组件。存储管理器层面既讨论了基于关系和键值数据库的非原生存储方案，也呈现了针对 RDF 或属性图进行优化的原生存储模式。在查询处理引擎部分说明了查询语法规则、查询优化和查询计划执行过程。数据访问接口方面，结合常用的知识图谱数据库系统，展示了本地编程 API 与远程服务协议等多种访问方式。

通过本章的学习，读者能够从整体上把握知识图谱数据库的存储与查询基本原理，熟悉面向知识图谱数据的主要查询语言与执行机制，并了解如何使用多种数据访问接口实现知识图谱数据管理功能。这些知识对于后续在实际项目中选择合适的知识图谱数据库系统、进行数据导入、设计查询与推理服务，以及开发可视化与应用分析场景，都具有重要的指导意义。

参考文献

- [1] (RDF 标准) Richard Cyganiak, David Wood, Markus Lanthaler (2014). RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation. <https://www.w3.org/TR/rdf11-concepts/>
- [2] (语义万维网中的 RDF 模型) Dean Allemang, James A. Hendler. (2011). Semantic Web for the

- Working Ontologist (2nd ed.). Amsterdam: Morgan Kaufmann. ISBN 978-0-12-385965-5.
- [3] (属性图定义) Gutierrez, C., Hidders, J., Wood, P.T. (2018). Graph Data Models. In: Sakr, S., Zomaya, A. (eds) Encyclopedia of Big Data Technologies. Springer, Cham. https://doi.org/10.1007/978-3-319-63962-8_81-1
- [4] (属性图与 Neo4j 图数据库) Ian Robinson, Jim Webber, Emil Eifrem (2015). Graph Databases 2nd. ISBN 9781491930892.
- [5] (SPARQL 查询语言) Steve Harris, Andy Seaborne. (2013). SPARQL 1.1 Query Language. W3C Recommendation. <https://www.w3.org/TR/sparql11-query/>
- [6] (SPARQL 更新语言) Paula Gearon, Alexandre Passant, Axel Polleres. (2013). SPARQL 1.1 Update. W3C Recommendation. <https://www.w3.org/TR/sparql11-update/>
- [7] (SPARQL 联邦查询) Eric Prud'hommeaux, Carlos Buil-Aranda. (2013). SPARQL 1.1 Federated Query. W3C Recommendation. <https://www.w3.org/TR/sparql11-federated-query/>
- [8] (SPARQL 协议) Lee Feigenbaum, Gregory Todd Williams, Kendall Grant Clark, Elias Torres. (2013). SPARQL 1.1 Protocol. W3C Recommendation. <http://www.w3.org/TR/sparql11-protocol/>
- [9] (Wikidata 官方的 SPARQL 教程) Wikidata:SPARQL tutorial (2025). https://www.wikidata.org/wiki/Wikidata:SPARQL_tutorial
- [10] (Wikidata 数据集的另一个查询服务教程) Wikidata Query Service Tutorial (2025). <https://wdqs-tutorial.toolforge.org/>
- [11] (Neo4j 官方的 Cypher 语言文档) Neo4j Cypher Manual (2025). <https://neo4j.com/docs/cypher-manual/current/introduction/>
- [12] (GQL 的 ISO 国际标准) ISO. (2024). ISO/IEC 39075:2024 Information technology — Database languages — GQL. Available: <https://www.iso.org/standard/76120.html>
- [13] (Gremlin 语言文档) Project TinkerPop. (2023). Apache TinkerPop and Gremlin Documentation. Retrieved from <https://tinkerpop.apache.org/docs/current/>
- [14] (RDF 图的垂直划分存储方案) Abadi D J, Marcus A, Madden S R. SW-Store: a vertically partitioned DBMS for Semantic Web data management. Vldb Journal, 2009, 18(2):385-406.
- [15] (RDF 图的六重索引存储方案) Weiss C, Karras P, Bernstein A. Hexastore: Sextuple Indexing for Semantic Web Data Management. Proceedings of the VLDB Endowment, 2008: 1008-1019.
- [16] (AgensGraph 属性图数据库) AgensGraph: Powerful Multi-Model Graph Database. <https://github.com/bitnine-oss/agensgraph>.
- [17] (RDF 图数据库 RDF4J) Eclipse RDF4J. <https://rdf4j.org/>
- [18] (属性图数据库 JanusGraph) JanusGraph. <https://janusgraph.org/>
- [19] (RDF 图数据库 Apache Jena) Apache Jena. <https://jena.apache.org/>
- [20] (Python 语言的 RDF 库 RDFLib) RDFLib. <https://rdflib.dev/>

本章习题

一、选择题

- 知识图谱中采用的两种主要数据模型是：（ B ）
 - A. XML 图和 JSON 图
 - B. RDF 图和属性图
 - C. 层次模型和网状模型
 - D. 关系模型和对象模型
- 以下关于 RDF 图数据模型中国际资源标识符(IRI)的描述,哪一项是正确的?（ D ）
 - A. IRI 是 RDF 图中用于表示实体之间关系的标准标识系统。
 - B. IRI 是 RDF 图中用于表示实体属性的标准标识系统,但不保证全局唯一性。
 - C. IRI 仅在属性图数据模型中使用,用于在节点和边上附加属性。
 - D. IRI 在 RDF 图中用于唯一标识实体,确保每个实体在全球范围内具有唯一的标识符。
- 以下关于 SPARQL 中 ORDER BY 子句的描述,哪一项是正确的?（ B ）
 - A. ORDER BY 子句用于限制查询结果的数量。
 - B. ORDER BY 子句用于对查询结果进行排序。
 - C. ORDER BY 子句用于跳过指定数量的结果。
 - D. ORDER BY 子句用于指定结果返回的起始位置。

二、填空题

- RDF 图是由三元组(s, p, o)组成的集合,其中,s、p、o 分别称为三元组的__主语__、__谓语__和__宾语__。而属性图的基本要素包括__节点__、__边__和__属性__。
- 一个 RDF 数据集中有一个__默认图__以及零个或多个__命名图__。
- Neo4j 实现节点和边快速定位的一个关键之处是采取了__定长记录__的存储方案。

三、简答题

- 请简要说明 RDF 图和属性图的主流查询语言及其特点是什么。

答: RDF 图的标准查询语言是 SPARQL。SPARQL 是一种专门为 RDF 数据模型设计的查询语言,支持对 RDF 图进行复杂的查询、更新和分析。属性图的主流查询语言包括 Cypher 和 Gremlin。

(1)Cypher 是 Neo4j 图数据库的声明式查询语言,语法直观,适合属性图的查询和操作。

(2)Gremlin 是一种图遍历语言,支持多种图数据库,适用于复杂的图遍历和操作。

这两种查询语言分别针对不同的数据模型,SPARQL 适用于 RDF 图,而 Cypher 和 Gremlin 适用于属性图。

- 分别使用 SPARQL、Cypher 和 Gremlin 语言查询指挥“四渡赤水”战役的指挥官。

(1) SPARQL 查询:

```
SELECT ?name ?type
WHERE {
```

```

?battle rdfs:label "四渡赤水" .
?person ex:command ?battle .
?person rdfs:label ?name .
?person a ?type .
OPTIONAL { ?person a ex:Commander } .
}

```

(2) Cypher 查询:

```

MATCH (b:Battle {name: "四渡赤水"})
MATCH (p:Person)-[c:COMMAND]->(b)
OPTIONAL MATCH (p)-[e:EXECUTE]->()
RETURN p.name AS commander, e

```

(3) Gremlin 查询:

```

g.V().has("name", "四渡赤水").inE("COMMAND").as("e").outV().as("p")
.select("p", "e").by("name")

```

3. 知识图谱数据库的核心组成部分有哪些，请分别简述其用途。

答：知识图谱数据库的核心组成部分，包括存储管理器、查询处理引擎和数据访问接口。存储管理器负责数据的持久化存储，通过优化的数据结构和索引机制，支持对大规模知识图谱数据的高效存取。查询处理引擎则处理用户的查询请求，生成查询执行计划，高效率执行查询算法，以提供精确的查询结果。数据访问接口为开发者提供与知识图谱数据库交互的手段，通过这些接口，用户可以方便地与数据库进行各类交互。知识图谱数据库是实现知识存储与查询的重要基础设施。

4. 请简述 Neo4j 数据库的基本文件结构。

答：

- (1) 节点存储文件 (node store)：每个节点在创建时分配一个独立的内部 ID；节点存储文件中记录节点的基本元信息，如节点 ID、指向关系记录的起始指针、指向属性记录的指针、节点标签信息等。
- (2) 关系存储文件 (relationship store)：每条关系同样有一个唯一 ID；在关系存储文件中，记录了起始节点 ID、终止节点 ID、关系类型、关系属性指针等信息。关系文件也维护指向上一条或下一条关系的指针，以便快速遍历一条链上的关系。
- (3) 属性存储文件 (property store)：无论是节点，还是关系，其属性都会存放在属性存储文件里，通过某个属性记录指针与节点存储文件和关系存储文件相连。属性通常以(属性键 ID, 类型, 属性值) 形式存储。
- (4) 标签/索引存储文件 (label/index store)：节点标签 (label) 也通过一个 ID 记录在相应的标签存储文件里，节点记录中会引用该标签 ID；若在某些属性上创建了索引，会给这部分属性建立 B+树或类似索引文件，以加速按属性值的查询。

5.4.1 本章习题

以下习题结合了本章“知识存储与查询”的主要内容，分为选择题、填空题和简答题三种类型。请读者根据本章内容进行解答。

一、选择题

1. 下列关于 RDF 图与属性图的说法，哪一项是正确的？
 - A. RDF 图掌握在数据库领域，属性图仅用于语义万维网领域。
 - B. RDF 图以三元组为基本单元，属性图则以节点、边及属性为主要构造要素。
 - C. RDF 图可以天然支持节点和边上的多属性，属性图只能表示三元组。
 - D. RDF 图没有可扩展性，属性图更适用于局部知识存储。
2. 以下关于 SPARQL 和 Cypher 语言的比较，哪一项是正确的？
 - A. SPARQL 专门用于属性图操作，Cypher 用于 RDF 图。
 - B. SPARQL 与 Cypher 都无法进行数据更新操作。
 - C. SPARQL 使用三元组模式匹配数据，Cypher 使用图形模式（ASCII 艺术表示法）匹配数据。

- D. SPARQL 和 Cypher 必须使用相同的前缀语法才可执行查询。
3. 在 Neo4j 中，下列哪种特性体现了其高效的图遍历能力？
- A. 无需建立任何索引即可检索
 - B. 借助“定长记录”与“无索引邻接”使节点直接持有邻居的物理指针
 - C. 所有属性都存放在同一个文件中，可进行全文检索
 - D. 仅支持点对点查询，无法进行多跳查询
4. 以下关于 Gremlin 语言的叙述，哪一项是错误的？
- A. Gremlin 是一种过程式图遍历语言。
 - B. Gremlin 提供了 `out()`、`in()` 等步骤用于节点间的导航。
 - C. Gremlin 只能操作属性图，无法更新属性图数据。
 - D. Gremlin 可在多种图数据库上执行，包括 JanusGraph 和 Apache TinkerPop 生态系统。
-

二、填空题

1. SPARQL 中用 _____（关键字）可以将多个数据源的查询合并起来，使得一个 SPARQL 查询可访问多个远程端点的数据。

2. 在 RDF 数据模型中，若要表示一个实体在同一条边（关系）上附加属性，可采用“_____（英文）”的方法，将该三元组本身具体化为一个中间资源，并把附加信息加到这个中间资源上。
 3. 在属性图 Cypher 语言中，节点用圆括号“()”来表示，如果节点有标签，则在括号里通过“_____”符号指定标签，如 (n:Person)。
 4. Neo4j 中，为了实现图数据的高效存储和遍历，其节点存储文件会采用“_____”形式记录节点，使得按节点 ID 可以直接访问到对应的存储位置。
 5. SPARQL 查询语句中用于可选匹配的关键字是“_____”，而与之类似，Cypher 语言中则使用“OPTIONAL MATCH”实现类似功能。
-

三、简答题

1. 结合本章内容，简述 RDF 图与属性图在数据建模和关系属性表达方面的主要差异。
2. 简要说明在 RDF 图中如何表示关系上的属性，并举例说明采用了哪种方法（或技术）来实现。
3. Cypher 与 SPARQL 在查询中的“模式并集”功能分别使用了什么关键字？请分别举一个示例。

4. 在属性图数据库 Neo4j 中，查询处理一般要经历哪些步骤？可结合 “MATCH (p:Person)-[r:COMMAND]->(b:Battle {name: '四渡赤水'}) RETURN ...” 示例说明。
-

5.4.2 参考答案

一、选择题

1. B
 - 解析：RDF 图以三元组(s, p, o)为基础，常用于语义万维网；属性图使用节点、边及属性键值对描述数据，是工业界常用的图数据库模型。
 2. C
 - 解析：SPARQL 基于三元组模式匹配，Cypher 基于可视化风格（ASCII 艺术）描述的图模式来匹配节点和边。
 3. B
 - 解析：Neo4j 采用“定长记录”与“无索引邻接”特性，使得每个节点可直接持有相邻节点（及关系）的存储指针，提升了多跳图遍历性能。
 4. C
 - 解析：Gremlin 不仅可以查询属性图数据，也能对图数据进行更新操作；它是一个可执行“查询-分析-更新”全流程的过程式语言。
-

二、填空题

1. SERVICE
 - 说明：在 SPARQL 中使用 SERVICE { ... } 进行联邦查询，将多个远程端点数据源的查询合并到一个 SPARQL 查询中。

2. reification

- 说明：RDF 中可通过“具体化”方法（reification）给某个三元组赋予一个空节点或新的 IRI，并在该空节点/IRI 上关联其他属性。

3. 冒号 “:”

- 说明：在 Cypher 语法中对节点和边指定标签或类型时使用冒号，例如 (v:Battle), (p:Person), (p)-[:COMMAND]->(b)。

4. 定长记录

- 说明：在 Neo4j 中，节点存储、关系存储均采用固定长度的记录结构，可根据节点或关系 ID 直接计算所在文件的偏移量。

5. OPTIONAL

- 说明：SPARQL 语言使用 OPTIONAL { ... } 作为可选匹配；与之类似，Cypher 语言则是“OPTIONAL MATCH”。

三、简答题

1. 答：

- RDF 图以三元组为基本结构，用(s, p, o)三元组表示实体与实体、实体与属性值之间的关系；关系上的属性需使用 reification 或空节点等方式表示。
- 属性图直接在节点及边上附加属性，能够天然支持关系上的属性；且节点/边还可带有标签，便于分类和快速检索。

2. 答：

- 在 RDF 图中，如果想在主语—谓语—宾语这条边（关系）上再附加额外信息，需要使用“具体化”（reification）方法。
- 例如，要为“毛泽东指挥四渡赤水”这条三元组添加“角色为战略指挥”属性，可先创建一个新资源 :t1，声明:t1 rdf:subject ex:maoZedong, rdf:predicate ex:command, rdf:object ex:crossing，然后将 role 这一属性附加到 _:t1 上。

3. 答：

- SPARQL 中使用 UNION 实现模式并集，比如：
`{ ?s a ex:Battle } UNION { ?s a ex:Campaign }。`
- Cypher 中使用 UNION 关键字将多个 MATCH 查询的结果合并：
`MATCH (b:Battle) RETURN b.name UNION MATCH (c:Campaign) RETURN c.name。`

4. 答：

- 在 Neo4j 中一次查询处理的大致步骤包括：
 - (1) 解析：将 Cypher 查询解析为语法树；
 - (2) 优化：分析标签、属性索引，以及连接顺序等，生成最佳执行计划；
 - (3) 执行：根据执行计划，先使用索引定位起始节点或属性，再进行关系扩展（expand）或过滤，得到结果；
 - (4) 返回结果：将查询匹配的节点/关系或属性值返回给用户。
- 例如，“`MATCH (p:Person)-[r:COMMAND]->(b:Battle {name:'四渡赤水'}) RETURN ...`”会先根据标签(:Battle)与属性(name='四渡赤水')索引锁定战役节点，再沿 COMMAND 边扩展到人物节点 p，最终返回所需的信息。

请读者根据本章的相关内容，结合自身理解进行更加深入的分析与思考。通过上述习题，可以进一步巩固对“知识存储与查询”相关概念与技术的掌握。