法律声明

- 本课件包括：演示文稿，示例，代码，题库，视频和声音等，小象学院和讲者共同拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意，我们将保留一切通过法律手段追究违反者的权利。

- 课程详情请咨询
  - 微信公众号：小象
  - 新浪微博：ChinaHadoop

第五讲　　知识存储

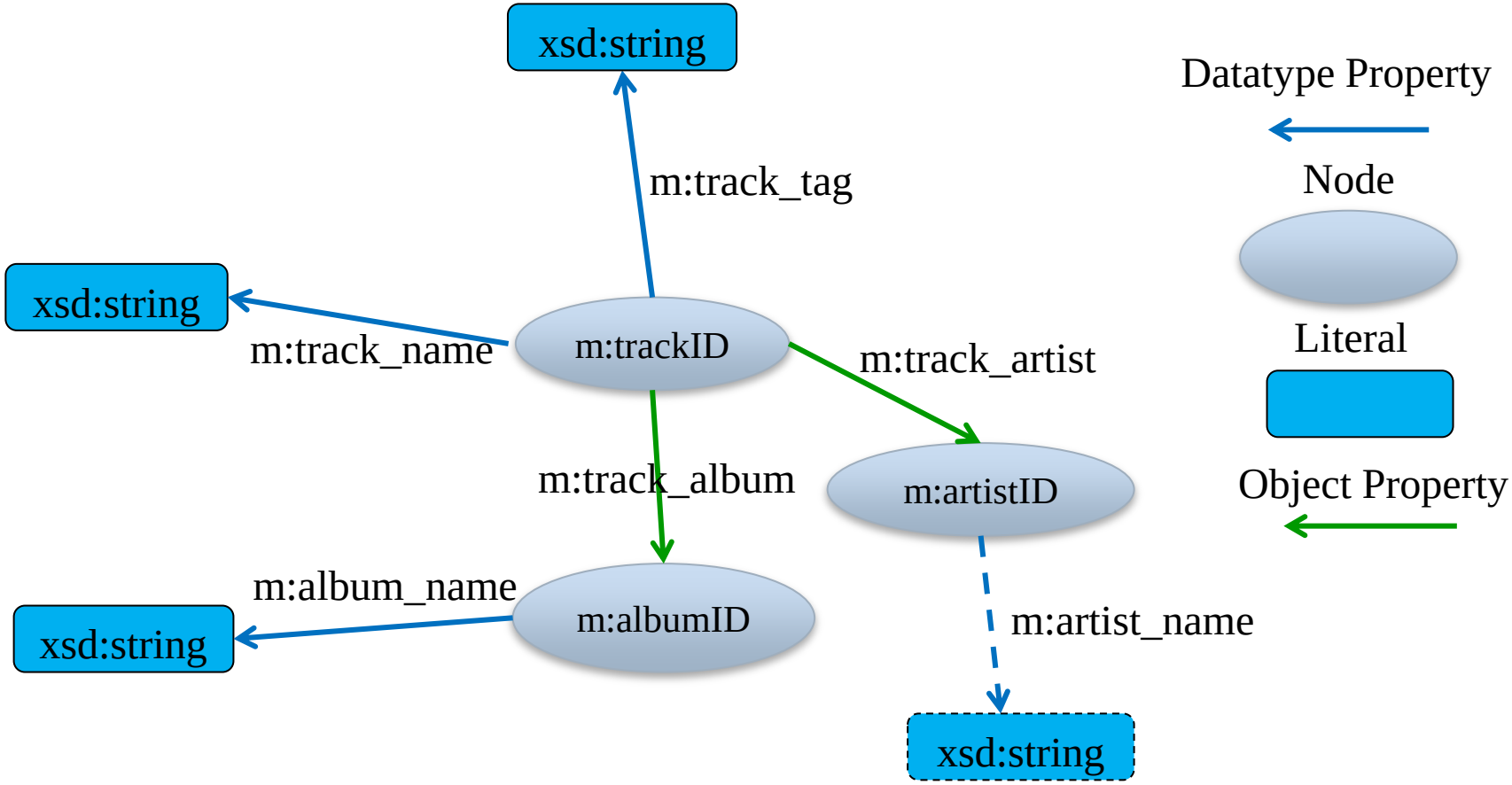
大纲

- 从一个例子开始
  - 数据来源
  - 数据描述
  - 数据导入
  - 数据查询
  - 数据更新
- 图数据库介绍
  - 开源数据库介绍：RDF4j 、gStore等
  - 商业数据库介绍：Virtuoso、AllegroGraph、Stardog等
  - 原生图数据库介绍：Neo4j、OrientDB、Titan等
- 图数据库实现细节

第一部分:从一个例子开始

音乐知识图谱Schema定义

PREFIX m: <http://kg.course/music/>



图谱数据生成

示例数据为Python代码随机生成

以下为生成的数据部分截图

```
<http://kg.course/music/track_00001>  <http://kg.course/music/track_name>  "track_name_00001" .
<http://kg.course/music/track_00001>  <http://kg.course/music/track_album>  <http://kg.course/music/album_0001> .
 <http://kg.course/music/album_0001>  <http://kg.course/music/album_name>  "album_name_0001" .
<http://kg.course/music/track_00001>  <http://kg.course/music/track_artist>  <http://kg.course/music/artist_001> .
<http://kg.course/music/track_00001>  <http://kg.course/music/track_tag>  "tag_name_02" .
<http://kg.course/music/track_00002>  <http://kg.course/music/track_name>  "track_name_00002" .
<http://kg.course/music/track_00002>  <http://kg.course/music/track_album>  <http://kg.course/music/album_0001> .
 <http://kg.course/music/album_0001>  <http://kg.course/music/album_name>  "album_name_0001" .
<http://kg.course/music/track_00002>  <http://kg.course/music/track_artist>  <http://kg.course/music/artist_002> .
<http://kg.course/music/track_00002>  <http://kg.course/music/track_tag>  "tag_name_03" .
<http://kg.course/music/track_00003>  <http://kg.course/music/track_name>  "track_name_00003" .
<http://kg.course/music/track_00003>  <http://kg.course/music/track_album>  <http://kg.course/music/album_0004> .
 <http://kg.course/music/album_0001>  <http://kg.course/music/album_name>  "album_name_0001" .
<http://kg.course/music/track_00003>  <http://kg.course/music/track_artist>  <http://kg.course/music/artist_003> .
<http://kg.course/music/track_00003>  <http://kg.course/music/track_tag>  "tag_name_04" .
<http://kg.course/music/track_00004>  <http://kg.course/music/track_name>  "track_name_00004" .
<http://kg.course/music/track_00004>  <http://kg.course/music/track_album>  <http://kg.course/music/album_0003> .
 <http://kg.course/music/album_0001>  <http://kg.course/music/album_name>  "album_name_0001" .
<http://kg.course/music/track_00004>  <http://kg.course/music/track_artist>  <http://kg.course/music/artist_004> .
<http://kg.course/music/track_00004>  <http://kg.course/music/track_tag>  "tag_name_01" .
```

为了演示方便，生成的数据中没有artist_name这个属性，后面演示通过SPARQL Update语句添加

音乐图谱数据生成器

Python脚本文件为： kg_music_triples.py

例子：python kg_music_triples.py

默认生成1000个三元组 生成的文件在当前目录，文件名为music_1000_triples.nt

例子：python kg_music_triples.py 500

可传入参数，比如传入500，
会在当前目录生成 music_500_triples.nt

生成了nt文件如下图：

```
➜ 程序 git:(master) ✗ python kg_music_triples.py 500
➜ 程序 git:(master) ✗ ls music_500_triples.nt
music_500_triples.nt
```

图谱存储工具 – 图数据库

图数据库

图数据库源起欧拉和图理论(graph theory)，也可称为面向/基于图的数据库，对应的英文是Graph Database。图数据库的基本含义是以“图”这种数据结构存储和查询数据。它的数据模型主要是以节点和关系(边)来体现，也可处理键值对。它的优点是快速解决复杂的关系问题。

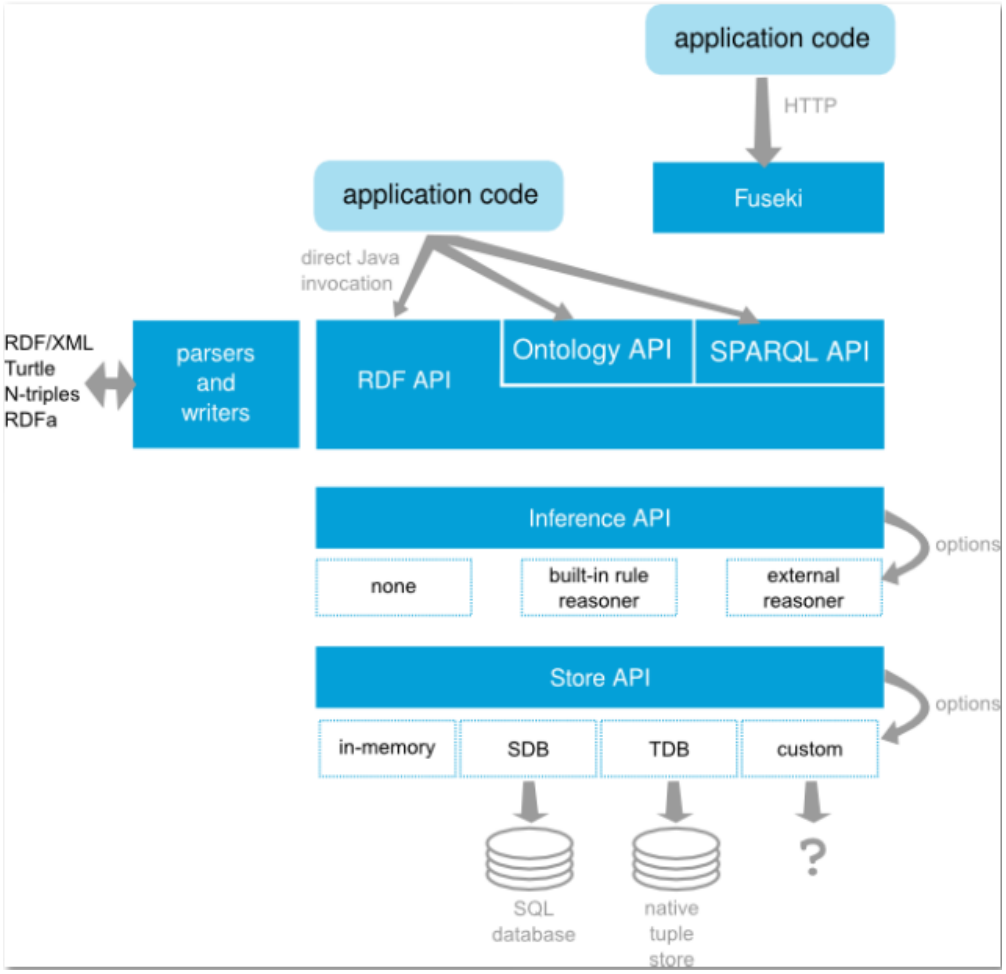
图具有如下特征：

1.包含节点和边
2.节点上有属性(键值对)
3.边有名字和方向，并总是有一个开始节点和一个结束节点
4.边也可以有属性

https://en.wikipedia.org/wiki/Graph_database

开源数据库 – Apache Jena

- 一个免费开源的支持构建语义网络和数据链接应用的 Java 框架
- 由惠普实验室开发
- 支持内存和永久存储



http://jena.apache.org

开源数据库 – Apache Jena

数据导入方法：

1.Fuseki 手动导入(稍后演示)

2.使用TDB导入

使用TDB导入的命令如下

/jena-fuseki/tdbloader --loc=/jena-fuseki/data filename

Fuseki启动的命令如下，需要指定tdb生成的文件路径并指定数据库名

/jena-fuseki/fuseki-server --loc=/jena-fuseki/data --update /music

```
data tree
.
├── GOSP.dat
├── GOSP.idn
├── GPOS.dat
├── GPOS.idn
├── GSPO.dat
├── GSPO.idn
├── journal.jrnl
├── node2id.dat
├── node2id.idn
├── nodes.dat
├── nodes.dat-jrnl
├── OSP.dat
├── OSPG.dat
├── OSPG.idn
├── OSP.idn
├── POS.dat
├── POSG.dat
├── POSG.idn
├── POS.idn
├── prefix2id.dat
├── prefix2id.idn
├── prefixes.dat
├── prefixes.dat-jrnl
├── prefixIdx.dat
├── prefixIdx.idn
├── SPO.dat
├── SPOG.dat
├── SPOG.idn
├── SPO.idn
├── stats.opt
└── tdb.lock
```

完成导入之后/jena-fuseki/data的文件结构

开源数据库 – Apache Jena

查询

1.Fuseki 界面查询 (稍后演示)

2.使用endpoint接口查询

Endpoint地址：

SPARQL Query: http://localhost:3030/music/query

SPARQL Update: http://localhost:3030/music/update

更多内容请查询Fuseki文档

开源数据库 – Apache Jena

- Python操作Apache Jena
  - 使用 Jena SPARQL endpoint 接口进行查询和更新
  - 使用SPARQLWrapper包查询和更新

```
def query(query_string=None):

    sparql_query = SPARQLWrapper(query_url)
    sparql_query.setQuery(query_string)
    sparql_query.setReturnFormat(JSON)

    results = None
    try:
        results = sparql_query.query().convert()

    except:
        return False

    return results
```

```
def update(update_string):

    sparql_update = SPARQLWrapper( updateEndpoint=update_url)

    sparql_update.setQuery(update_string)
    sparql_update.method = "POST"
    sparql_update.setReturnFormat(JSON)

    results = None
    try:
        results = sparql_update.query()
    except:
        return False

    if "Success" in results.response.read():
        return True

    return False
```

https://rdflib.github.io/sparqlwrapper/

安装Apache Jena

使用Docker来运行Jena

Docker 安装请参考 docker.com

1.从仓库pull jena-fuseki镜像

```
➜  ~ git:(master) ✗ docker pull stain/jena-fuseki
Using default tag: latest
latest: Pulling from stain/jena-fuseki
709515475419: Pull complete
38a1c0aaa6fd: Pull complete
cd134db5e982: Downloading [==================================================>       ] 34.27 MB/39.68 MB
23b3641459b2: Download complete
05dacbea9031: Download complete
0b04716f8e40: Downloading [======>                                            ] 2.711 MB/24.54 MB
5df178e45afd: Download complete
1026bcbdbe0b: Download complete
2000d807ddcc: Download complete
d385f82d5407: Download complete
0a95dc5fc82c: Download complete
63ae8bd2ac3e: Waiting
```

安装Apache Jena

镜像 pull完成：

```
➜  ~ git:(master) ✗ docker pull stain/jena-fuseki
Using default tag: latest
latest: Pulling from stain/jena-fuseki
709515475419: Pull complete
38a1c0aaa6fd: Pull complete
cd134db5e982: Pull complete
23b3641459b2: Pull complete
05dacbea9031: Pull complete
0b04716f8e40: Pull complete
5df178e45afd: Pull complete
1026bcbdbe0b: Pull complete
2000d807ddcc: Pull complete
d385f82d5407: Pull complete
0a95dc5fc82c: Pull complete
63ae8bd2ac3e: Pull complete
Digest: sha256:e85d1a5afc71309f855fd7a6c7973831f95e4d1f15e6475bca60576060ba1361
Status: Downloaded newer image for stain/jena-fuseki:latest
```

启动服务

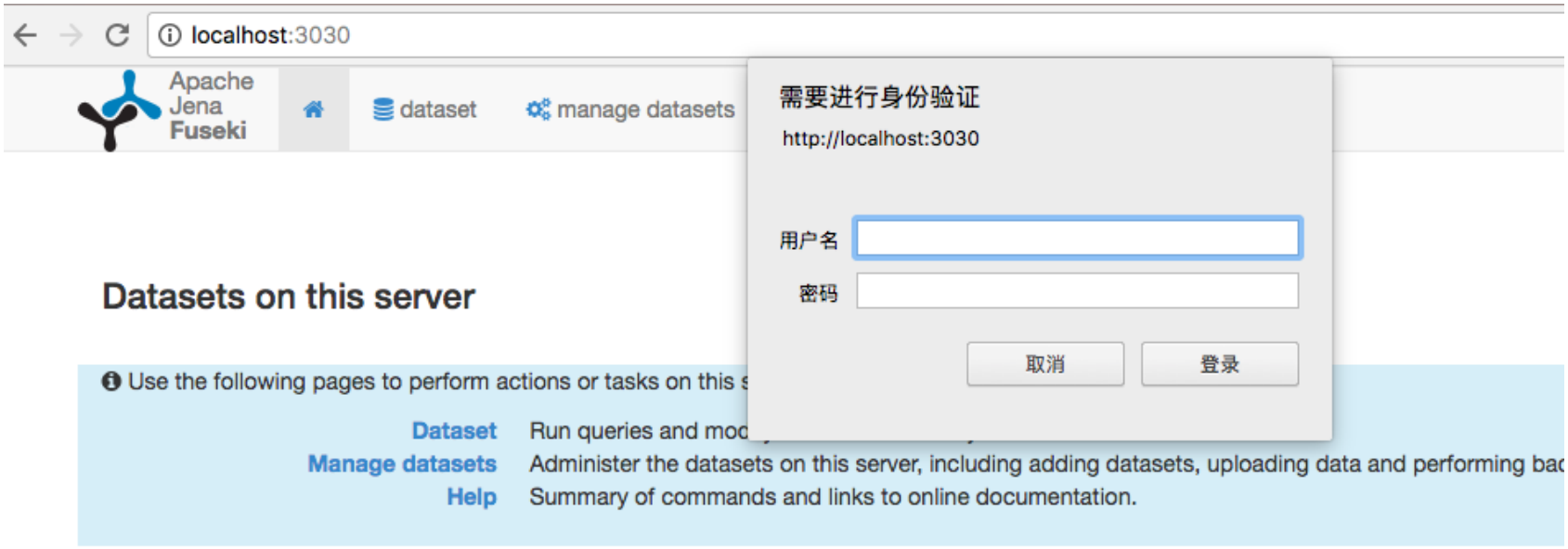
2.启动Jena-fuseki

```
➜  ~ git:(master) ✗ docker run -d -p 3030:3030 -e "ADMIN_PASSWORD=test@jena"  stain/jena-fuseki
1c8aa095ec432f1341e0358e9512db3aaa0004649b99a74760763299dda86f13
```

docker run -d -p 3030:3030 -e "ADMIN_PASSWORD=test@jena" stain/jena-fuseki

端口为： 3030
admin的密码为： test@jena

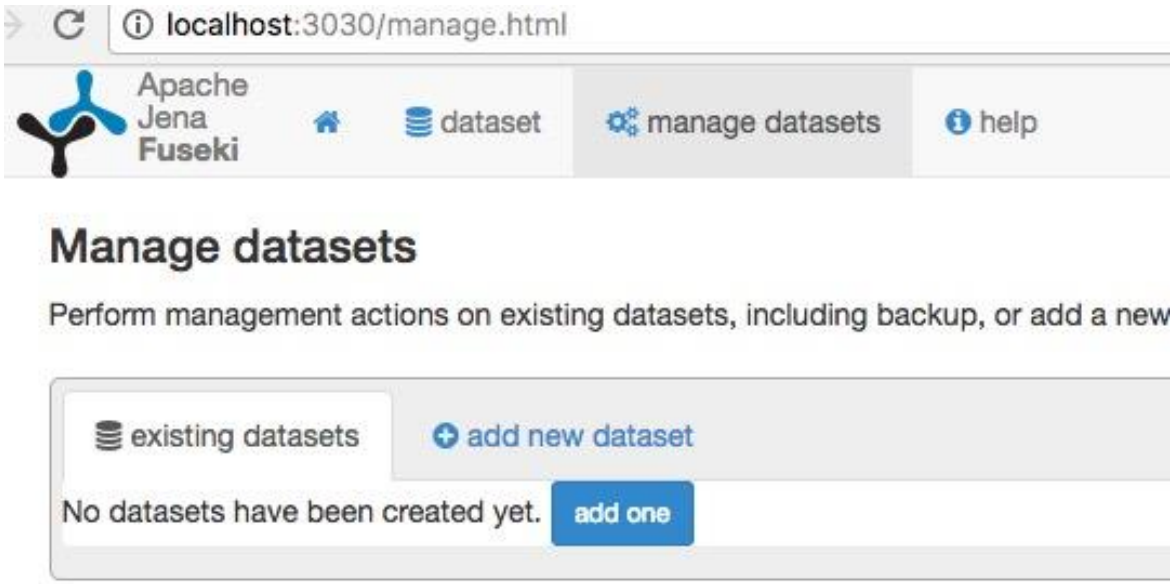
Fuseki访问界面



3.使用浏览器打开localhost:3030

用户名：admin
密码： test@jena

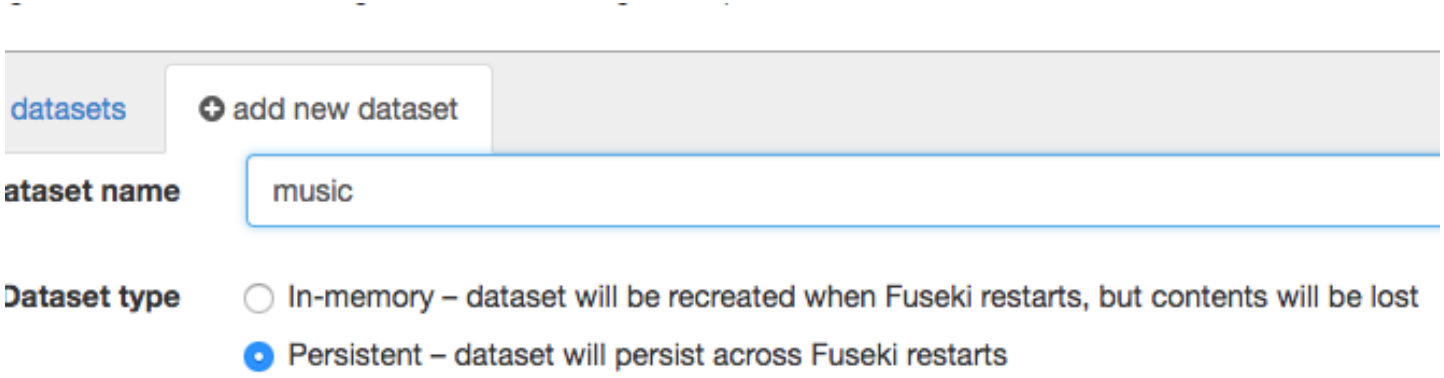
增加数据库



点击 manage datasets

点击 add one 增加一个数据库

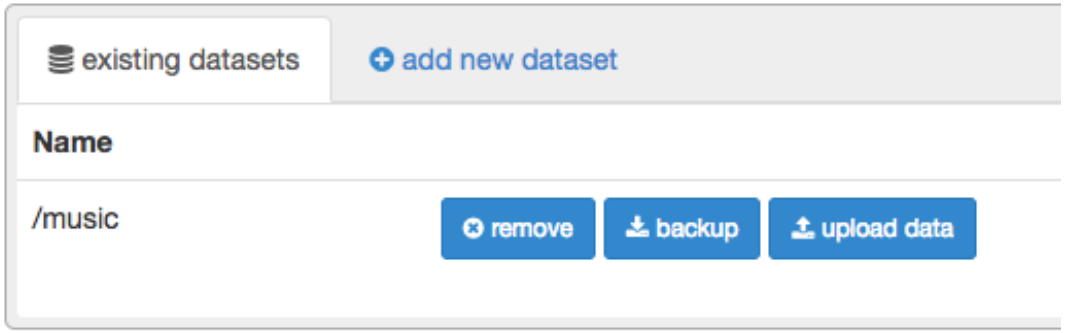
创建数据库



输入dataset name为: music
选择 Persistent

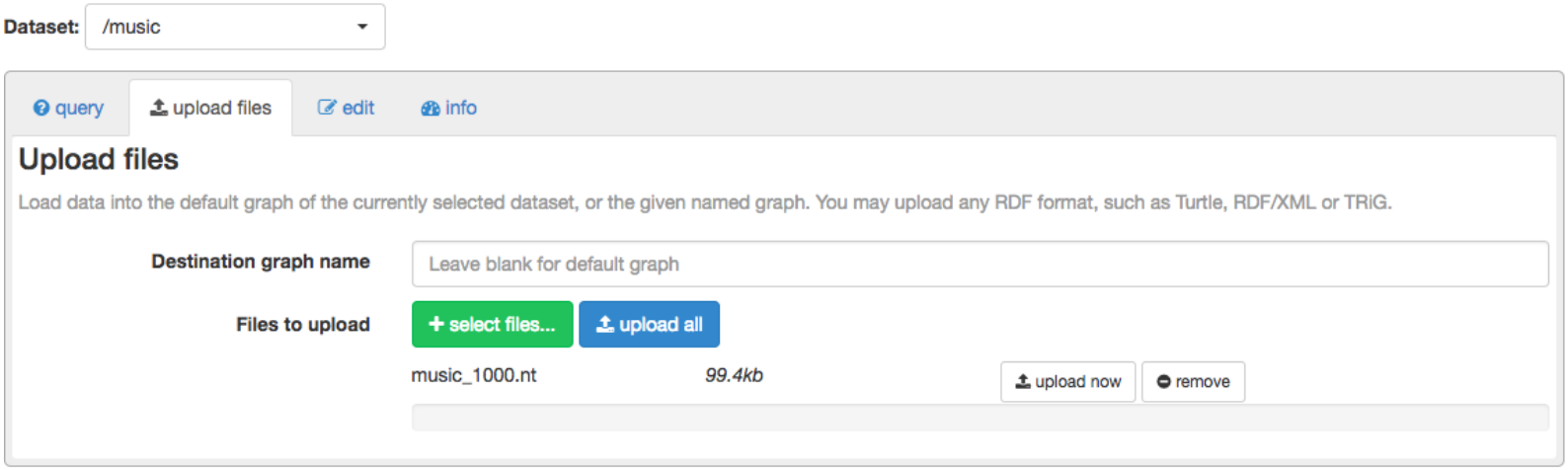
点击 create dataset 完成数据库的创建

导入数据



点击最右的upload data会出现导入数据的界面

导入数据

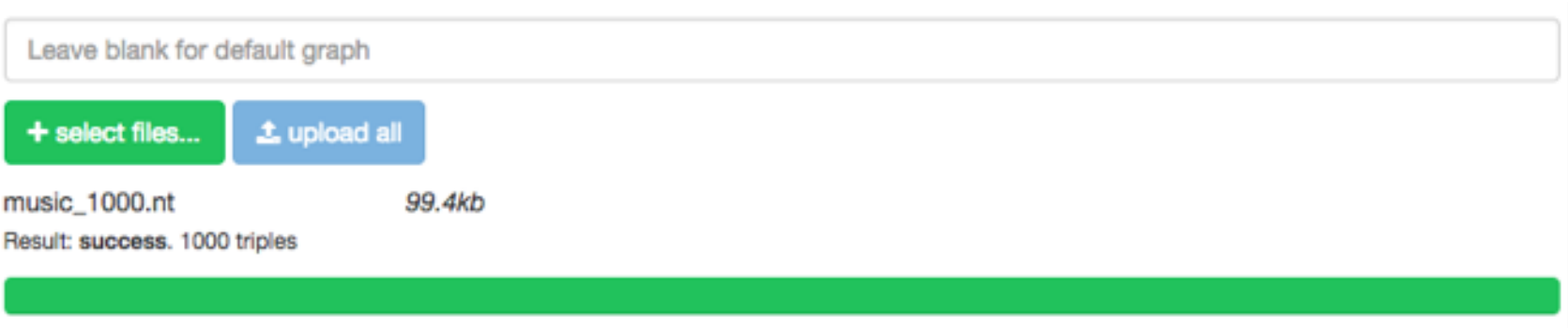


选择导入的nt文件

点击 upload all 完成数据导入

Jena支持任意的RDF格式导入

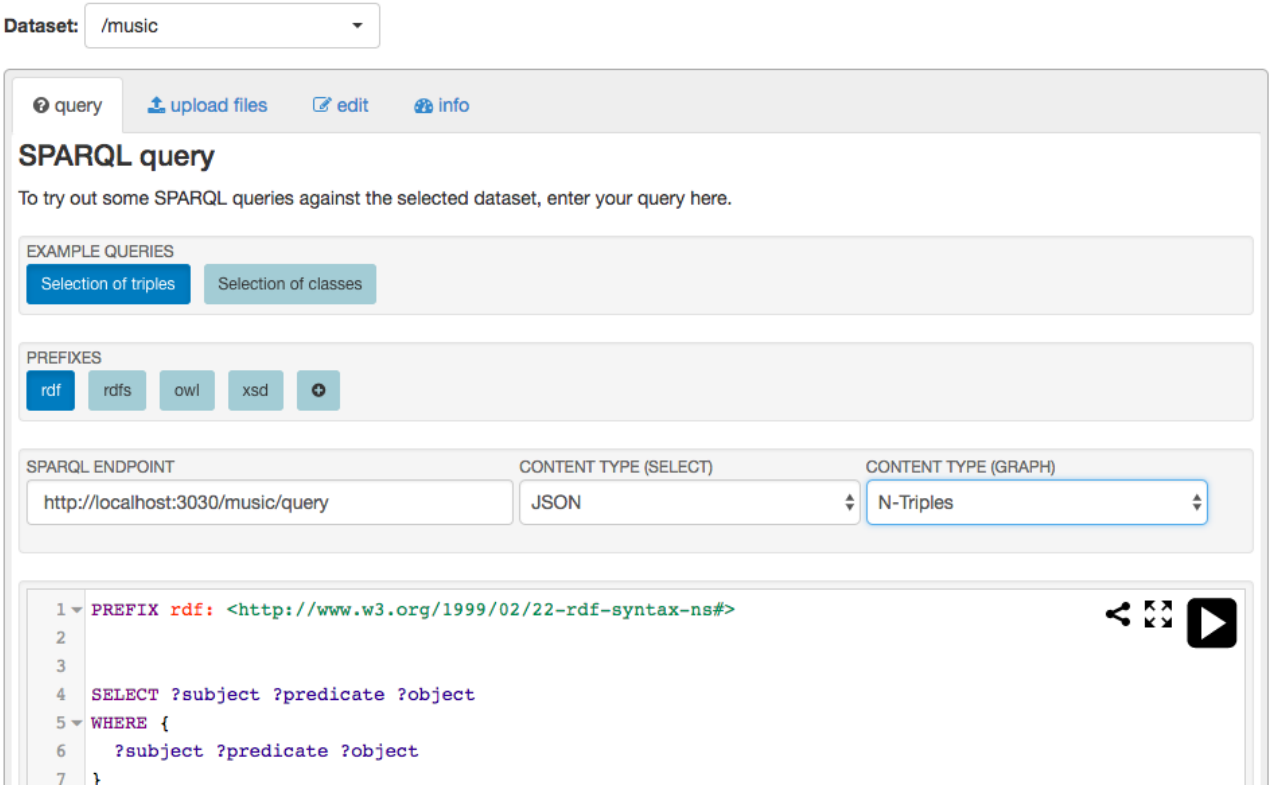
导入数据



显示导入成功，并可以查看导入了多少的三元组

```
bash-4.3# ls
GOSP.dat        GSPO.dat        OSPG.dat        POSG.dat        SPOG.dat        node2id.idn     prefix2id.idn   prefixes.dat-jrnl
GOSP.idn        GSPO.idn        OSPG.idn        POSG.idn        SPOG.idn        nodes.dat       prefixIdx.dat   tdb.lock
GPOS.dat        OSP.dat         POS.dat         SPO.dat         journal.jrnl    nodes.dat-jrnl  prefixIdx.idn
GPOS.idn        OSP.idn         POS.idn         SPO.idn         node2id.dat     prefix2id.dat   prefixes.dat
bash-4.3# pwd
/fuseki/databases/music
```

准备查询



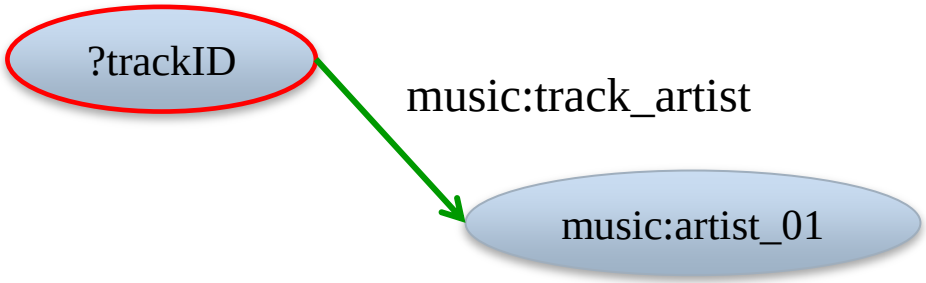
点击 query，来到SPARQL 查询界面，选择对应的设置，我们就可以开始查询图谱了

在中间的编辑器中国添加我们自己的SPARQL语句，然后点击编辑器右上的执行按钮就可以执行SPARQL查询了

查询例子

查询某一歌手唱的所有歌曲

```
SELECT  DISTINCT   ?trackID
WHERE {
 ?trackID  track_artist  artistID
}
```




```
1  PREFIX music: <http://kg.course/music/>
2
3  SELECT  DISTINCT   ?trackID
4  WHERE {
5    ?trackID  music:track_artist  music:artist_01
6  }
7
```

查询结果

可以看到这个艺术家有八首歌曲

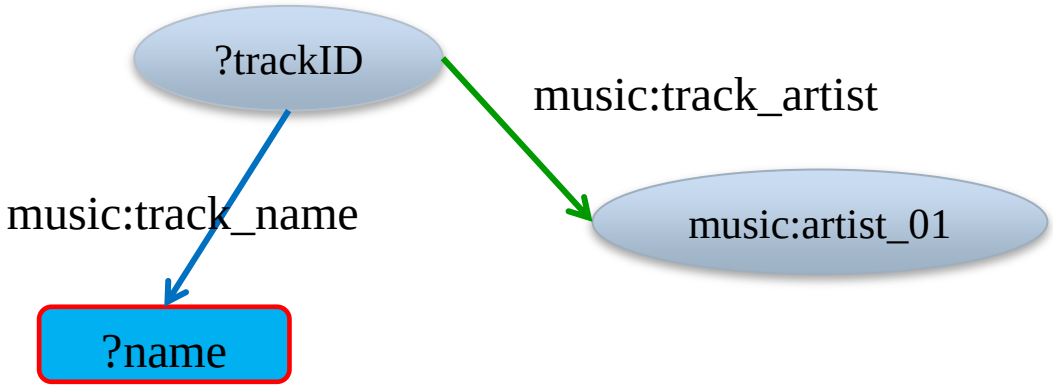
	trackID
1	music:track_00001
2	music:track_00015
3	music:track_00069
4	music:track_00071
5	music:track_00073
6	music:track_00149
7	music:track_00171
8	music:track_00181

Showing 1 to 8 of 8 entries

查询例子

- 如果我想得到某一位歌手所有歌曲的歌曲名，应该怎么办？

```
SELECT ?name
WHERE {
?trackID   track_artist   artistID .
?trackID   track_name  ?name
}
```




```
1  PREFIX music: <http://kg.course/music/>
2
3  SELECT ?name
4  WHERE {
5    ?trackID  music:track_artist music:artist_01  .
6    ?trackID  music:track_name   ?name
7   }
```

查询结果

	name
1	"track_name_00001"
2	"track_name_00015"
3	"track_name_00069"
4	"track_name_00071"
5	"track_name_00073"
6	"track_name_00149"
7	"track_name_00171"
8	"track_name_00181"

Showing 1 to 8 of 8 entries

只显示歌曲名

	trackID	name
1	music:track_00001	"track_name_00001"
2	music:track_00015	"track_name_00015"
3	music:track_00069	"track_name_00069"
4	music:track_00071	"track_name_00071"
5	music:track_00073	"track_name_00073"
6	music:track_00149	"track_name_00149"
7	music:track_00171	"track_name_00171"
8	music:track_00181	"track_name_00181"

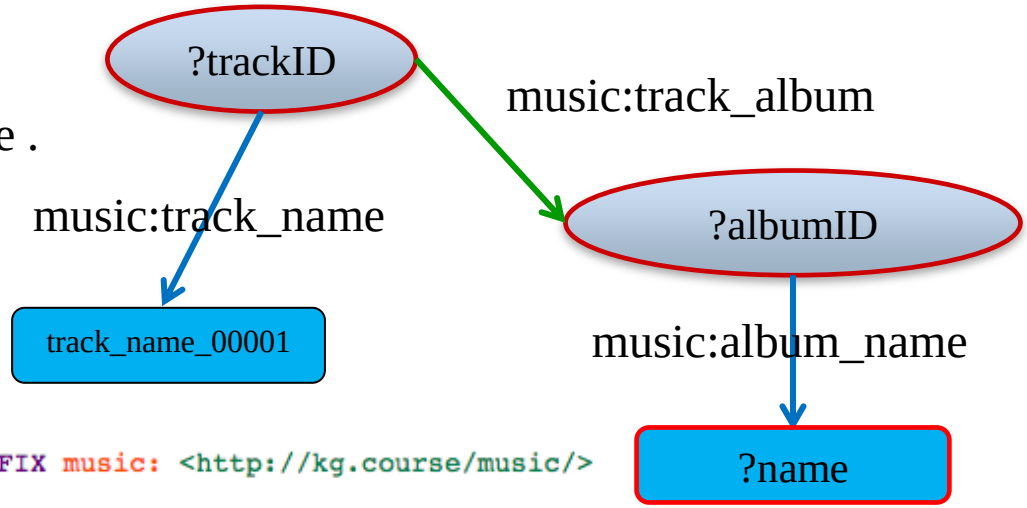
Showing 1 to 8 of 8 entries

把歌曲id也显示出来，可以看见都一一对应

查询例子

查询某一首歌曲名对应的专辑信息

```
SELECT ?name
WHERE {
 ?trackID   track_name    track_name .
 ?trackID   track_album   ?albumID .
 ?albumID   album_name    ?name
}
```



歌曲名为歌曲的属性，需要查出歌曲id，再查询属性专辑信息

```
PREFIX music: <http://kg.course/music/>

SELECT ?trackID ?albumID ?name
WHERE {
 ?trackID    music:track_name    "track_name_00001"  .
 ?trackID    music:track_album   ?albumID .
 ?albumID    music:album_name    ?name
}
```

查询结果

	name
1	"album_name_0001"

Showing 1 to 1 of 1 entries

只显示专辑名

	trackID	albumID	name
1	music:track_00001	music:album_0001	"album_name_0001"

Showing 1 to 1 of 1 entries

显示歌曲id 专辑id 和专辑名

```
1  PREFIX music: <http://kg.course/music/>
2
3  SELECT ?歌曲id ?专辑id ?专辑名
4  WHERE {
5    ?歌曲id  music:track_name   "track_name_00001"   .
6    ?歌曲id  music:track_album  ?专辑id  .
7    ?专辑id  music:album_name   ?专辑名
8  }
9
```

Fuseki支持使用中文定义SELECT变量

QUERY RESULTS

Table | Raw Response

Showing 1 to 1 of 1 entries Search: Show 50 entries

	歌曲id	专辑id	专辑名
1	music:track_00001	music:album_0001	"album_name_0001"

查询例子

- 我想在专辑名前面加一些描述，应该怎么办？

```
SELECT ?歌曲id ?专辑id (CONCAT("专辑名",":",?专辑名) AS ?专辑信息)
WHERE {
  ?歌曲id track_name  track_name  .
  ?歌曲id track_album ?专辑id .
  ?专辑id album_name ?专辑名
}
```

使用CONCAT连接

```
PREFIX music: <http://kg.course/music/>

SELECT ?歌曲id ?专辑id (CONCAT("专辑名",":",?专辑名) AS ?专辑信息)
WHERE {
  ?歌曲id  music:track_name   "track_name_00001"   .
  ?歌曲id  music:track_album  ?专辑id  .
  ?专辑id  music:album_name   ?专辑名
}
```

QUERY RESULTS

Table | Raw Response

Showing 1 to 1 of 1 entries Search: Show 50 entries

	歌曲id	专辑id	专辑信息
1	music:track_00001	music:album_0001	"专辑名:album_name_0001"

查询例子

查询某个专辑里面的所有歌曲

```
SELECT  ?trackID
WHERE {
  ?albumID       album_name  album_name  .
  ?trackID       track_album   ?albumID
}
```

```
PREFIX music: <http://kg.course/music/>

SELECT  ?trackID
WHERE {
  ?albumID    music:album_name   "album_name_0002"   .
  ?trackID    music:track_album   ?albumID
}
```

查询结果

	trackID
1	music:track_00003
2	music:track_00005
3	music:track_00006
4	music:track_00007
5	music:track_00008
6	music:track_00010
7	music:track_00011
8	music:track_00012
9	music:track_00013

Showing 1 to 9 of 9 entries

Showing 1 to 9 of 9 entries Search: Show 50 entries

	albumID	trackID
1	music:album_0002	music:track_00003
2	music:album_0002	music:track_00005
3	music:album_0002	music:track_00006
4	music:album_0002	music:track_00007
5	music:album_0002	music:track_00008
6	music:album_0002	music:track_00010
7	music:album_0002	music:track_00011
8	music:album_0002	music:track_00012
9	music:album_0002	music:track_00013

Showing 1 to 9 of 9 entries

显示专辑id和歌曲id的结果

如果要查询某个专辑里面的前2首歌曲呢?

查询例子

```
PREFIX music: <http://kg.course/music/>

SELECT  ?albumID ?trackID
WHERE {
  ?albumID    music:album_name    "album_name_0002"  .
  ?trackID    music:track_album   ?albumID
}

LIMIT 2
```

QUERY RESULTS

Table | Raw Response

Showing 1 to 2 of 2 entries Search: Show 50 entries

	albumID	trackID
1	music:album_0002	music:track_00003
2	music:album_0002	music:track_00005

Showing 1 to 2 of 2 entries

使用Limit 限制查询结果的条数

查询结果

计算某一个专辑的歌曲数目呢?

```
SELECT (COUNT(?trackID) AS ?num)
WHERE {
 ?albumID   album_name    album_name  .
 ?trackID     track_album     ?albumID
}
```

使用COUNT进行计数

```
1  PREFIX music: <http://kg.course/music/>
2
3  SELECT  (COUNT(?trackID) as ?num)
4  WHERE {
5    ?albumID     music:album_name    "album_name_0002"  .
6    ?trackID     music:track_album    ?albumID
7  }
8
```

QUERY RESULTS

Table | Raw Response

Showing 1 to 1 of 1 entries Search: Show 50 entries

	num
1	"9"^^xsd:integer

"9"^^xsd:integer integer 为数据类型

查询例子

查询某一首歌是哪一个歌手的作品

```
SELECT ?trackID  ?artistID
WHERE {
?trackID track_name  track_name .
?trackID track_artist  ?artistID
}
```

```
PREFIX music: <http://kg.course/music/>

SELECT ?trackID  ?artistID
WHERE {
?trackID  music:track_name   "track_name_00001"  .
?trackID music:track_artist ?artistID
}
```

QUERY RESULTS

Table | Raw Response

Showing 1 to 1 of 1 entries Search: Show 50 entries

	trackID	artistID
1	music:track_00001	music:artist_001

查询例子

查询某一首歌属于什么歌曲标签

```
SELECT  ?tag_name
WHERE {
?trackID  track_name  track_name .
?trackID  track_tag     ?tag_name
}
```

```
PREFIX music: <http://kg.course/music/>

SELECT  ?trackID ?tag_name
WHERE {
?trackID  music:track_name   "track_name_00001" .
?trackID  music:track_tag    ?tag_name
}
```

QUERY RESULTS

Table | Raw Response

Showing 1 to 1 of 1 entries Search: Show 50 entries

	trackID	tag_name
1	music:track_00001	"tag_name_02"

显示歌曲id和标签的结果

查询例子

查询某一歌手唱过歌曲的所有标签

```
SELECT  DISTINCT ?tag_name
WHERE {
 ?trackID    track_artist  artistID  .
 ?trackID    track_tag    ?tag_name
 }
```

```
1  PREFIX music: <http://kg.course/music/>
2
3  SELECT    ?tag_name
4  WHERE {
5    ?trackID  music:track_artist  music:artist_001 .
6    ?trackID  music:track_tag     ?tag_name
7  }
8
```

	tag_name
1	"tag_name_02"
2	"tag_name_02"
3	"tag_name_06"
4	"tag_name_05"
5	"tag_name_02"
6	"tag_name_02"
7	"tag_name_02"
8	"tag_name_08"
9	"tag_name_03"
10	"tag_name_02"
11	"tag_name_02"
12	"tag_name_04"

Showing 1 to 12 of 12 entries

直接查询所有的歌曲标签，发现标签有重复

查询例子

```
PREFIX music: <http://kg.course/music/>

SELECT  DISTINCT ?tag_name
WHERE {
 ?trackID    music:track_artist  music:artist_001  .
 ?trackID    music:track_tag    ?tag_name
 }

```

使用DISTINCT 去重

结果如右下图所示

	tag_name
1	"tag_name_02"
2	"tag_name_06"
3	"tag_name_05"
4	"tag_name_08"
5	"tag_name_03"
6	"tag_name_04"

Showing 1 to 6 of 6 entries

另一个需求来了，我需要标签排序

查询例子

```
SELECT  DISTINCT ?tag_name
WHERE {
 ?trackID    track_artist  artistID  .
 ?trackID    track_tag    ?tag_name
 }
```

使用ORDER BY 排序

ORDER BY DESC(?tag_name)

```
1  PREFIX music: <http://kg.course/music/>
2
3  SELECT  DISTINCT ?tag_name
4  WHERE {
5   ?trackID    music:track_artist  music:artist_001  .
6   ?trackID    music:track_tag    ?tag_name
7   }
8
9  ORDER BY ?tag_name
10
```

	tag_name
1	"tag_name_02"
2	"tag_name_03"
3	"tag_name_04"
4	"tag_name_05"
5	"tag_name_06"
6	"tag_name_08"

Showing 1 to 6 of 6 entries

那么ORDER BY DESC (?tag_name) 就是逆排序

查询例子

查询某几类歌曲标签中的歌曲数目

```
SELECT  (COUNT(?trackID ) AS ?num)
WHERE {
  {
    ?trackID track_tag     tag_name .
  }
UNION
  {
   ?trackID track_tag   tag_name2  .
  }
}
```

```
1  PREFIX music: <http://kg.course/music/>
2
3  SELECT  (COUNT(?trackID ) AS ?num)
4  WHERE {
5  {
6   ?trackID  music:track_tag    "tag_name_01" .
7  }
8  UNION
9  {
10    ?trackID music:track_tag    "tag_name_02"   .
11 }
12 }
13
14
```

QUERY RESULTS

Table | Raw Response

Showing 1 to 1 of 1 entries Search:

	num
1	"85"^^xsd:integer

使用UNION联合查询

查询例子

查询某几类歌曲标签中的歌曲数目

```
SELECT  (count(?trackID ) as ?num)
WHERE {
 ?trackID track_tag  ?trag_name
 FILTER (?tag_name =  tag_name1  ||
?tag_name = tag_name2)
 }
```

```
1  PREFIX music: <http://kg.course/music/>
2
3  SELECT  (count(?trackID ) as ?num)
4  WHERE {
5   ?trackID music:track_tag  ?tag_name
6   FILTER (?tag_name = "tag_name_01"|| ?tag_name = "tag_name_02")
7   }
8
9
```

QUERY RESULTS

Table | Raw Response

Showing 1 to 1 of 1 entries Search: Show 50 entries

	num
1	"85"^^xsd:integer

使用FILTER达到同样的效果

查询例子

询问是否存在带有'xx'字符的歌曲名

```
ASK
{
 ?trackID  track_name  ?track_name .
 FILTER  regex(?track_name,"xx")
}
```

```
1  PREFIX music: <http://kg.course/music/>
2
3  ASK
4  {
5    ?trackID  music:track_name  ?track_name .
6    FILTER  regex(?track_name,"008")
7  }
8
```

QUERY RESULTS

Table | Raw Response

True

使用ASK来询问是否存在

ASK的回答只有True或False

更新例子

- 给艺术家id新增属性艺术家名字

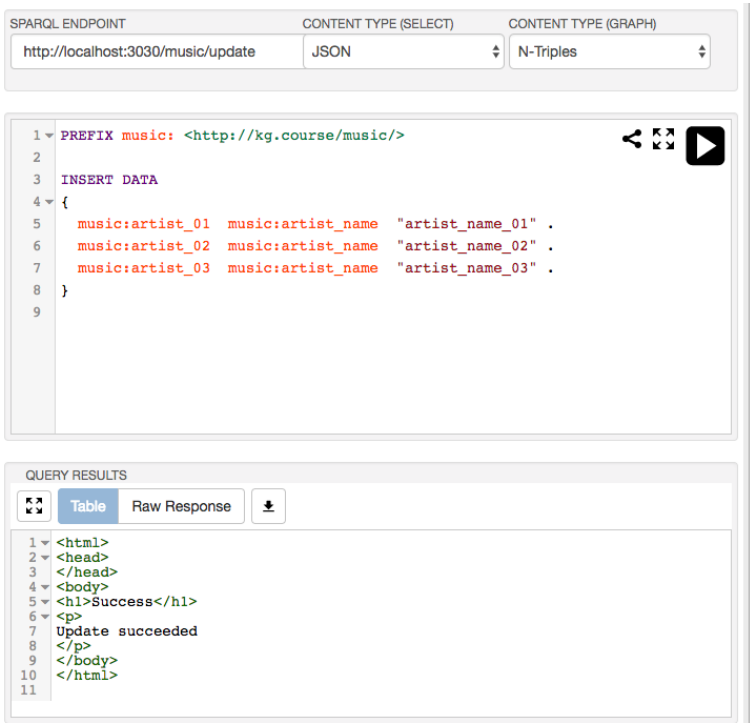
注意在Fuseki进行更新操作时，需要更改endpoin地址
从
http://localhost:3030/music/query
变为
http://localhost:3030/music/update

我们需要进行insert操作

更新例子

给艺术家id新增属性艺术家名字

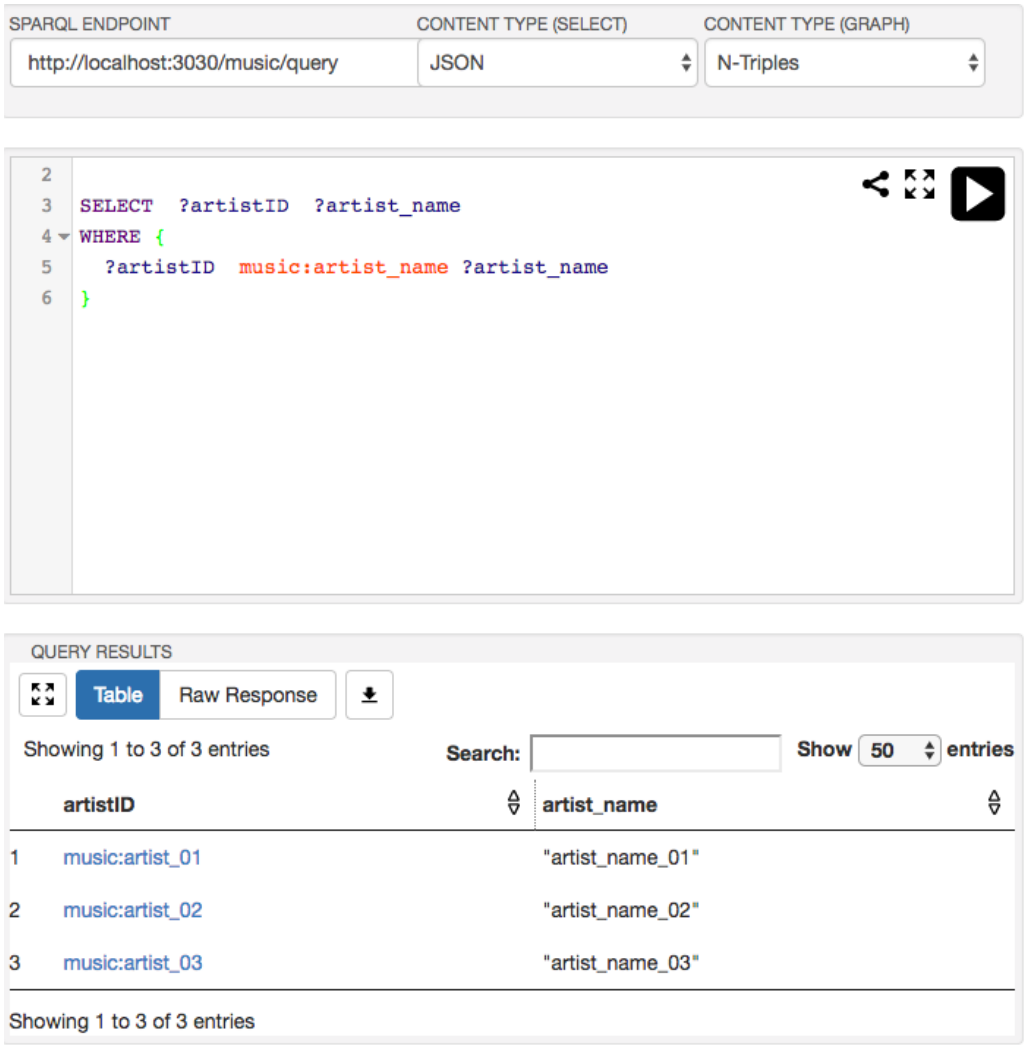
INSERT DATA
{
 artistID artist_name   artist_name .
}



我们使用INSERT DATA的方式，对artist_01,artist_02,artist_03新增了artist_name属性

执行结果显示更新成功

更新例子



把我们刚才插入的艺术家名字属性查询出来，发现是我们刚才插入的值

注意：查询的时候endpoint需要改为query

更新例子

删除增加的属性艺术家名字

```
DELETE
{
 artistID   artist_name ?x .
}

WHERE
{
  artistID  artist_name ?x .
}
```

使用
WHERE定位
DELETE删除

SPARQL ENDPOINT: http://localhost:3030/music/update | CONTENT TYPE (SELECT): JSON | CONTENT TYPE (GRAPH): N-Triples

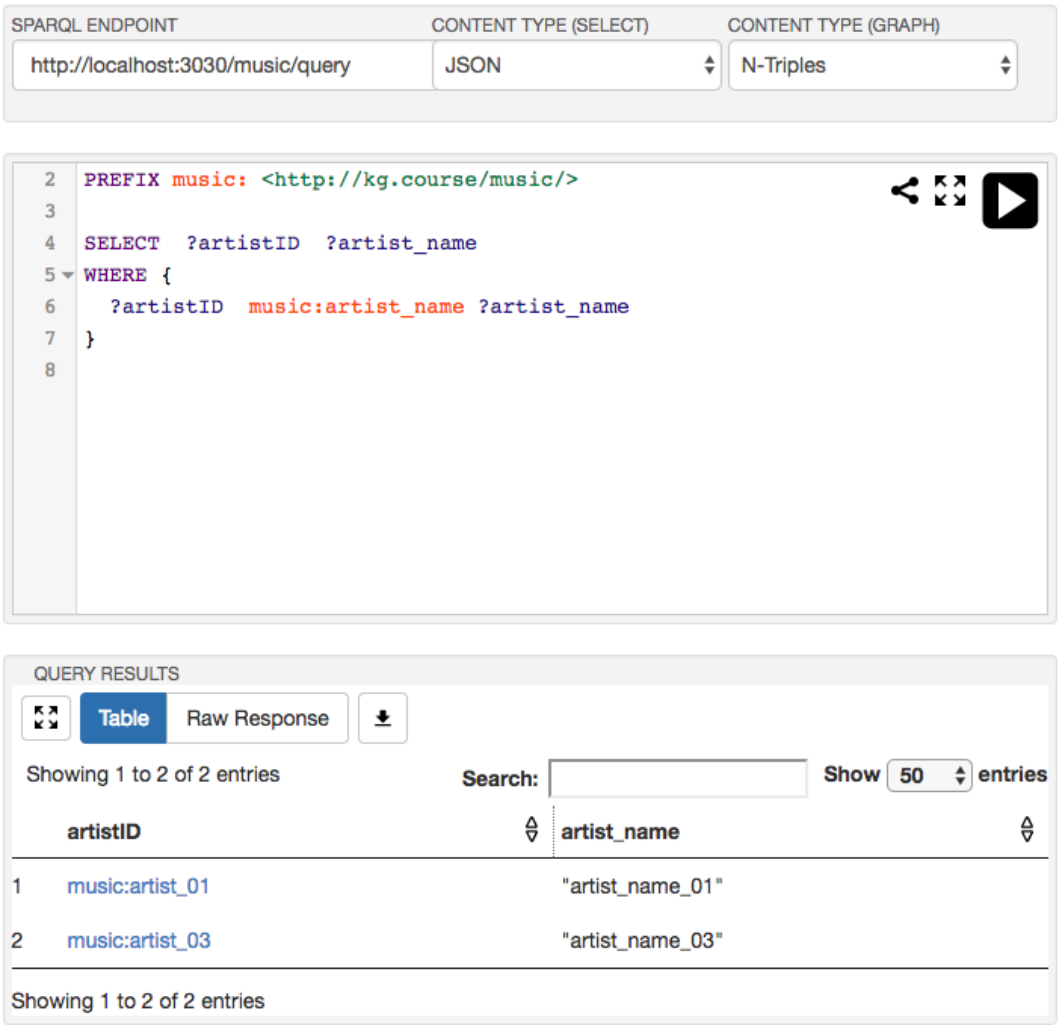
```
PREFIX music: <http://kg.course/music/>

DELETE
{
  music:artist_02   music:artist_name ?x .
}
WHERE
{
   music:artist_02  music:artist_name ?x .
}
```

QUERY RESULTS

Table | Raw Response

```
<html>
<head>
</head>
<body>
<h1>Success</h1>
<p>
Update succeeded
</p>
</body>
</html>
```

更新例子

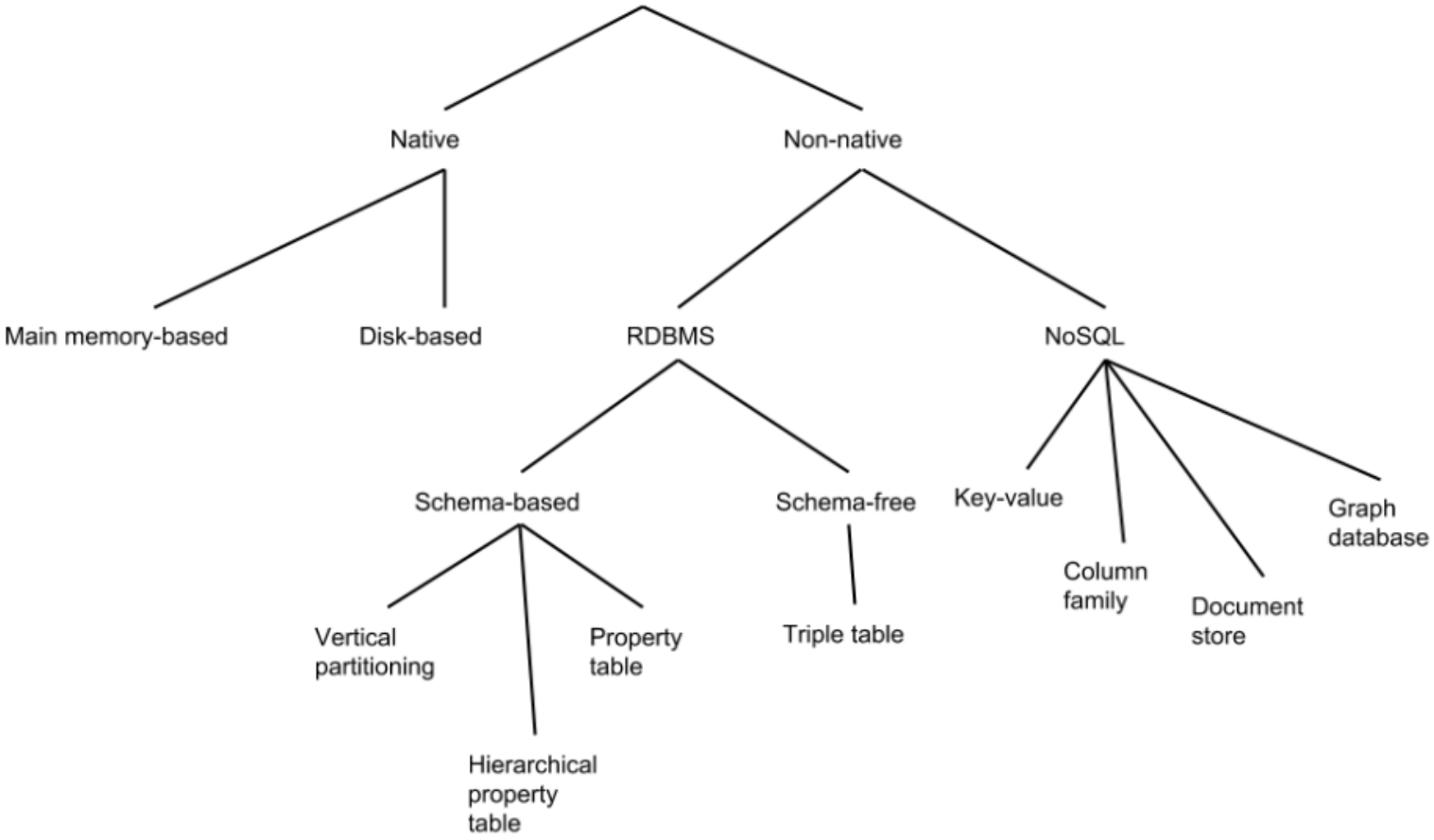


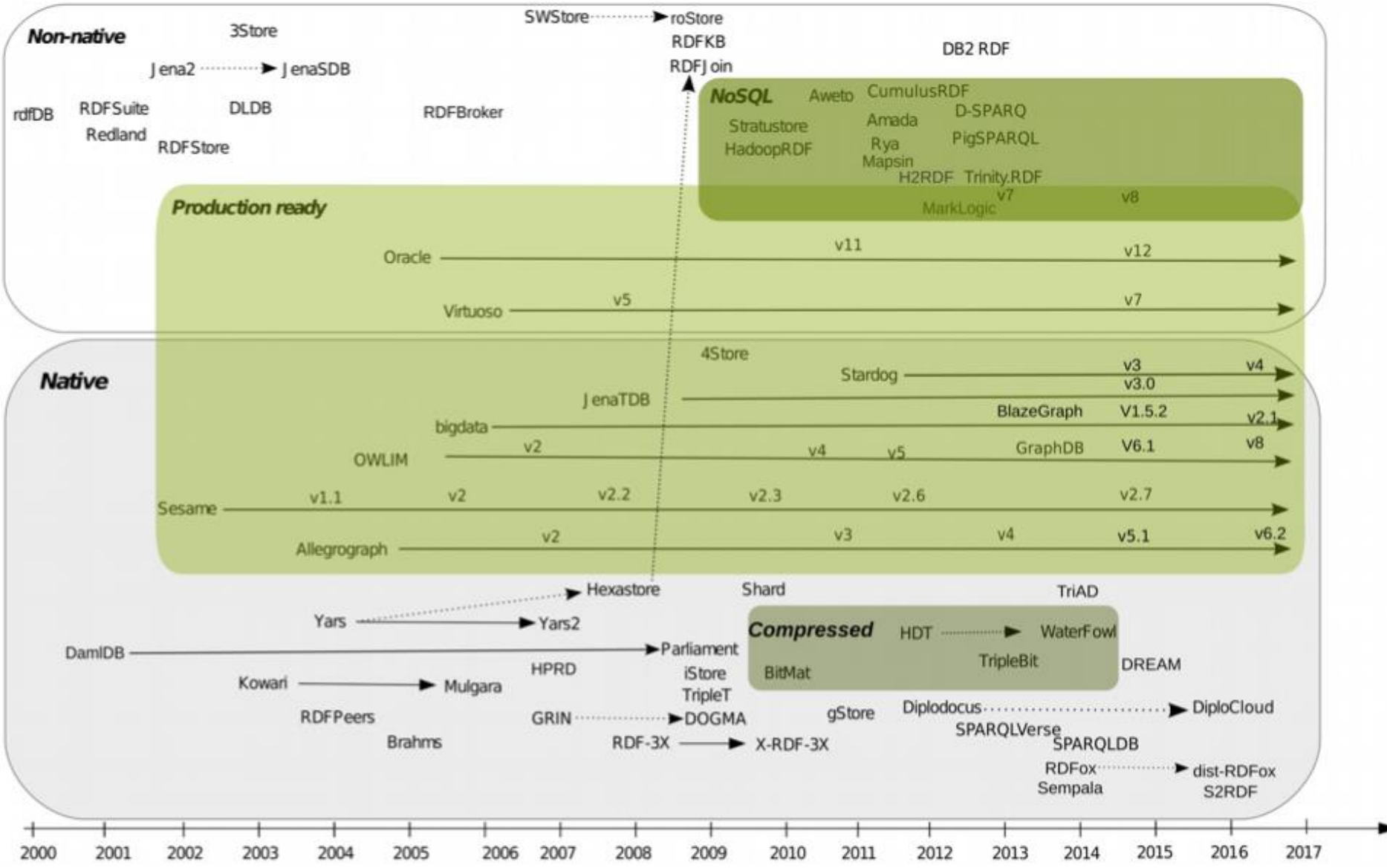
查询发现，artist_02的artist_name_02已经被删除

更多的SPARQL用法请查阅文档

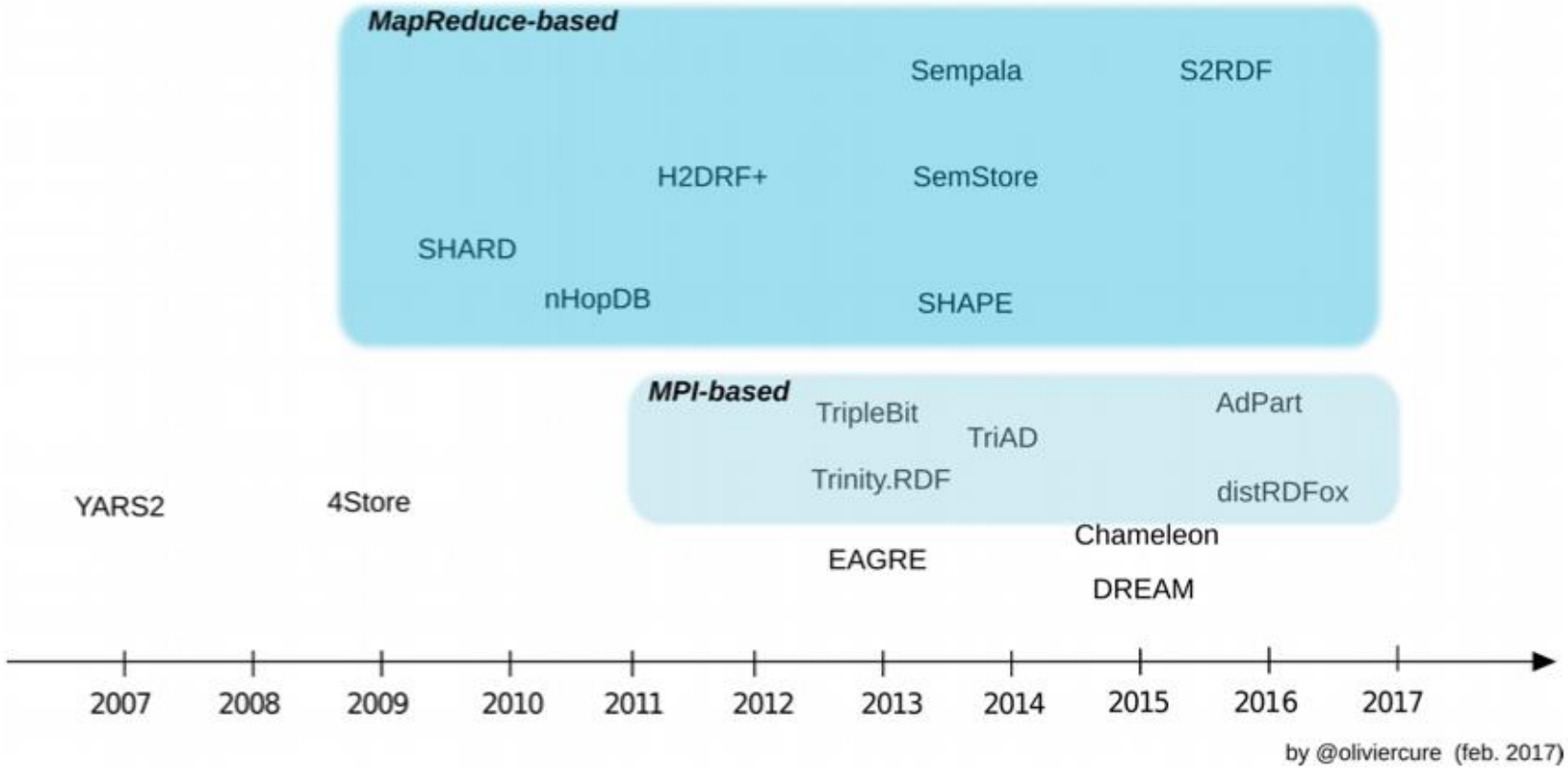
第二部分：图数据库介绍

图数据库分类

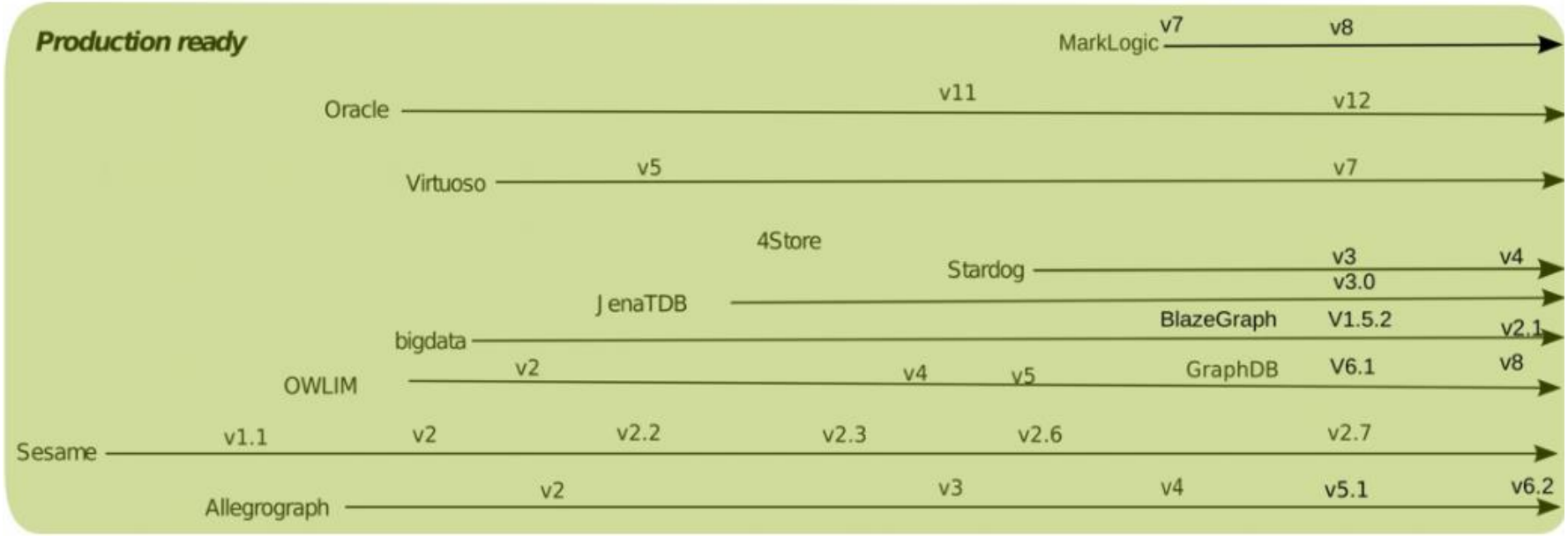

Native
Non-native
Main memory-based
Disk-based
RDBMS
NoSQL
Schema-based
Schema-free
Key-value
Graph database
Column family
Document store
Vertical partitioning
Property table
Triple table
Hierarchical property table


Non-native
3Store
SWStore
roStore
RDFKB
RDFJoin
DB2 RDF
Jena2
JenaSDB
rdfDB
RDFSuite
Redland
DLDB
RDFStore
RDFBroker
NoSQL
Aweto
CumulusRDF
D-SPARQ
Amada
Stratustore
HadoopRDF
Rya
PigSPARQL
Mapsin
H2RDF
Trinity.RDF
v7
v8
MarkLogic
Production ready
Oracle
v11
v12
Virtuoso
v5
v7
Native
4Store
Stardog
v3
v4
v3.0
JenaTDB
BlazeGraph
V1.5.2
v2.1
bigdata
OWLIM
v2
v4
v5
GraphDB
V6.1
v8
Sesame
v1.1
v2
v2.2
v2.3
v2.6
v2.7
Allegrograph
v2
v3
v4
v5.1
v6.2
Hexastore
Shard
TriAD
Yars
Yars2
Compressed
HDT
WaterFowl
DamlDB
Parliament
BitMat
TripleBit
DREAM
HPRD
iStore
Kowari
Mulgara
TripleT
RDFPeers
GRIN
DOGMA
gStore
Diplodocus
DiploCloud
SPARQLVerse
Brahms
RDF-3X
X-RDF-3X
SPARQLDB
RDFox
dist-RDFox
Sempala
S2RDF
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017

图数据库发展


MapReduce-based
Sempala
S2RDF
H2DRF+
SemStore
SHARD
nHopDB
SHAPE
MPI-based
TripleBit
AdPart
TriAD
Trinity.RDF
distRDFox
YARS2
4Store
Chameleon
EAGRE
DREAM
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
by @oliviercure (feb. 2017)

图数据库发展


Production ready
MarkLogic
v7
v8
Oracle
v11
v12
Virtuoso
v5
v7
4Store
Stardog
v3
v4
JenaTDB
v3.0
BlazeGraph
V1.5.2
bigdata
v2.1
OWLIM
v2
v4
v5
GraphDB
V6.1
v8
Sesame
v1.1
v2
v2.2
v2.3
v2.6
v2.7
Allegrograph
v2
v3
v4
v5.1
v6.2

Triple store	Full-text search	Cloud-ready	Extra features
Allegrograph	Integrated + solr	AMI	
Blazegraph	Integrated + solr	AMI	Reification done right
GraphDB	Integrated + solr + elacticsearch (ent.)	AMI	RDF ranking
MarkLogic	Integrated	AMI	With Xquery, Javascript
Oracle	Integrated		Inline in SQL
Stardog	Integrated + Lucene	AMI	Integrity constraints, Explanations
Virtuoso	Integrated	AMI	Inline in SQL

开源图数据库介绍

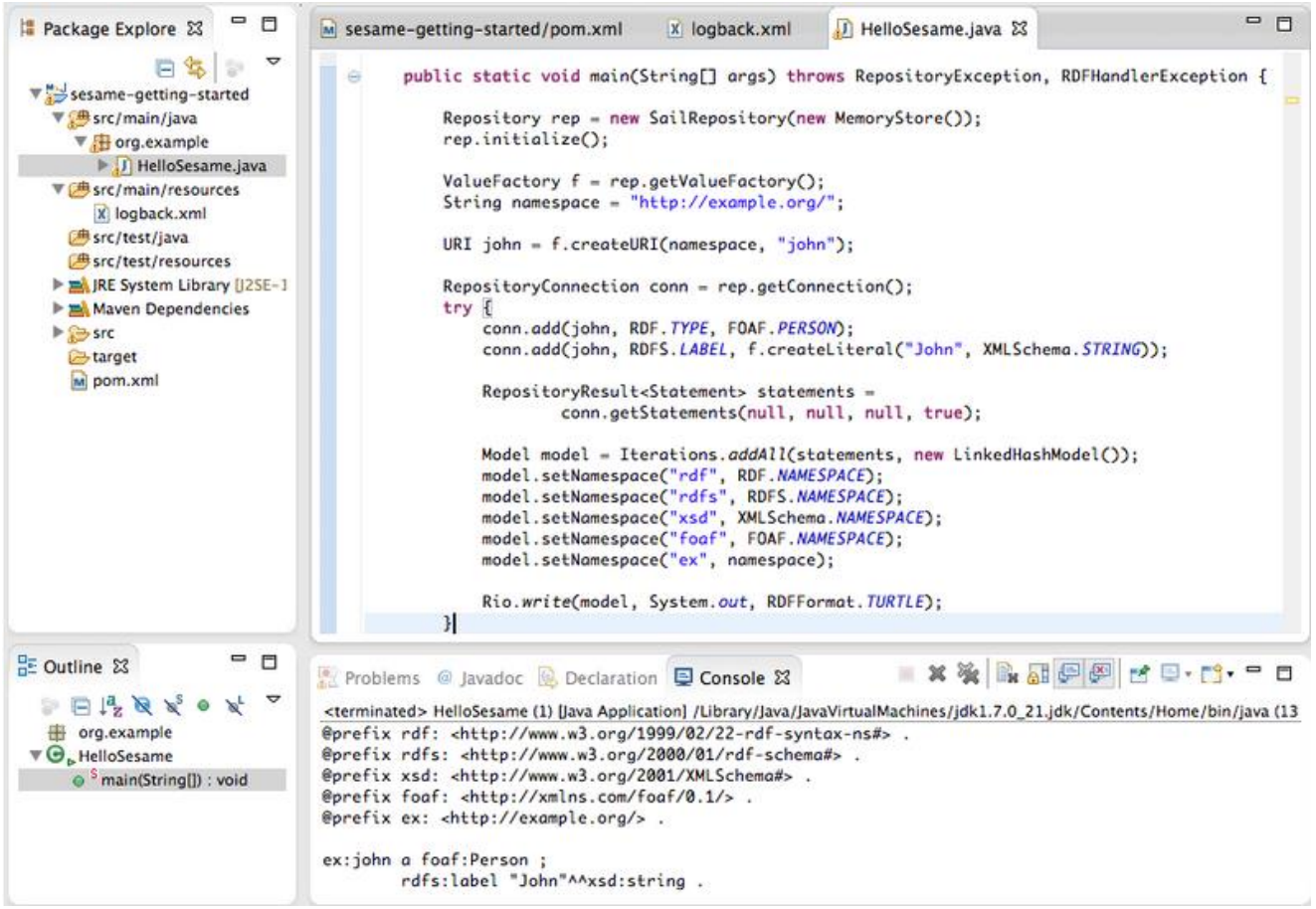
开源数据库 – RDF4J

- 处理RDF数据的Java框架
- 使用简单可用的API来实现RDF存储
- 支持SPARQL endpoints
- 支持两种RDF存储机制
- 支持所有主流的RDF文件格式

http://rdf4j.org

RDF4J

- RDF4J的前身是Sesame



http://docs.rdf4j.org/migration/

开源图数据库 – gStore

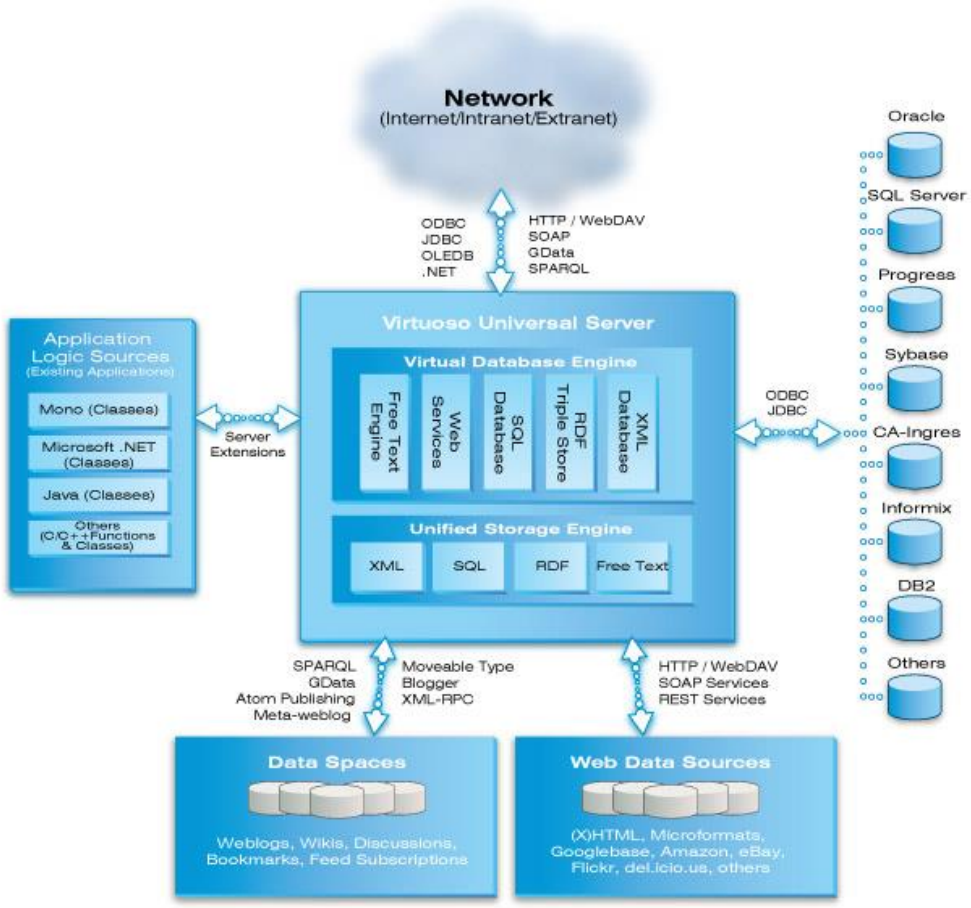
- gStore从图数据库角度存储和检索RDF知识图谱数据；
- gStore支持W3C定义的SPARQL 1.1标准，包括含有Union，OPTIONAL，FILTER和聚集函数的查询；gStore支持有效的增删改操作
- gStore单机可以支持1Billion（十亿）三元组规模的RDF知识图谱的数据管理任务。

http://www.gstore-pku.com

商业图数据库介绍

商业数据库 – Virtuoso

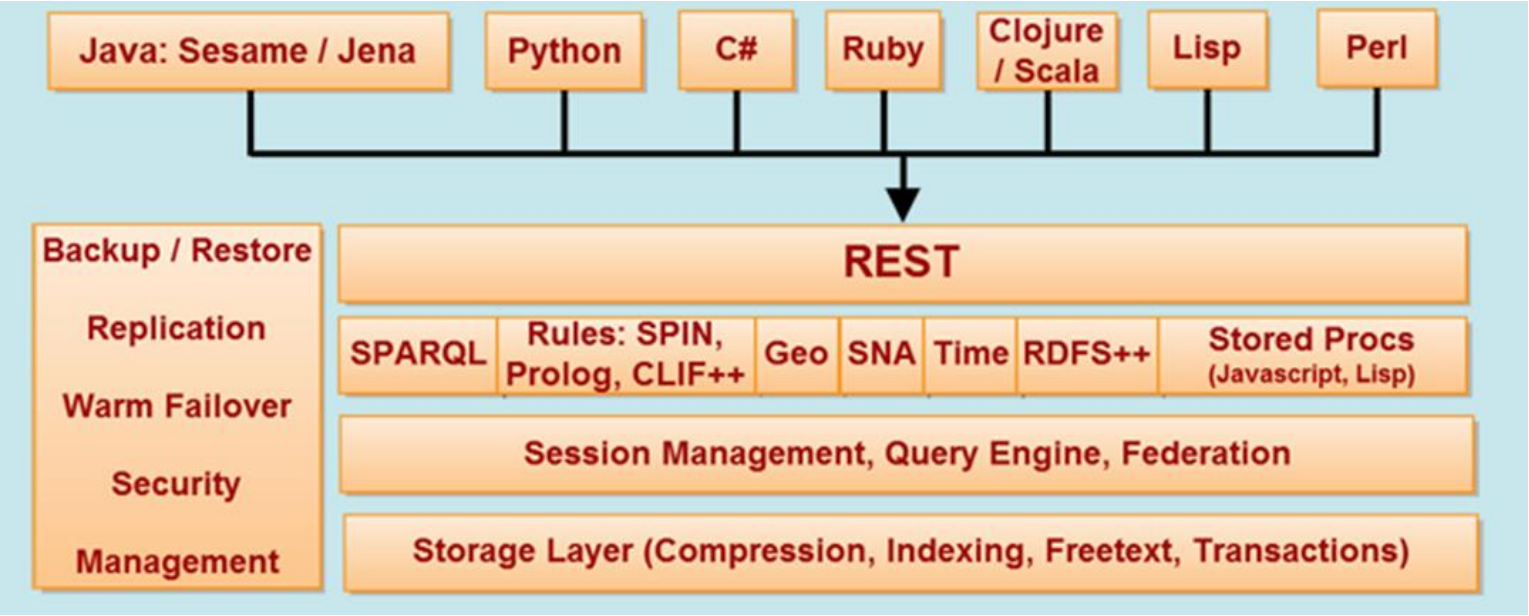
- 智能数据，可视化与整合
- 可扩展和高性能的数据管理
- 支持Web 扩展和安全



http://virtuoso.openlinksw.com

商业数据库 – Allgrograph

- 一个现代的，高性能的，支持永久存储的图数据库
- 基于Restful接入支持多语言编程



http://www.franz.com/agraph/allegrograph

Allgrograph

- AllegroGraph一开始就被设计成为可以提供强大的加载速度，查询速度和高性能的存储图数据库

Load Test	# Triples	Time	Load Rate (T/Sec)
LUBM(8000)*	1.106 Billion	36min, 49 sec	500,679
LUBM(160,000)*	22.12 Billion	12 hrs, 18m, 16s	499,188
AllegroGraph Pre-release**	310.269 Billion	78 hrs, 9m, 23s	1,102,737
AllegroGraph Pre-release***	**1.009 Trillion**	338 hrs, 5m	829,556

商业数据库 – Stardog

The Knowledge Graph Platform for the Enterprise

With Stardog you can unify, query, search, and analyze all your data. Say goodbye to data silos forever.

Benefits

Unifies all the data

Full data variety and every data velocity. Unify disparate and external data sources and turn data into actionable insight.

Reusable Data Model

The right abstraction for all the data enables the Knowledge Graph instead of creating new silos, eliminating significant and costly efforts.

Enterprise Scalability

Query, search, and analyze. Stardog provides enterprise performance and support over massive data sets no matter how complex the data.

Eliminates Complexity

No Knowledge Graph knows as much. Stardog is the only platform to fuse machine learning, reasoning, and rules for contextualized insight of all the data.

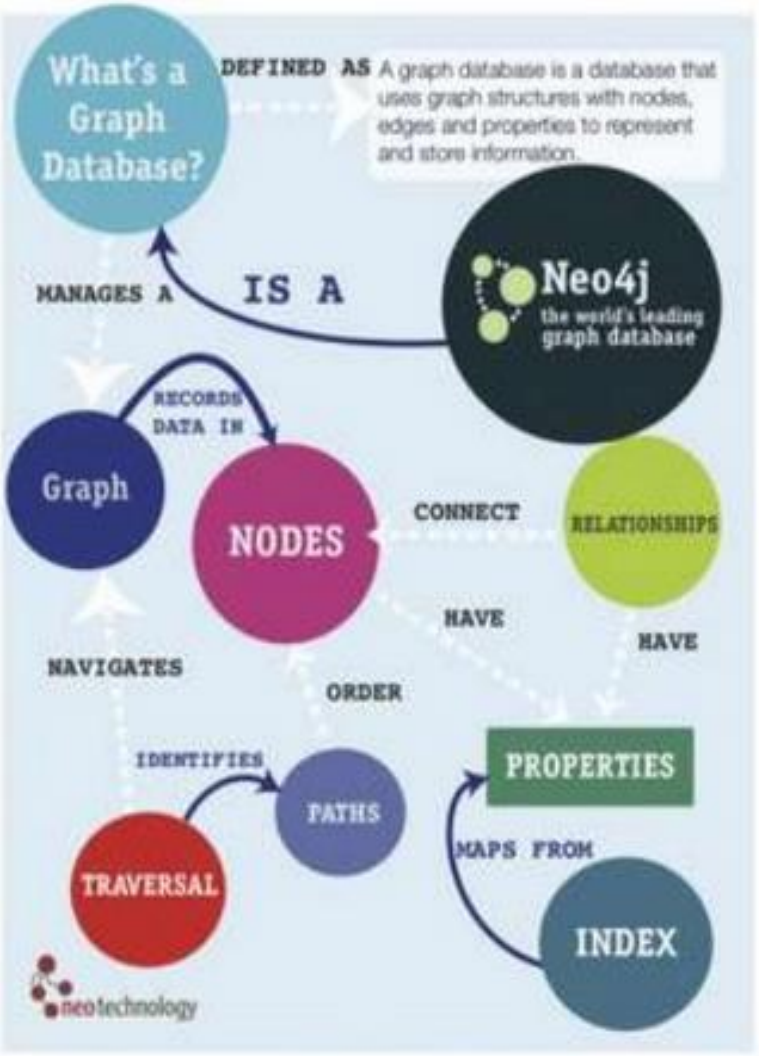
http://www.stardog.com

原生图数据库介绍

原生图数据库 – Neo4j

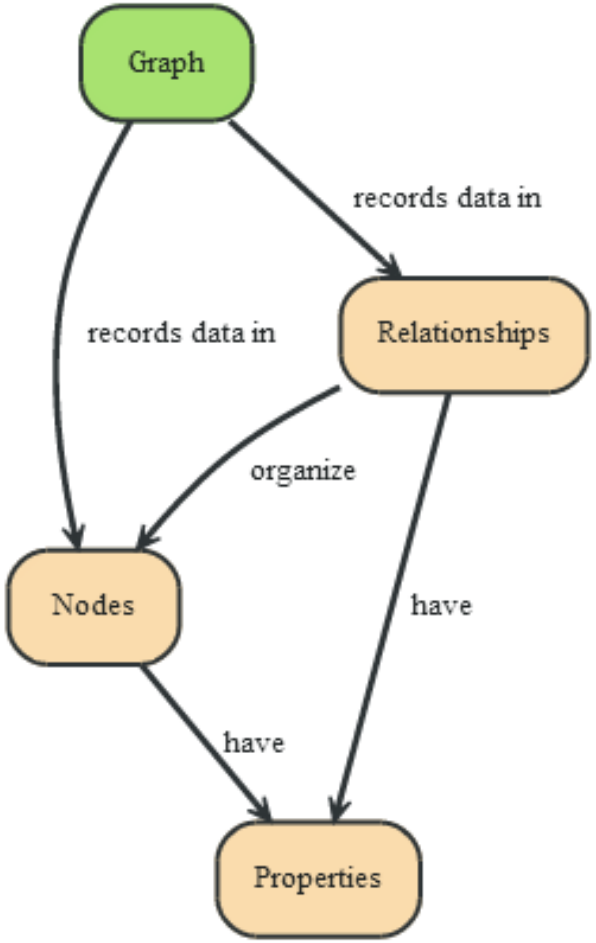
- 图数据库+Lucene索引
- 支持属性图
- 支持ACID
- 高可用性
- 支持320亿的结点，320亿的关系结点，640亿的属性
- REST API接口

http://neo4j.com



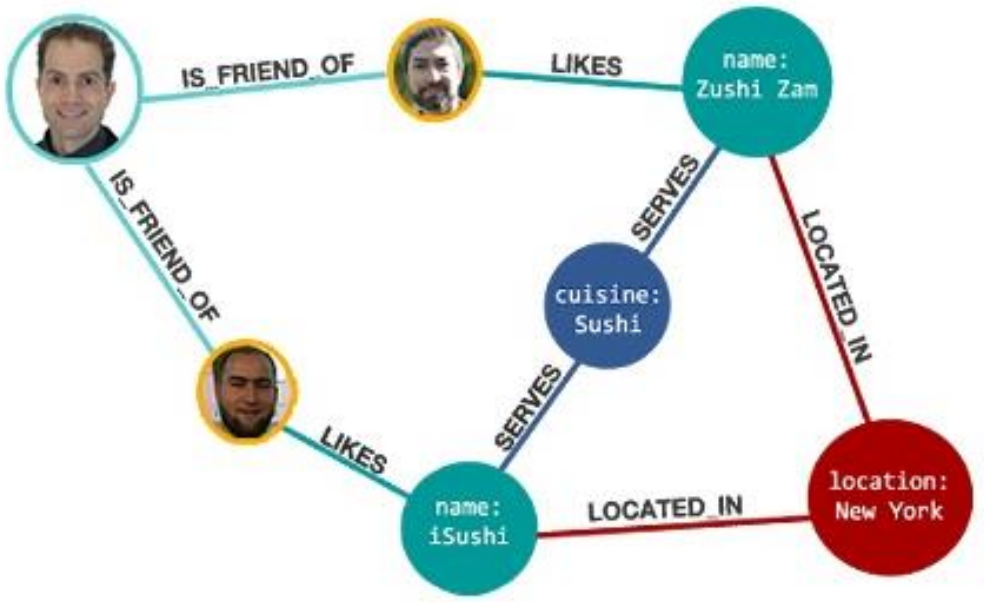
Neo4j – 数据结构

- 在一个图中包含两种基本的数据类型：Nodes (节点) 和 Relationships (关系)。
- Nodes 和 Relationships 包含 key/value形式的属性。Nodes通过Relationships所定义的关系相连起来，形成关系型网络结构。



Neo4j – 优点

- 优点
  - 高连通数据
  - 推荐
  - 路径查找
  - A*算法
  - 数据优先



Neo4j – 安装 (docker)

Docker pull neo4j 官方镜像

```
➜  ~ git:(master) ✗ docker pull neo4j
Using default tag: latest
latest: Pulling from library/neo4j
b56ae66c2937: Downloading [==========>                                        ] 493.1 kB/1.991 MB
81cebc5bcaf8: Download complete
3b27fd892ecb: Downloading [>                                                  ] 525.9 kB/54.29 MB
f120ec548211: Downloading [==>                                                ] 81.11 kB/1.539 MB
0c039dfb6b17: Waiting
c7bf503f11ca: Waiting
94ca2e957a63: Waiting
```

```
➜  ~ git:(master) ✗ docker run -d -p 7473-7474:7473-7474  -p 7687:7687 neo4j
65477caf20a1bc50ee0f4c7a54fcd83e5ad0c6de125b556c01b0dc1b83f6995f
```

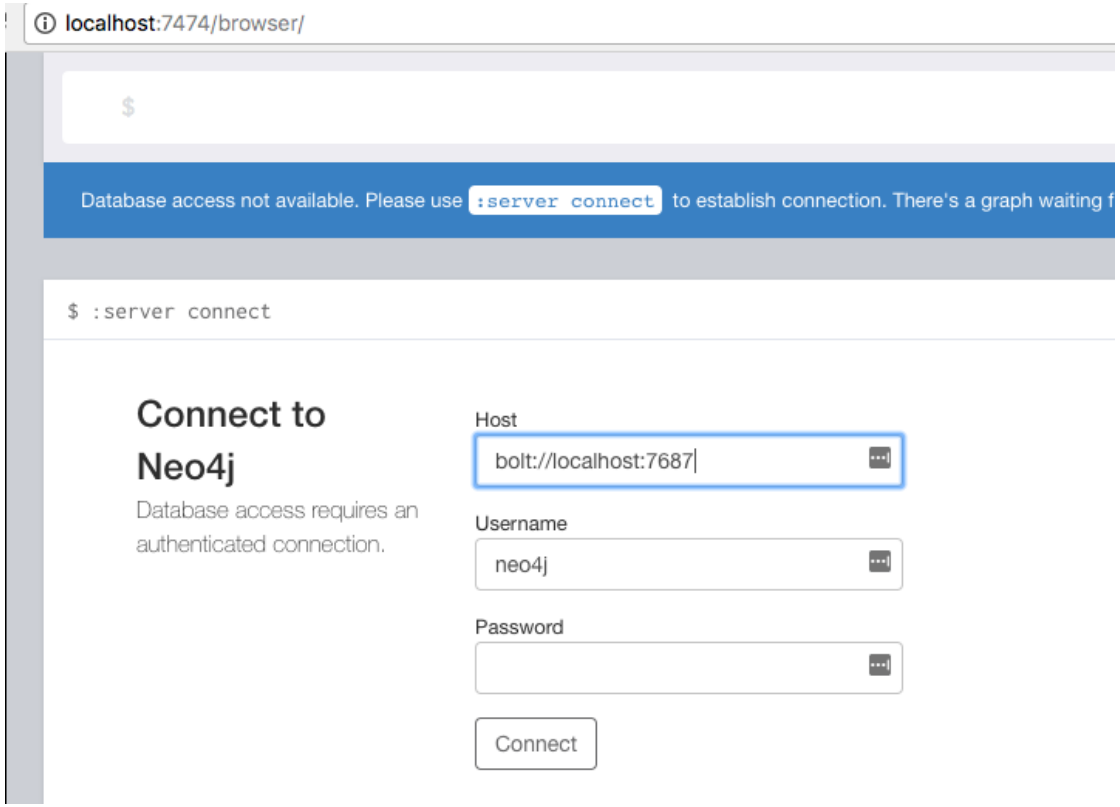
运行neo4j

docker run -d -p 7473-7474:7473-7474 -p 7687:7687 neo4j

注：7473端口:https前端服务；7474端口：http前端服务；
7687端口：bolt端口，后台数据库连接服务

Neo4j – 官方WEB前端

用浏览器打开
http://localhost:7474/browser/



Host地址：
即bolt地址，指定数据库连接服务端口

默认用户名：neo4j
密码： neo4j

点击Connect会出现修改密码界面，修改密码之后即进去管理界面

顶部输入框：
执行脚本、查询语句
下方动态显示结果

Neo4j – 数据导入

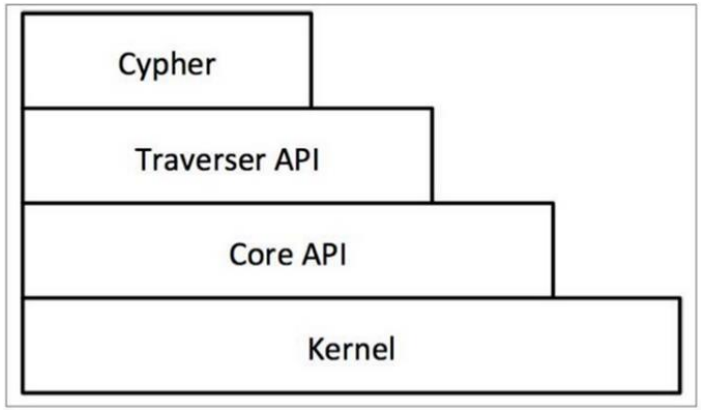
1. Cypher CREATE语句，为每一条数据写一个CREATE
2. Cypher LOAD CSV 语句，将数据转成CSV格式，通过LOAD CSV读取数据
3. 官方提供的Java API — Batch Inserter
4. 官方提供的 neo4j-import 工具
5. 第三方开发者编写的 Batch Import 工具

Neo4j – 数据存储

```
graph.db ls
index                                   neostore.nodestore.db.labels.id
neostore                                neostore.propertystore.db
neostore.counts.db.a                    neostore.propertystore.db.arrays
neostore.counts.db.b                    neostore.propertystore.db.arrays.id
neostore.id                             neostore.propertystore.db.id
neostore.labelscanstore.db              neostore.propertystore.db.index
neostore.labeltokenstore.db             neostore.propertystore.db.index.id
neostore.labeltokenstore.db.id          neostore.propertystore.db.index.keys
neostore.labeltokenstore.db.names       neostore.propertystore.db.index.keys.id
neostore.labeltokenstore.db.names.id    neostore.propertystore.db.strings
neostore.nodestore.db                   neostore.propertystore.db.strings.id
neostore.nodestore.db.id                neostore.relationshipgroupstore.db
neostore.nodestore.db.labels            neostore.relationshipgroupstore.db.id

neostore.relationshipstore.db
neostore.relationshipstore.db.id
neostore.relationshiptypestore.db
neostore.relationshiptypestore.db.id
neostore.relationshiptypestore.db.names
neostore.relationshiptypestore.db.names.id
neostore.schemastore.db
neostore.schemastore.db.id
neostore.transaction.db.51
neostore.transaction.db.52
schema
store_lock
```

Neo4j – 查询数据

Cypher 查询语言

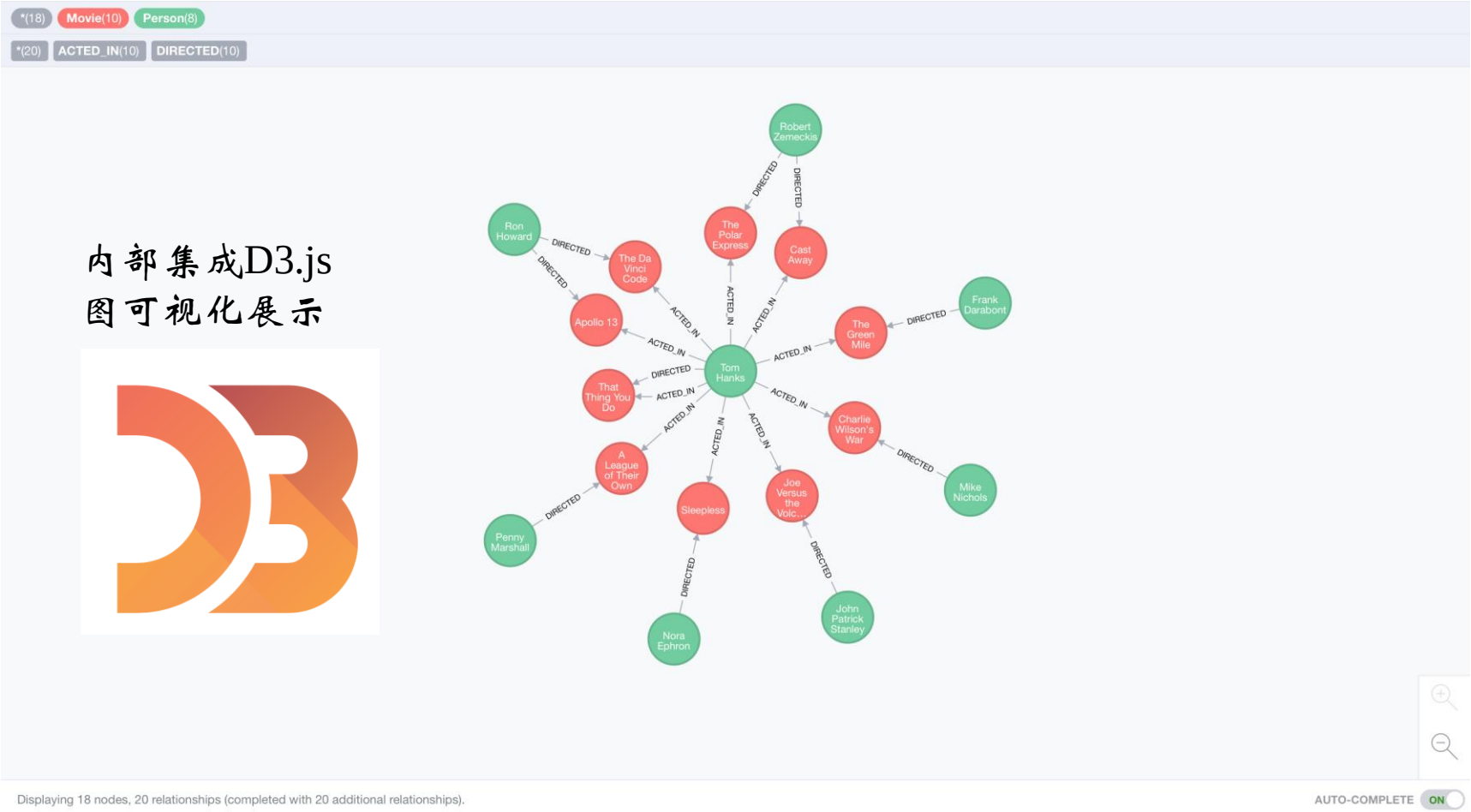


Query: Papers cited by a paper written by *Peter Smith*.

```
START author=node:authors(
                'firstname:"Peter" AND lastname:"Smith"')
MATCH (author)-[:WROTE]->()-[:CITES](papers)
RETURN papers
```

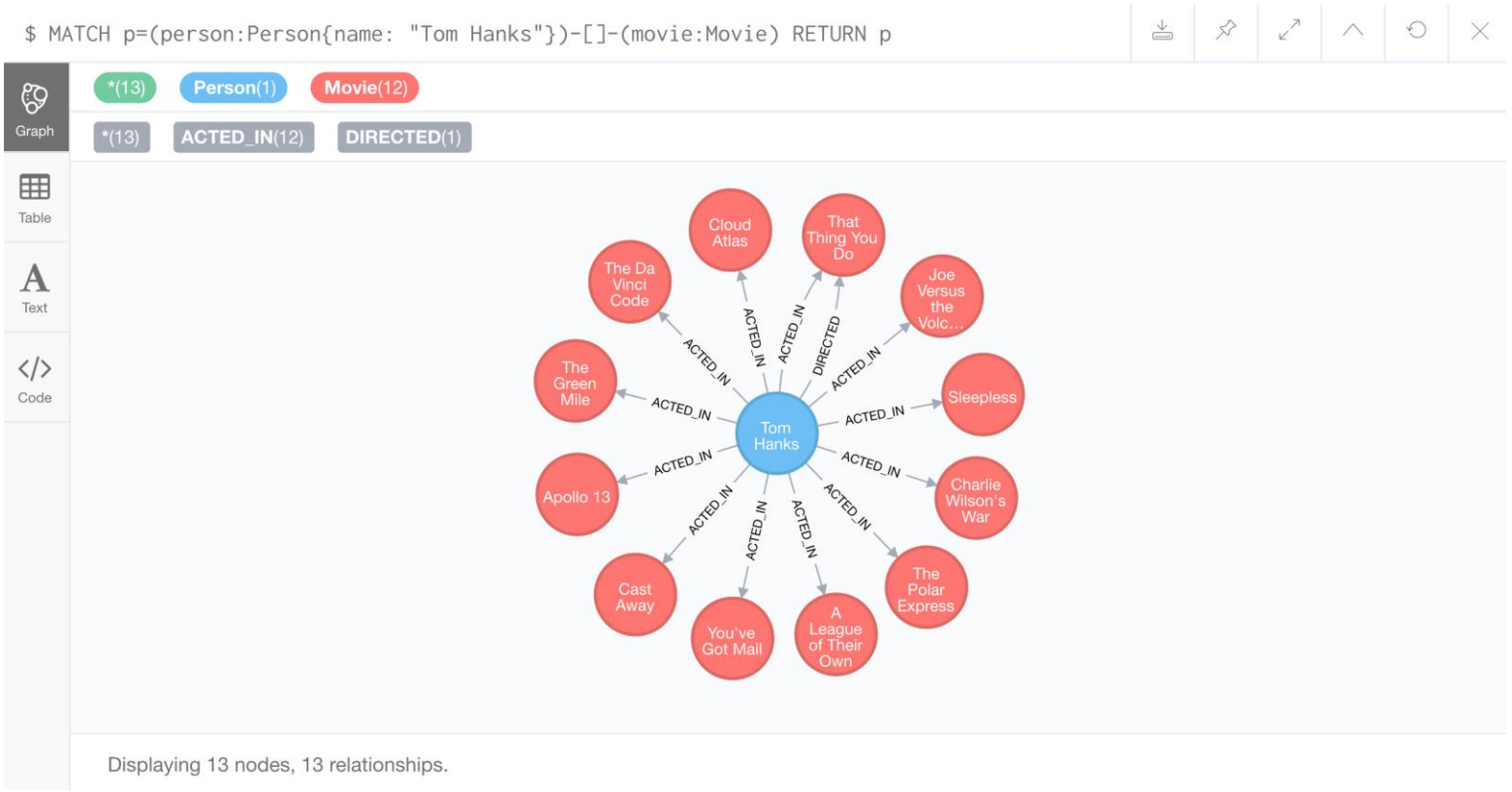
Neo4j – Movie Graph示例

内部集成D3.js
图可视化展示



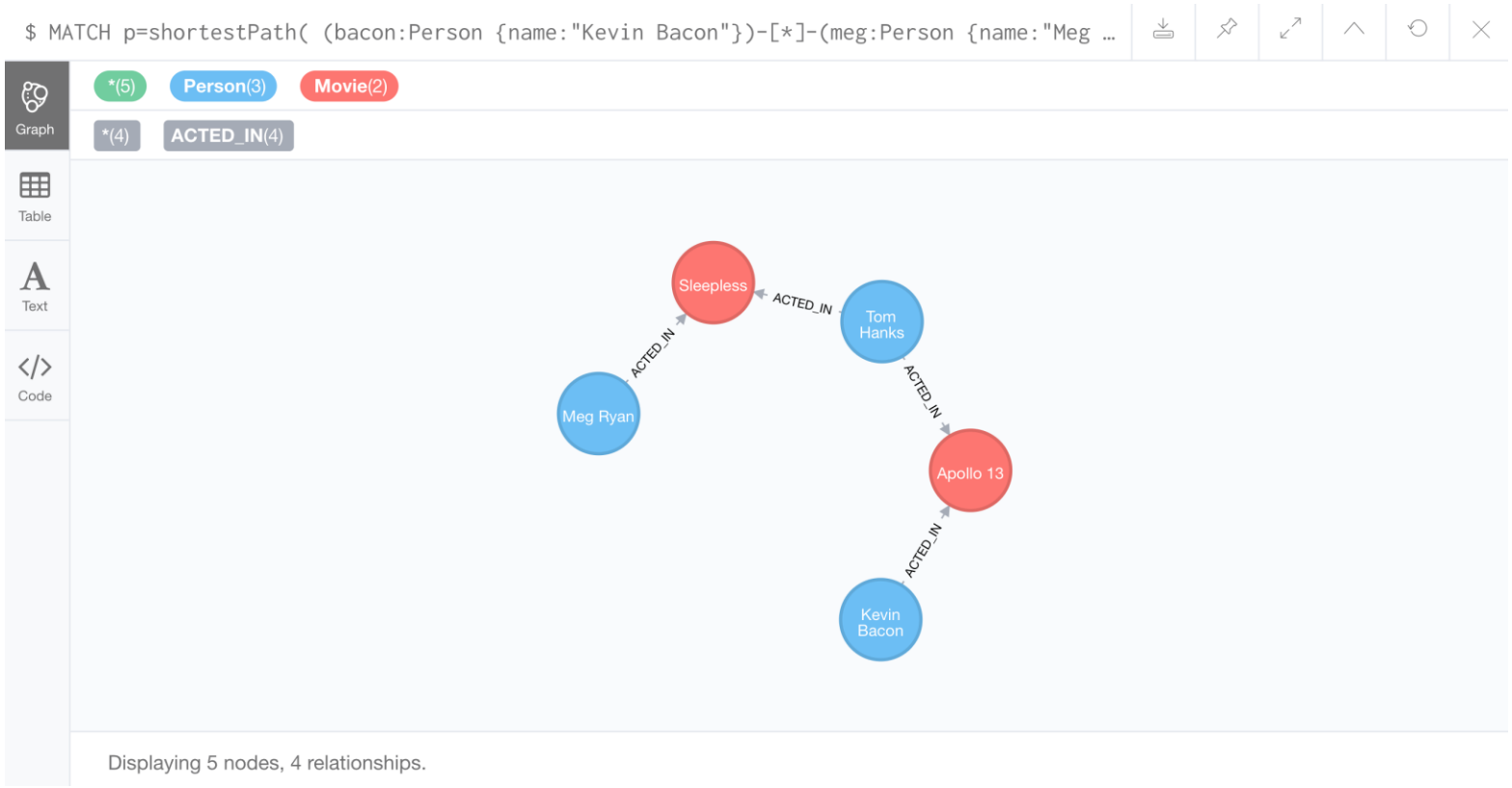
Neo4j – Movie Graph示例

- 查询和“Tom Hanks”有关的电影



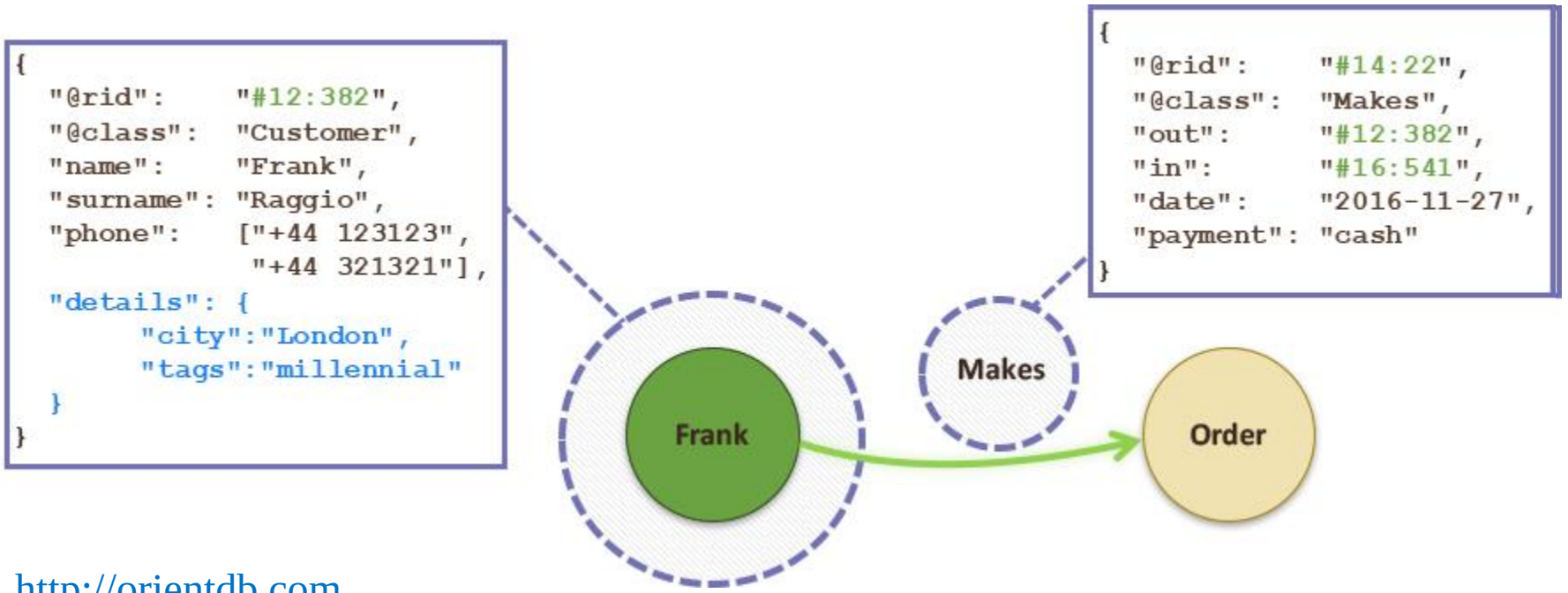
Neo4j – Movie Graph示例

- 查询两个节点间的最短路径



原生图数据库 – OrientDB

OrientDB是一个用Java实现的开源NoSQL数据库管理系统。它是一个多模式的数据库，支持图形、文档、键值对、对象模型和关系，也可以为图数据库的管理与记录之间的提供连接



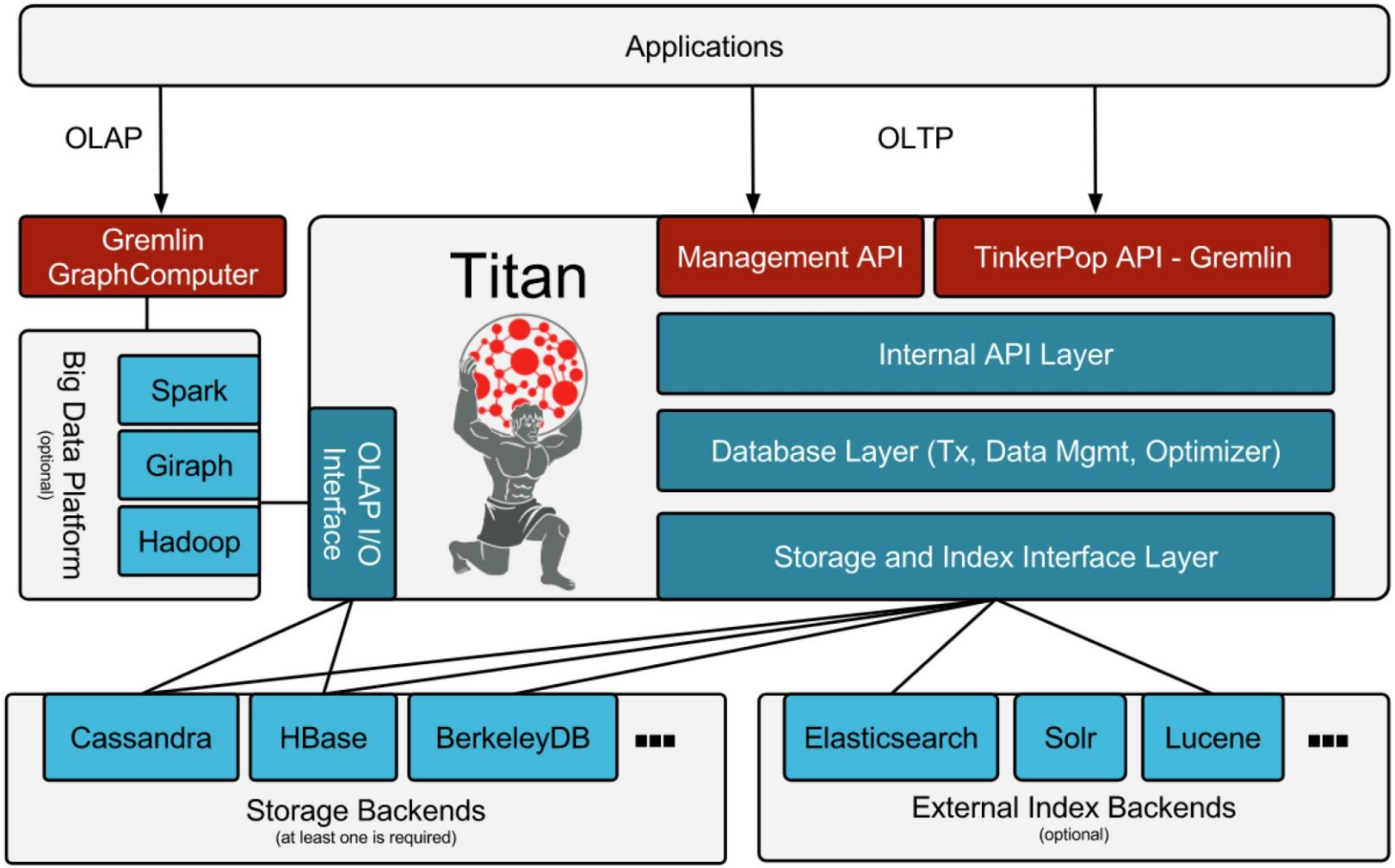
http://orientdb.com

原生图数据库 – Titan

- 弹性和线性增长的数据和用户的可扩展性
- 数据分布和复制性能和容错性
- 支持增删改查，支持一致性
- 支持各种后端存储
- 支持全局图数据分析，报告，并通过ETL连接大数据平台
- 支持全文检索

http://titan.thinkaurelius.com

原生图数据库 – Titan



Benchmark

常用Benchmark：

RDF Store Benchmarking

This page collects references to RDF benchmarks, benchmarking results and papers about RDF benchmarking.

At the end of the page, we collect use cases for RDF benchmarking and offer ideas for future discussion on benchmarking triple stores.

RDF Benchmarks

- Berlin SPARQL Benchmark (BSBM), provides for comparing the performance of RDF and Named Graph stores as well as RDF-mapped relational databases and other systems that expose SPARQL endpoints. Designed along an e-commerce use case. SPARQL and SQL version available.
- Lehigh University Benchmark (LUBM) is developed to facilitate the evaluation of Semantic Web repositories in a standard and systematic way. The benchmark is intended to evaluate the performance of those repositories with respect to extensional queries over a large data set that commits to a single realistic ontology.
- Ontology Benchmark (UOBM) extends the LUBM benchmark in terms of inference and scalability testing. UOBM ontology and data set.
- The SP^{2}Bench SPARQL Performance Benchmark, provides a scalable RDF data generator and a set of benchmark queries, designed to test typical SPARQL operator constellations and RDF data access patterns.
- Social Network Intelligence Benchmark (SIB) A benchmark suite developed by people at CWI and Openlink taking the schema from Social Networks for generating test areas where RDF/SPARQL can truly excel, and challenging query processing over highly connected graph.
- DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data.
- FedBench – Benchmark for measuring the performance of federated SPARQL query processing (ISWC2011 paper about FedBench).
- Linked Data Integration Benchmark (LODIB) is a benchmark for comparing the expressivity as well as the runtime performance of Linked Data translation/integration systems.
- JustBench analyses the performance of OWL reasoners based on justifications for entailments (project website).
- The ISLab Instance Matching Benchmark, provides for benchmarking instance matching and itentity resolution tools.
- THALIA Testbed for testing the expressiveness of relational-to-RDF mapping languages.
- A Benchmark for Spatial Semantic Web Systems, by Dave Kolas, extends the LUBM with sample spatial data.
- Linked Open Data Quality Assessment (LODQA) is a benchmark for comparing data quality assessment and data fusion systems.
- LinkBench – A Database Benchmark Based on the Facebook Social Graph (SIGMOD, 2013 Paper)
- Waterloo SPARQL Diversity Test Suite (WatDiv) is a highly customizable stress testing tool for RDF data management systems (ISWC, 2014 Paper).
- Semantic Publishing Benchmark (SPB) developed by the LDBC according to use case requirements from the BBC.
- FEASIBLE is a Feature-Based SPARQL Benchmark Generation Framework (ISWC, 2015 Paper).

https://www.w3.org/wiki/RdfStoreBenchmarking

Benchmark

- 常用衡量指标
  - Load Time
  - Repository Size
  - Query Response Time
  - Throughputs
  - Inference Support

图数据库 vs 图计算

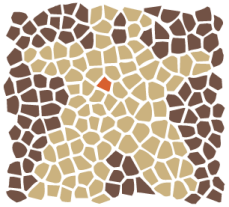
图查询

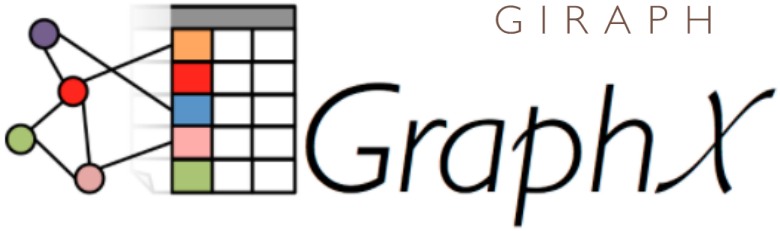
- 图数据结构存储
- 子图匹配与查询优化

利用图结构进行语义查询，包括节点、边和属性来表示和存储数据

图计算

- 图性质计算
  - PageRank
  - 最短路径
- 图挖掘





第三部分:图数据库实现细节

Example: Single Triple Table

ex:Katja	ex:teaches	ex:Databases;
	ex:works_for	ex:MPI_Informatics;
	ex:PhD_from	ex:TU_Ilmenau.
ex:Martin	ex:teaches	ex:Databases;
	ex:works_for	ex:MPI_Informatics;
	ex:PhD_from	ex:Saarland_University.
ex:Ralf	ex:teaches	ex:Information_Retrieval;
	ex:PhD_from	ex:Saarland_University;
	ex:works_for	ex:Saarland_University,
		ex:MPI_Informatics.

Example: Single Triple Table

```
ex:Katja    ex:teaches      ex:Databases;
            ex:works_for    ex:MPI_Informatics;
            ex:PhD_from     ex:TU_Ilmenau.
ex:Martin   ex:teaches      ex:Databases;
            ex:works_for    ex:MPI_Informatics;
            ex:PhD_from     ex:Saarland_University.
ex:Ralf     ex:teaches      ex:Information_Retrieval;
            ex:PhD_from     ex:Saarland_University;
            ex:works_for    ex:Saarland_University,
                            ex:MPI_Informatics.
```

subject	predicate	object
ex:Katja	ex:teaches	ex:Databases
ex:Katja	ex:works_for	ex:MPI_Informatics
ex:Katja	ex:PhD_from	ex:TU_Ilmenau
ex:Martin	ex:teaches	ex:Databases
ex:Martin	ex:works_for	ex:MPI_Informatics
ex:Martin	ex:PhD_from	ex:Saarland_University
ex:Ralf	ex:teaches	ex:Information_Retrieval
ex:Ralf	ex:PhD_from	ex:Saarland_University
ex:Ralf	ex:works_for	ex:Saarland_University
ex:Ralf	ex:works_for	ex:MPI_Informatics

Conversion of SPARQL to SQL

General approach to translate SPARQL into SQL:

(1) Each **triple pattern** is translated into a (self-) **JOIN** over the triple table

(2) **Shared variables** create **JOIN conditions**

(3) **Constants** create **WHERE conditions**

(4) **FILTER conditions** create **WHERE conditions**

(5) **OPTIONAL clauses** create **OUTER JOINS**

(6) **UNION clauses** create **UNION expressions**

Example: Conversion to SQL Query

```
SELECT ?a ?b ?t WHERE
{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }
OPTIONAL {?a teaches ?t}
FILTER (regex(?u, “Saar”))
```

Example: Conversion to SQL Query

```
SELECT ?a ?b ?t WHERE
{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }
OPTIONAL {?a teaches ?t}
FILTER (regex(?u, “Saar”))

SELECT
FROM Triples P1, Triples P2, Triples P3
```

Example: Conversion to SQL Query

```
SELECT ?a ?b ?t WHERE
{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }
OPTIONAL {?a teaches ?t}
FILTER (regex(?u, “Saar”))
```

```
SELECT
FROM Triples P1, Triples P2, Triples P3
WHERE P1.predicate=“works_for” AND P2.predicate=“works_for”
    AND  P3.predicate=“phd_from”
```

Example: Conversion to SQL Query

SELECT ?a ?b ?t WHERE
{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }
OPTIONAL {?a teaches ?t}
FILTER (regex(?u, “Saar”))

SELECT
FROM Triples P1, Triples P2, Triples P3
WHERE P1.predicate=“works_for” AND P2.predicate=“works_for”
AND P3.predicate=“phd_from”
AND P1.object=P2.object AND P1.subject=P3.subject AND P1.object=P3.object

Example: Conversion to SQL Query

SELECT ?a ?b ?t WHERE
{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }
OPTIONAL {?a teaches ?t}
FILTER (regex(?u, “Saar”))

SELECT
FROM Triples P1, Triples P2, Triples P3
WHERE P1.predicate=“works_for” AND P2.predicate=“works_for”
AND P3.predicate=“phd_from”
AND P1.object=P2.object AND P1.subject=P3.subject AND P1.object=P3.object
AND REGEXP_LIKE(P1.object, “Saar”)

Example: Conversion to SQL Query

SELECT ?a ?b ?t WHERE
{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }
OPTIONAL {?a teaches ?t}
FILTER (regex(?u, "Saar"))

```
SELECT P1.subject as A, P2.subject as B
FROM Triples P1, Triples P2, Triples P3
WHERE P1.predicate="works_for" AND P2.predicate="works_for"
    AND  P3.predicate="phd_from"
    AND  P1.object=P2.object AND P1.subject=P3.subject AND P1.object=P3.object
    AND REGEXP_LIKE(P1.object,  "Saar")
```

Example: Conversion to SQL Query

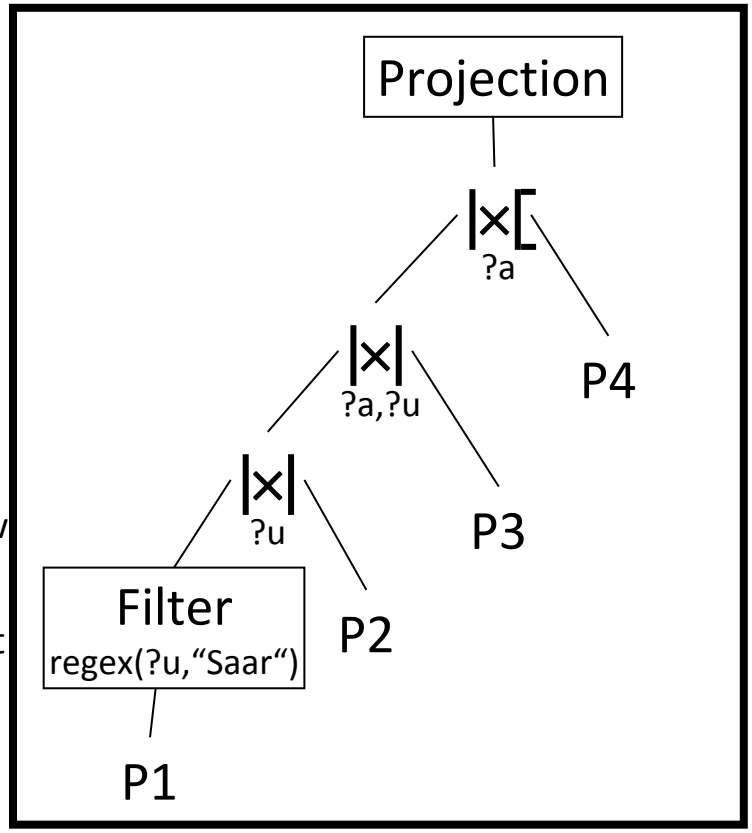
```
SELECT ?a ?b ?t WHERE
{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }
OPTIONAL {?a teaches ?t}
FILTER (regex(?u, "Saar"))
```

```
SELECT R1.A, R1.B, R2.T FROM
( SELECT P1.subject as A, P2.subject as B
  FROM Triples P1, Triples P2, Triples P3
  WHERE P1.predicate="works_for" AND P2.predicate="works_for"
      AND  P3.predicate="phd_from"
      AND  P1.object=P2.object AND P1.subject=P3.subject AND P1.object=P3.object
      AND REGEXP_LIKE(P1.object, "Saar")
 ) R1 LEFT OUTER JOIN
 ( SELECT P4.subject as A, P4.object as T FROM Triples P4
   WHERE P4.predicate="teaches") AS R2
) ON (R1.A=R2.A)
```

Example: Conversion to SQL Query

```
SELECT ?a ?b ?t WHERE
{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }
OPTIONAL {?a teaches ?t}
FILTER (regex(?u, "Saar"))
```

```
SELECT R1.A, R1.B, R2.T FROM
( SELECT P1.subject as A, P2.subject as B
  FROM Triples P1, Triples P2, Triples P3
  WHERE P1.predicate="works_for" AND P2.predicate="w
      AND  P3.predicate="phd_from"
      AND  P1.object=P2.object AND P1.subject=P3.subject
      AND REGEXP_LIKE(P1.object, "Saar")
 ) R1 LEFT OUTER JOIN
 ( SELECT P4.subject as A, P4.object as T FROM Triples P4
   WHERE P4.predicate="teaches") AS R2
) ON (R1.A=R2.A)
```



Is that all?

Well, no.

- Which **indexes** should be built?
  (to support efficient evaluation of triple patterns)
- How can we **reduce storage space**?
- How can we find the **best execution plan**?

Is that all?

Well, no.

- Which **indexes** should be built?
  (to support efficient evaluation of triple patterns)
- How can we **reduce storage space**?
- How can we find the **best execution plan**?

Existing databases need modifications:

- flexible, extensible, generic storage not needed here
- cannot deal with multiple self-joins of a single table
- often generate bad execution plans

Dictionary for Strings

Map all strings to unique integers (e.g., via hashing)

- Regular size (4-8 bytes), much easier to handle
- Dictionary usually small, can be kept in main memory

`<http://example.de/Katja>`	→	194760
`<http://example.de/Martin>`	→	679375
`<http://example.de/Ralf>`	→	4634

Dictionary for Strings

Map all strings to unique integers (e.g., via hashing)

- Regular size (4-8 bytes), much easier to handle
- Dictionary usually small, can be kept in main memory

<http://example.de/Katja> → 194760
<http://example.de/Martin> → 679375
<http://example.de/Ralf> → 4634

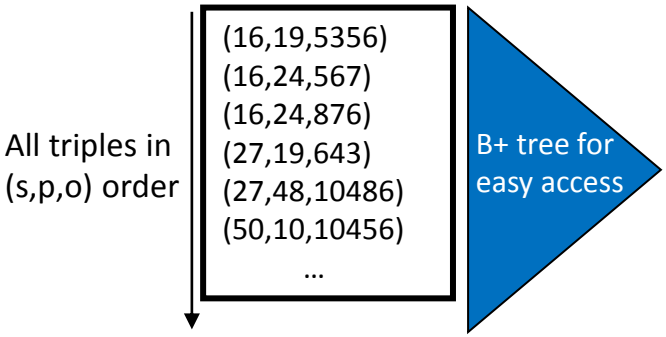
This may break original lexicographic sorting order
⇒ RANGE conditions (not in SPARQL) are difficult!
⇒ FILTER conditions may be more expensive!

Indexes for Commonly Used Triple Patterns

Patterns with a single variable are frequent

Example: `Albert_Einstein invented ?x`

⇒ Build **clustered index** over (s,p,o)

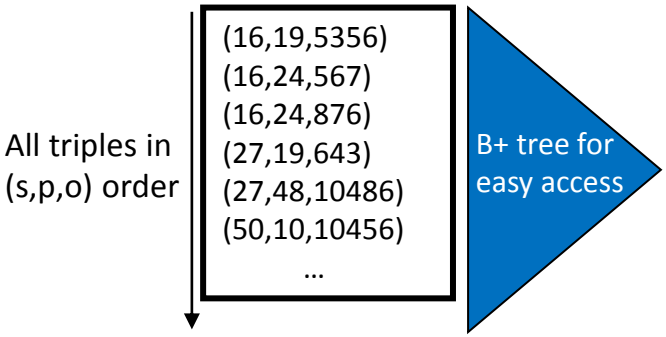


Indexes for Commonly Used Triple Patterns

Patterns with a single variable are frequent

Example: `Albert_Einstein invented ?x`

⇒ Build **clustered index** over (s,p,o)



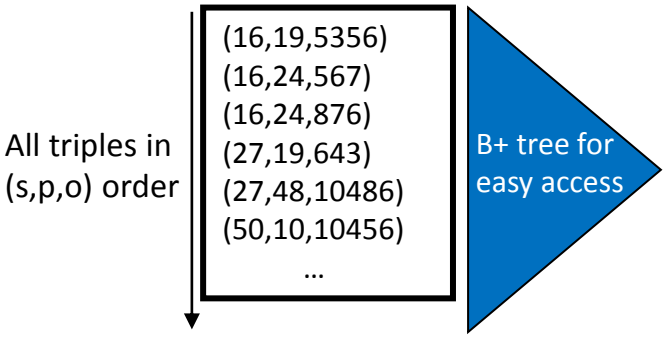
1. Lookup ids for constants:
   `Albert_Einstein` → 16, `invented` → 24

Indexes for Commonly Used Triple Patterns

Patterns with a single variable are frequent

Example: `Albert_Einstein invented ?x`

⇒ Build **clustered index** over (s,p,o)



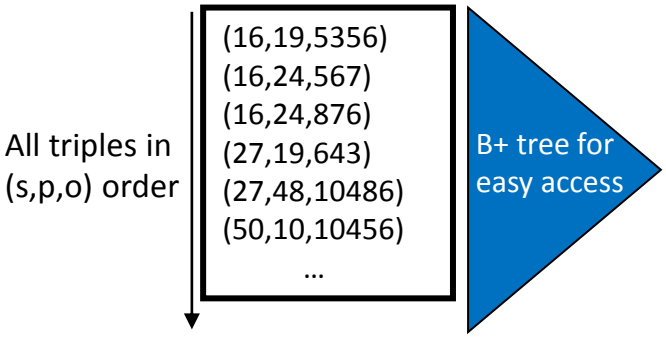
1. Lookup ids for constants:
   `Albert_Einstein` → 16, `invented` → 24
2. Lookup known prefix in index:
   (16,24,0)

Indexes for Commonly Used Triple Patterns

Patterns with a single variable are frequent

Example: `Albert_Einstein invented ?x`

⇒ Build **clustered index** over (s,p,o)



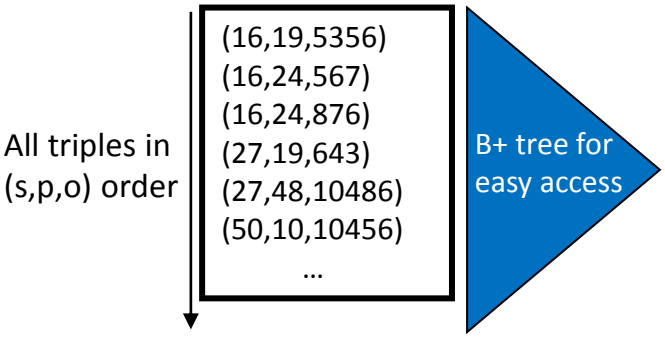
1. Lookup ids for constants:
   `Albert_Einstein` → 16, `invented` → 24
2. Lookup known prefix in index:
   (16,24,0)
3. Read results while prefix matches:
   (16,24,567), (16,24,876)
   come already sorted!

Indexes for Commonly Used Triple Patterns

Patterns with a single variable are frequent

Example: `Albert_Einstein invented ?x`

⇒ Build **clustered index** over (s,p,o)



1. Lookup ids for constants:
   `Albert_Einstein` → 16, `invented` → 24
2. Lookup known prefix in index:
   (16,24,0)
3. Read results while prefix matches:
   (16,24,567), (16,24,876)
   come already sorted!

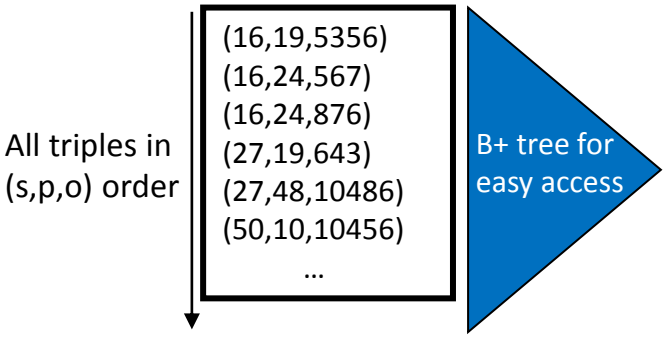
Can also be used for pattern like `Albert_Einstein ?p ?x`

Indexes for Commonly Used Triple Patterns

Patterns with a single variable are frequent

Example: `Albert_Einstein invented ?x`

⇒ Build **clustered index** over (s,p,o)



1. Lookup ids for constants:
   `Albert_Einstein` → 16, `invented` → 24
2. Lookup known prefix in index:
   (16,24,0)
3. Read results while prefix matches:
   (16,24,567), (16,24,876)
   come already sorted!

Can also be used for pattern like `Albert_Einstein ?p ?x`

Build similar clustered indexes for all **six permutations** (3 x 2 x 1 = 6)

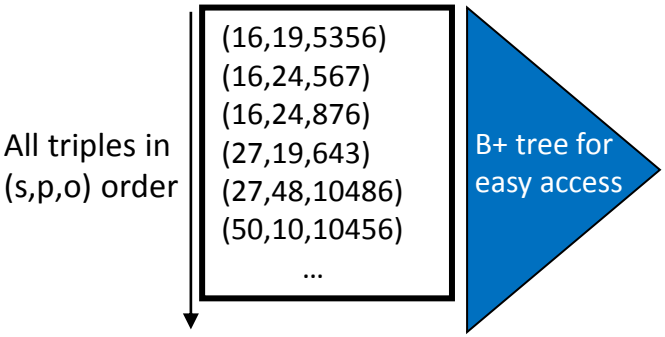
- SPO, POS, OSP to cover *all possible* triplet patterns
- SOP, OPS, PSO to have *all sort orders* for patterns with two var's

Indexes for Commonly Used Triple Patterns

Patterns with a single variable are frequent

Example: `Albert_Einstein invented ?x`

⇒ Build **clustered index** over (s,p,o)



1. Lookup ids for constants:
   `Albert_Einstein` → 16, `invented` → 24
2. Lookup known prefix in index:
   (16,24,0)
3. Read results while prefix matches:
   (16,24,567), (16,24,876)
   come already sorted!

Can also be used for pattern like `Albert_Einstein ?p ?x`

Build similar clustered indexes for all **six permutations** (3 x 2 x 1 = 6)

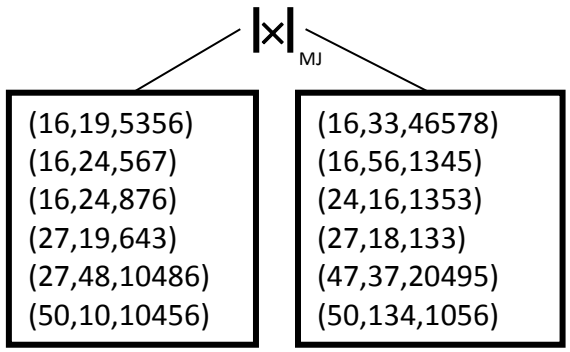
- SPO, POS, OSP to cover *all possible* triplet patterns
- SOP, OPS, PSO to have *all sort orders* for patterns with two var's

Triple table no longer needed, all triples in each index

Why Sort Order Matters for Joins

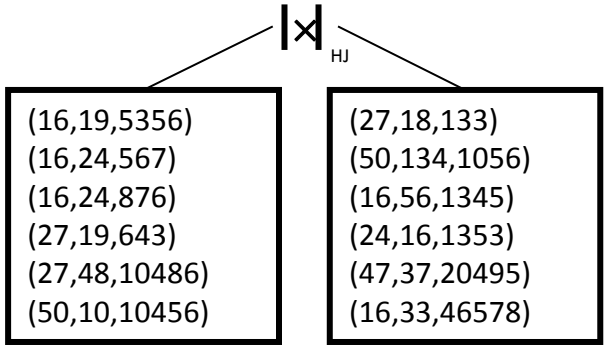
When inputs sorted by join attribute, use **Merge Join**:

- sequentially scan both inputs
- immediately join matching triples
- skip over parts without matches
- allows pipelining



When inputs are unsorted/sorted by wrong attribute, use **Hash Join**:

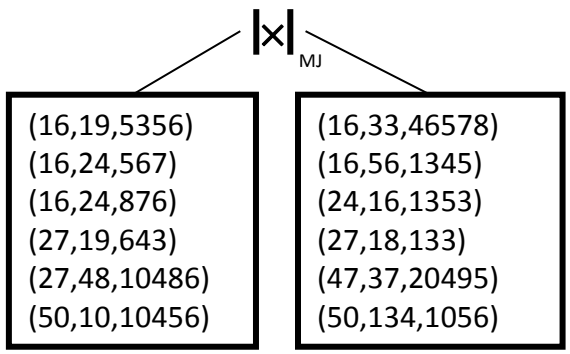
- build hash table from one input
- scan other input, probe hash table
- needs to touch every input triple
- breaks pipelining



Why Sort Order Matters for Joins

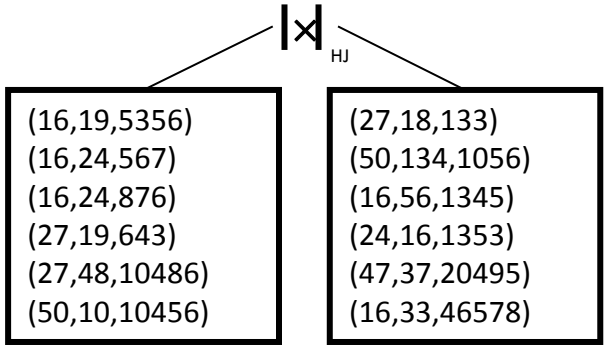
When inputs sorted by join attribute, use **Merge Join**:

- sequentially scan both inputs
- immediately join matching triples
- skip over parts without matches
- allows pipelining



When inputs are unsorted/sorted by wrong attribute, use **Hash Join**:

- build hash table from one input
- scan other input, probe hash table
- needs to touch every input triple
- breaks pipelining



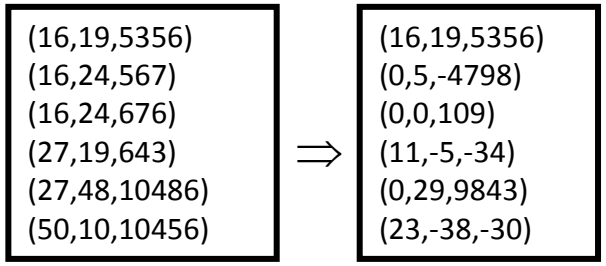
In general, Merge Joins are more preferable: small memory footprint, pipelining

RDF-3x: Compression Scheme for Triplets

- **Compress sequences of triples** in lexicographic order (v1;v2;v3); for SPO: v1=S, v2=P, v3=O

RDF-3x: Compression Scheme for Triplets

- **Compress sequences of triples** in lexicographic order (v1;v2;v3); for SPO: v1=S, v2=P, v3=O
- Step 1: compute **per-attribute deltas**

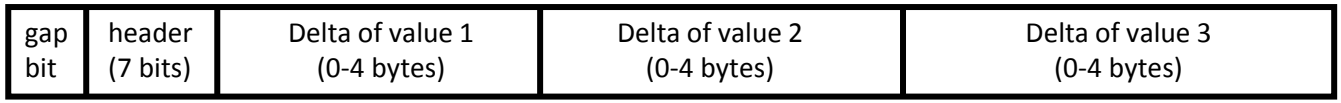


RDF-3x: Compression Scheme for Triplets

- **Compress sequences of triples** in lexicographic order (v1;v2;v3); for SPO: v1=S, v2=P, v3=O
- Step 1: compute **per-attribute deltas**

(16,19,5356) (16,24,567) (16,24,676) (27,19,643) (27,48,10486) (50,10,10456)	⇒	(16,19,5356) (0,5,-4798) (0,0,109) (11,-5,-34) (0,29,9843) (23,-38,-30)

- Step 2: **variable-byte encoding** for each delta triple



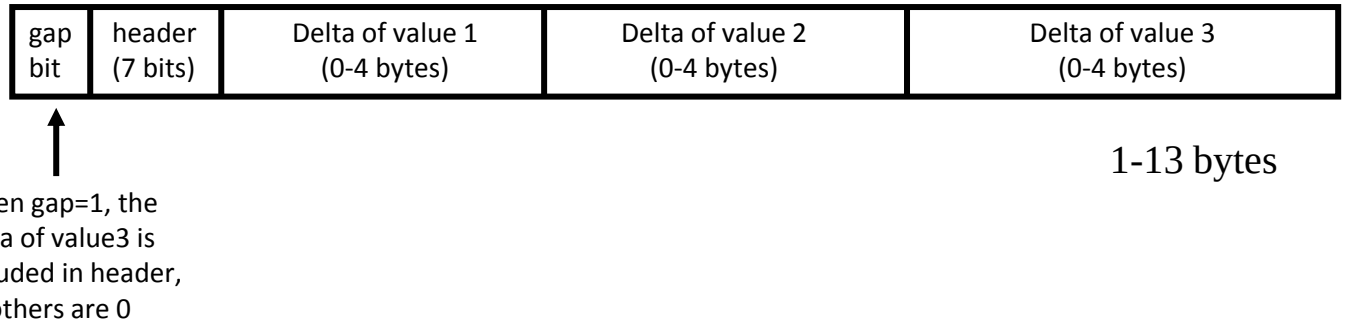
1-13 bytes

RDF-3x: Compression Scheme for Triplets

- **Compress sequences of triples** in lexicographic order (v1;v2;v3); for SPO: v1=S, v2=P, v3=O
- Step 1: compute **per-attribute deltas**

(16,19,5356) (16,24,567) (16,24,676) (27,19,643) (27,48,10486) (50,10,10456)	⇒	(16,19,5356) (0,5,-4798) (0,0,109) (11,-5,-34) (0,29,9843) (23,-38,-30)

- Step 2: **variable-byte encoding** for each delta triple

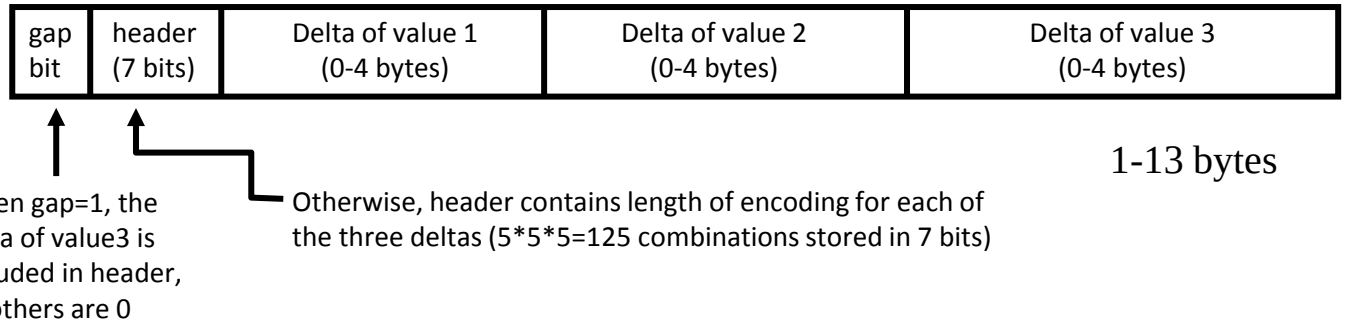


RDF-3x: Compression Scheme for Triplets

- **Compress sequences of triples** in lexicographic order (v1;v2;v3); for SPO: v1=S, v2=P, v3=O
- Step 1: compute **per-attribute deltas**

(16,19,5356) (16,24,567) (16,24,676) (27,19,643) (27,48,10486) (50,10,10456)	⇒	(16,19,5356) (0,5,-4798) (0,0,109) (11,-5,-34) (0,29,9843) (23,-38,-30)

- Step 2: **variable-byte encoding** for each delta triple

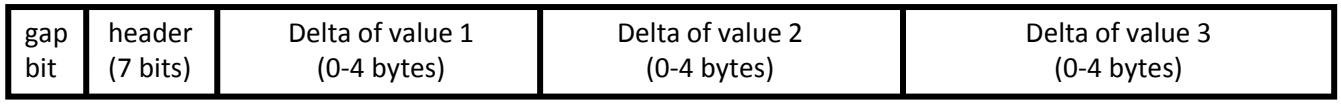


RDF-3x: Compression Scheme for Triplets

- **Compress sequences of triples** in lexicographic order (v1;v2;v3); for SPO: v1=S, v2=P, v3=O
- Step 1: compute **per-attribute deltas**

(16,19,5356) (16,24,567) (16,24,676) (27,19,643) (27,48,10486) (50,10,10456)	⇒	(16,19,5356) (0,5,-4798) (0,0,109) (11,-5,-34) (0,29,9843) (23,-38,-30)

- Step 2: **variable-byte encoding** for each delta triple



1-13 bytes

When gap=1, the delta of value3 is included in header, all others are 0

Otherwise, header contains length of encoding for each of the three deltas (5*5*5=125 combinations stored in 7 bits)

Many variants exist; this one is designed for triplets…

Compression Effectiveness vs. Efficiency

- Byte-level encoding almost as effective as bit-level encoding techniques (Gamma, Golomb, Rice, etc.)
- Much faster (10x) for decompressing
- Example for Barton dataset [Neumann & Weikum: VLDB'10]:
  - Raw data 51 million triples, 7GB uncompressed (as N-Triples)
  - All 6 main indexes:
    - 1.1GB size, 3.2s decompression with byte-level encoding
- Optionally: additional compression with LZ77 2x more compact, but much slower to decompress
  - Compression always on page level

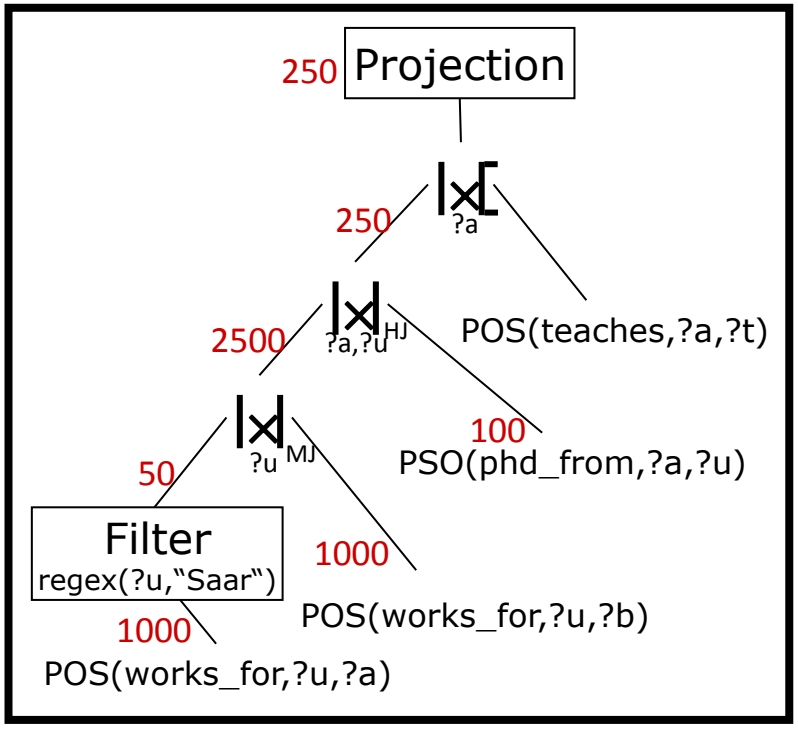
Back to the Example Query

SELECT ?a ?b ?t WHERE

{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }

OPTIONAL {?a teaches ?t}

FILTER (regex(?u, “Saar”))



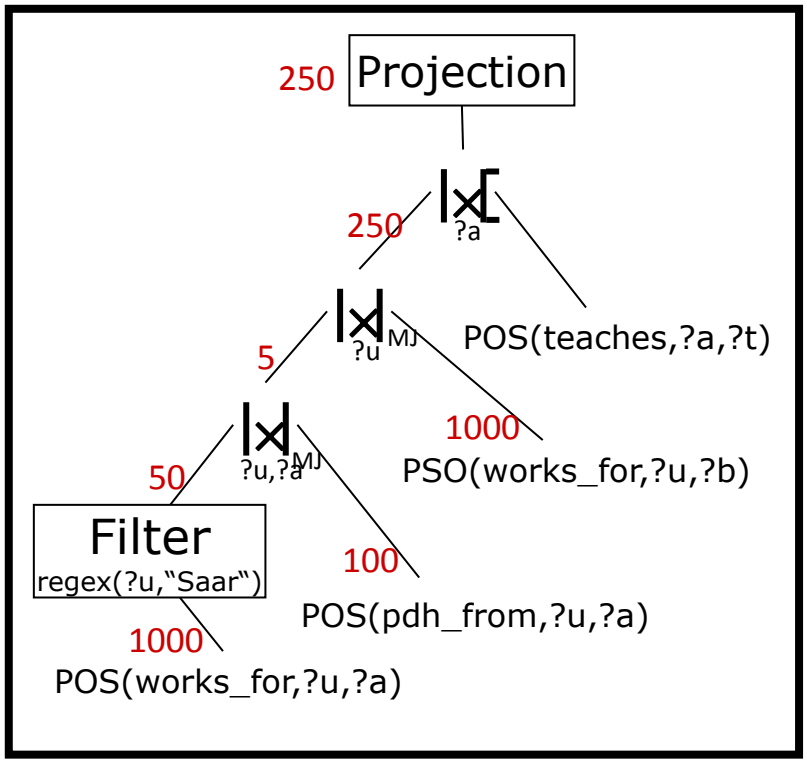
Back to the Example Query

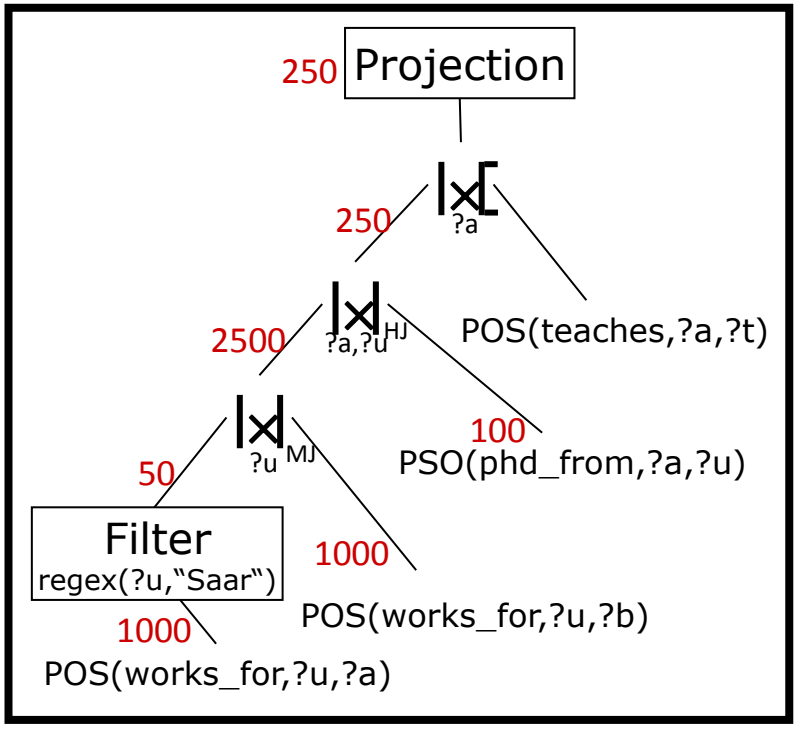
SELECT ?a ?b ?t WHERE

{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }

OPTIONAL {?a teaches ?t}

FILTER (regex(?u, "Saar"))





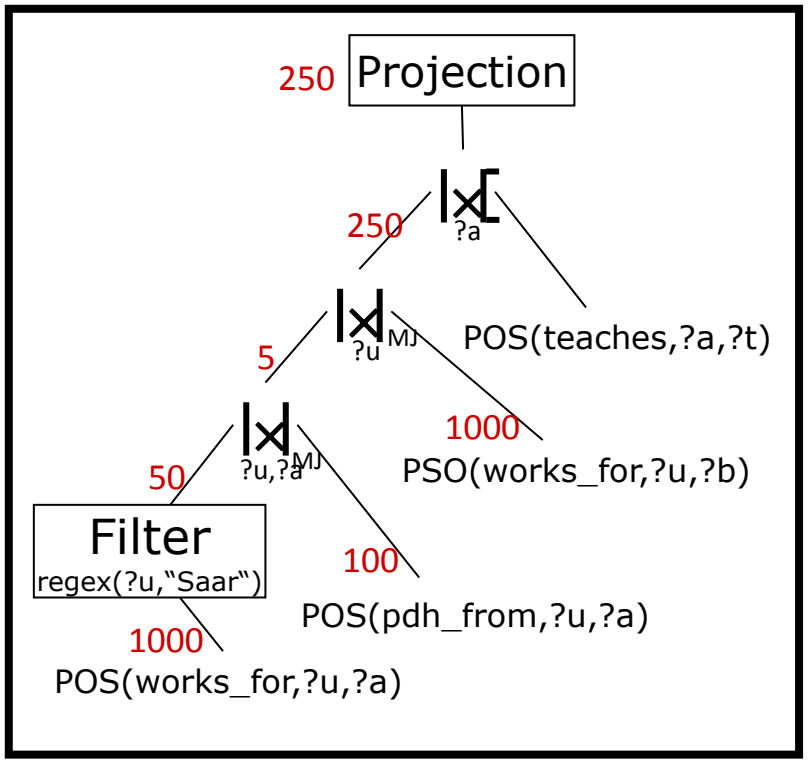
Back to the Example Query

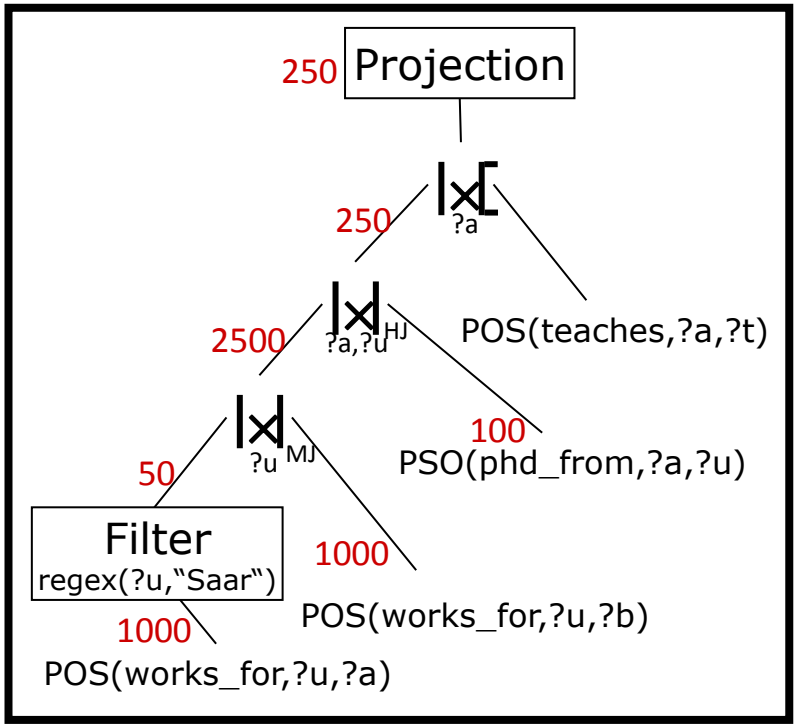
SELECT ?a ?b ?t WHERE

{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }

OPTIONAL {?a teaches ?t}

FILTER (regex(?u, "Saar"))





Which of the two plans is better?
How many intermediate results?

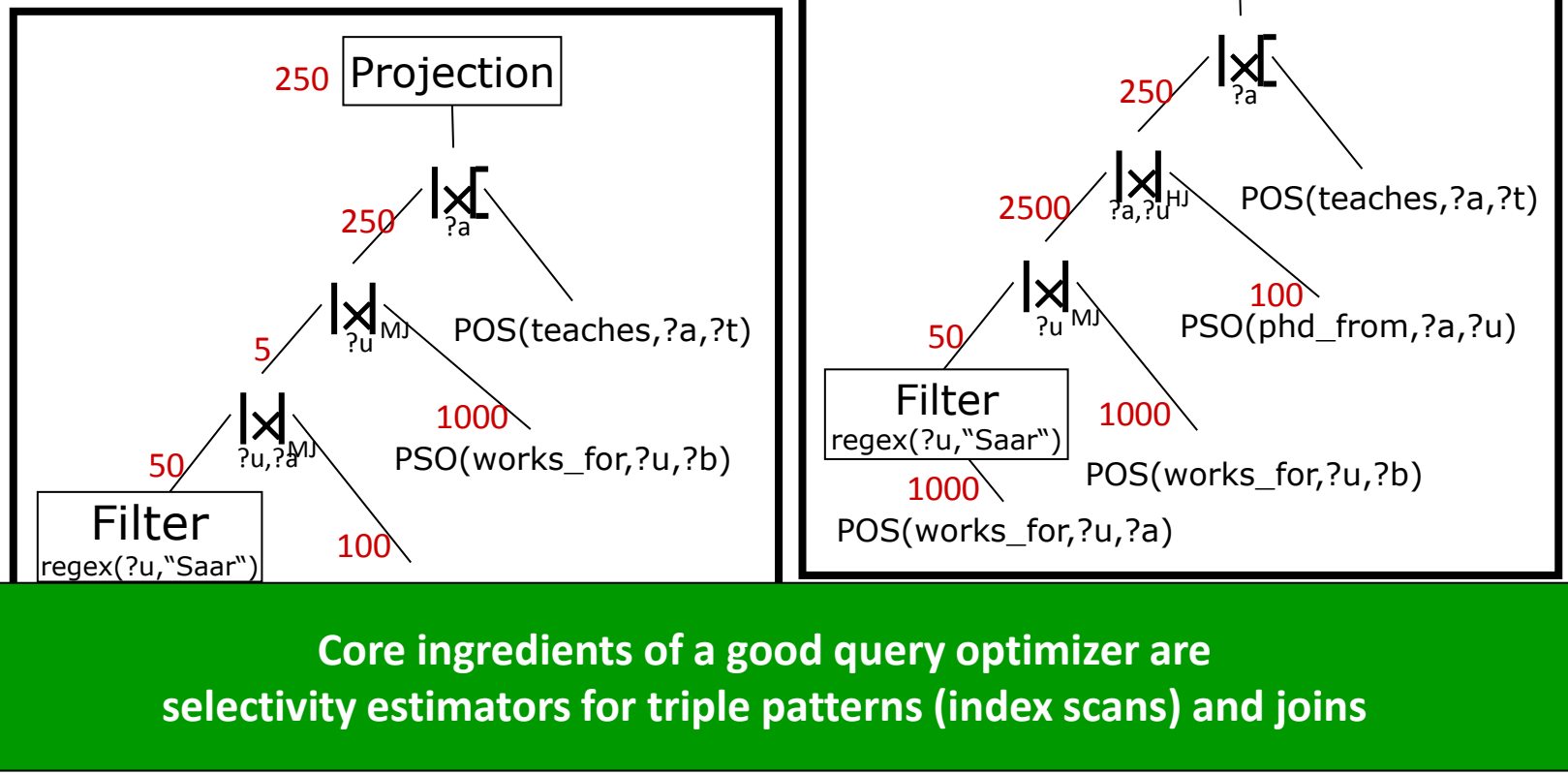
Back to the Example Query

SELECT ?a ?b ?t WHERE

{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }

OPTIONAL {?a teaches ?t}

FILTER (regex(?u, "Saar"))



Core ingredients of a good query optimizer are selectivity estimators for triple patterns (index scans) and joins

RDF-3x: Selectivity Estimation

How many results will a triple pattern have?

Standard databases:

- Per-attribute histograms
- Assume independence of attributes

} **too simplistic and inexact**

RDF-3x: Selectivity Estimation

How many results will a triple pattern have?

Standard databases:

- Per-attribute histograms
- Assume independence of attributes

} **too simplistic and inexact**

⇒ Use aggregated indexes for exact count

Additional **join statistics** for triple blocks (pages):

RDF-3x: Selectivity Estimation

How many results will a triple pattern have?

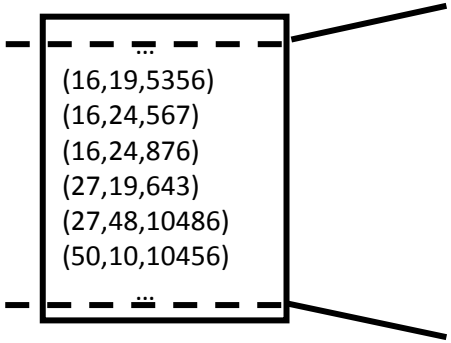
Standard databases:

- Per-attribute histograms
- Assume independence of attributes

} **too simplistic and inexact**

⇒ Use aggregated indexes for exact count

Additional **join statistics** for triple blocks (pages):



start (s,p,o)		end (s,p,o)	
number of triples			
number of distinct 2-prefixes			
number of distinct 1-prefixes			
join partners on subject			
	s=s	s=p	s=o
join partners on predicate			
	p=s	p=p	p=o
join partners on object			
	o=s	o=p	o=o

RDF-3x: Selectivity Estimation

How many results will a triple pattern have?

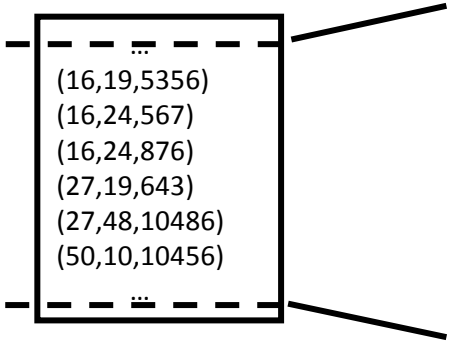
Standard databases:

- Per-attribute histograms
- Assume independence of attributes

} **too simplistic and inexact**

⇒ Use aggregated indexes for exact count

Additional **join statistics** for triple blocks (pages):



start (s,p,o)		end (s,p,o)	
number of triples			
number of distinct 2-prefixes			
number of distinct 1-prefixes			
join partners on subject			
	s=s	s=p	s=o
join partners on predicate			
	p=s	p=p	p=o
join partners on object			
	o=s	o=p	o=o

Assume independence between triple patterns; additionally precompute exact statistics for frequent paths in the data

Handling Updates

What should we do when our data changes?

(SPARQL 1.1 has updates!)

Handling Updates

What should we do when our data changes?

(SPARQL 1.1 has updates!)

Assumptions:

- Queries far more frequent than updates
- Updates mostly insertions, hardly any deletions
- Different applications may update concurrently

Handling Updates

What should we do when our data changes?

(SPARQL 1.1 has updates!)

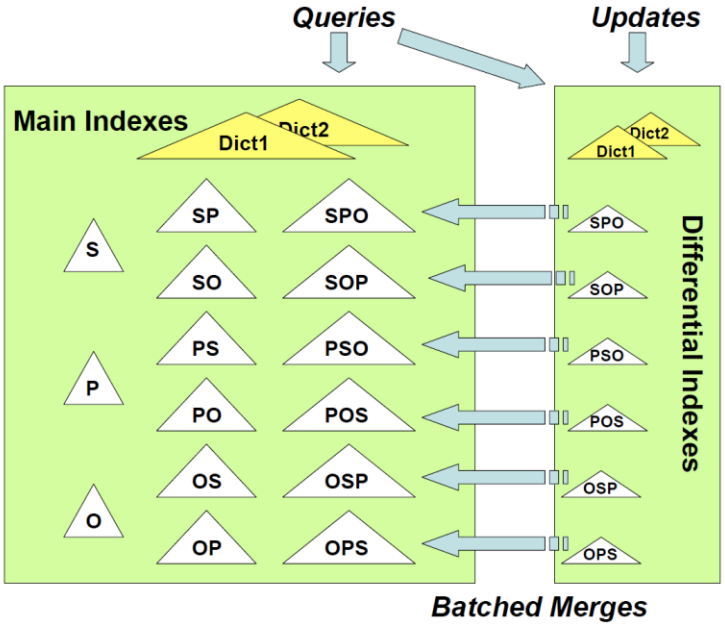
Assumptions:

- Queries far more frequent than updates
- Updates mostly insertions, hardly any deletions
- Different applications may update concurrently

Solution: Differential Indexing

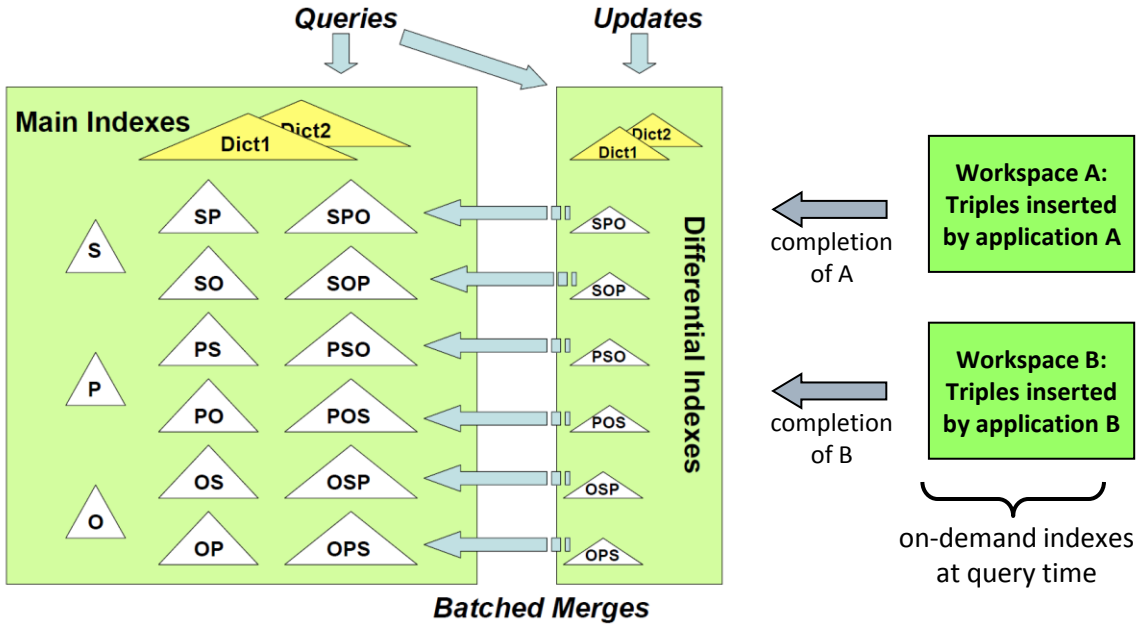
RDF-3x: Differential Updates

Staging architecture for updates in RDF-3X



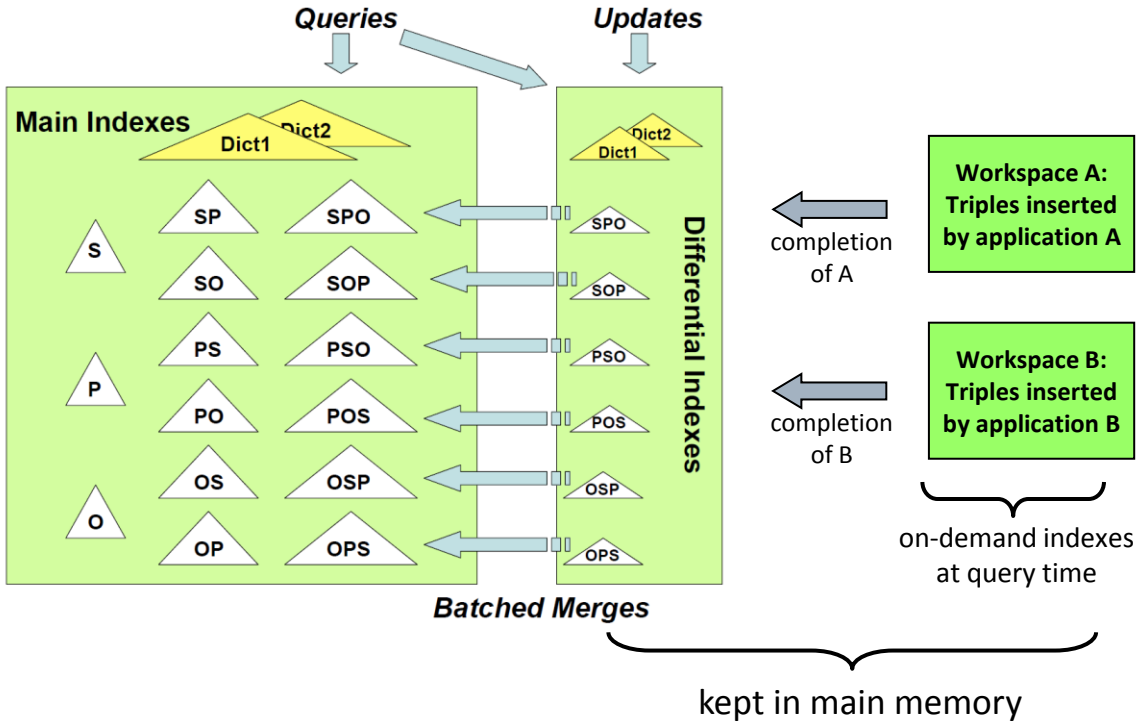
RDF-3x: Differential Updates

Staging architecture for updates in RDF-3X



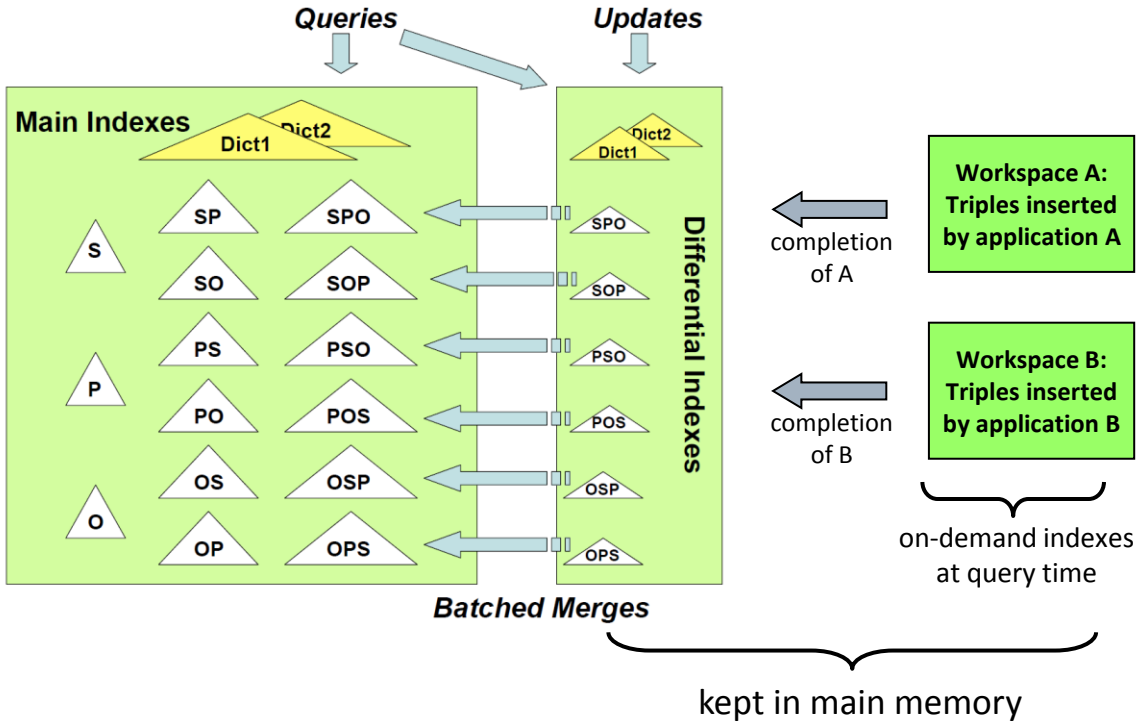
RDF-3x: Differential Updates

Staging architecture for updates in RDF-3X



RDF-3x: Differential Updates

Staging architecture for updates in RDF-3X



Deletions:

- Insert the same tuple again with “deleted” flag
- Modify scan/join operators: merge differential indexes with main index

Principles

Observations and assumptions:

- Not too many different predicates
- Triple patterns usually have fixed predicate
- Need to access all triples with one predicate

Principles

Observations and assumptions:

- Not too many different predicates
- Triple patterns usually have fixed predicate
- Need to access all triples with one predicate

Design consequence:

- Use one two-attribute table for each predicate

Example: Columnstores

ex:Katja ex:teaches ex:Databases;
ex:works_for ex:MPI_Informatics;
ex:PhD_from ex:TU_Ilmenau.
ex:Martin ex:teaches ex:Databases;
ex:works_for ex:MPI_Informatics;
ex:PhD_from ex:Saarland_University.
ex:Ralf ex:teaches ex:Information_Retrieval;
ex:PhD_from ex:Saarland_University;
ex:works_for ex:Saarland_University,
ex:MPI_Informatics.

works_for

subject	object
ex:Katja	ex:MPI_Informatics
ex:Martin	ex:MPI_Informtatics
ex:Ralf	ex:Saarland_University
ex:Ralf	ex:MPI_Informatics

teaches

subject	object
ex:Katja	ex:Databases
ex:Martin	ex:Databases
ex:Ralf	ex:Information_Retrieval

PhD_from

subject	object
ex:Katja	ex:TU_Ilmenau
ex:Martin	ex:Saarland_University
ex:Ralf	ex:Saarland_University

Simplified Example: Query Conversion

SELECT ?a ?b ?t WHERE

{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }

Simplified Example: Query Conversion

```
SELECT ?a ?b ?t WHERE
{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }
```

```
SELECT W1.subject as A, W2.subject as B
FROM works_for W1, works_for W2, phd_from P3
WHERE W1.object=W2.object
    AND W1.subject=P3.subject
    AND W1.object=P3.object
```

Simplified Example: Query Conversion

```
SELECT ?a ?b ?t WHERE
{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }
```

```
SELECT W1.subject as A, W2.subject as B
FROM works_for W1, works_for W2, phd_from P3
WHERE W1.object=W2.object
    AND W1.subject=P3.subject
    AND W1.object=P3.object
```

So far, this is yet another relational representation of RDF.
So, what is a columnstore?

Columnstores and RDF

Columnstores store *all* columns of a table separately.

Columnstores and RDF

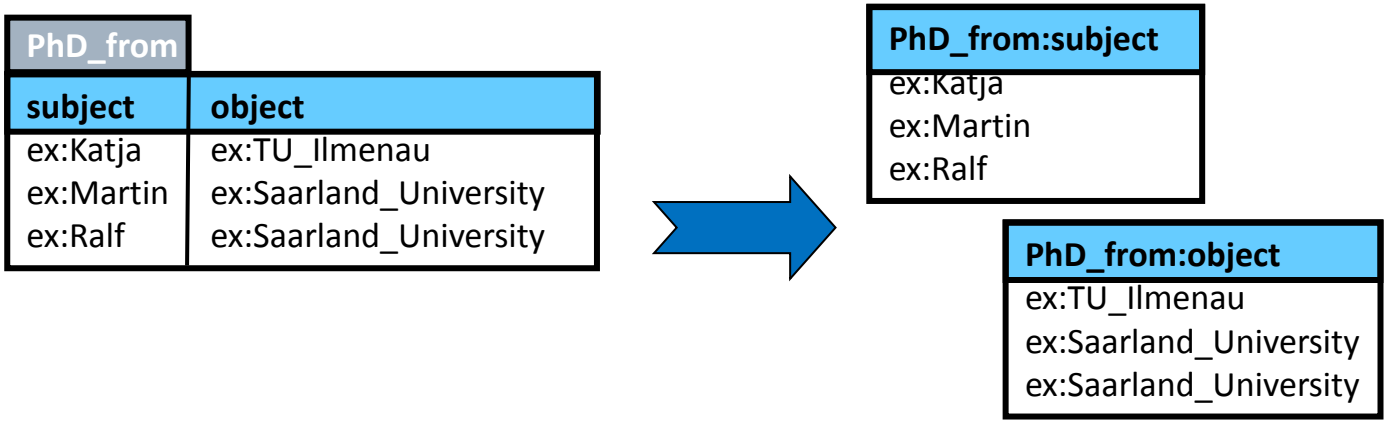
Columnstores store *all* columns of a table separately.

PhD_from

subject	object
ex:Katja	ex:TU_Ilmenau
ex:Martin	ex:Saarland_University
ex:Ralf	ex:Saarland_University

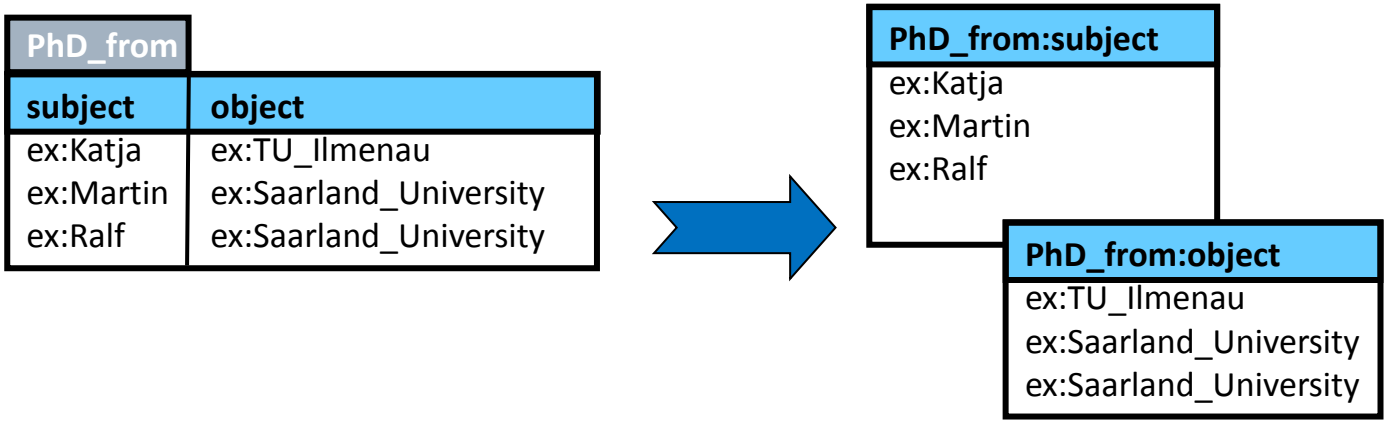
Columnstores and RDF

Columnstores store *all* columns of a table separately.



Columnstores and RDF

Columnstores store *all* columns of a table separately.

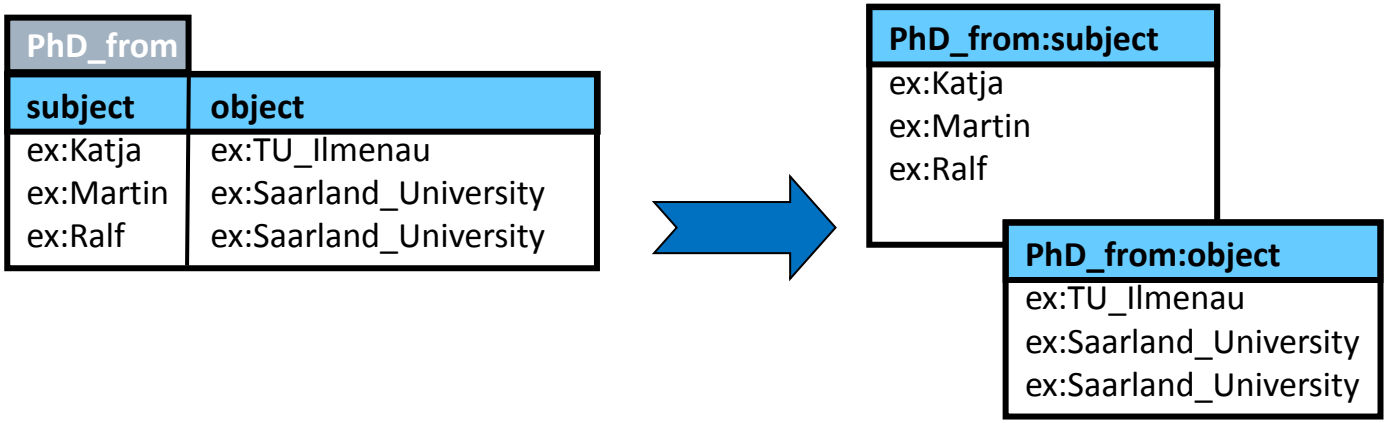


Advantages:

- **Fast if only subject or object are accessed, not both**
- **Allows for a very compact representation**

Columnstores and RDF

Columnstores store *all* columns of a table separately.



Advantages:
- **Fast if only subject or object are accessed, not both**
- **Allows for a very compact representation**

Problems:
- **Need to recombine columns if subject and object are accessed**
- **Inefficient for triple patterns with predicate variable**

Compression in Columnstores

General ideas:

- Store subject only once
- Use same order of subjects for all columns, including NULL values when necessary

Compression in Columnstores

General ideas:

- Store subject only once
- Use same order of subjects for all columns, including NULL values when necessary

subject
ex:Katja
ex:Martin
ex:Ralf
ex:Ralf

PhD_from
ex:TU_Ilmenau
ex:Saarland_University
ex:Saarland_University
NULL

teaches
ex:Databases
ex:Databases
ex:Information_Retrieval
NULL

works_for
ex:MPI_Informatics
ex:MPI_Informatics
ex:Saarland_University
ex:MPI_Informatics

Compression in Columnstores

General ideas:

- Store subject only once
- Use same order of subjects for all columns, including NULL values when necessary

subject
ex:Katja
ex:Martin
ex:Ralf
ex:Ralf

PhD_from
ex:TU_Ilmenau
ex:Saarland_University
ex:Saarland_University
NULL

teaches
ex:Databases
ex:Databases
ex:Information_Retrieval
NULL

works_for
ex:MPI_Informatics
ex:MPI_Informatics
ex:Saarland_University
ex:MPI_Informatics

- Additional compression to get rid of NULL values

PhD_from: bit[1110]
ex:TU_Ilmenau
ex:Saarland_University
ex:Saarland_University

Teaches: range[1-3]
ex:Databases
ex:Databases
ex:Information_Retrieval

Property Tables

Group entities with similar predicates into a relational table (for example using RDF types or a clustering algorithm).

```
ex:Katja    ex:teaches      ex:Databases;
            ex:works_for ex:MPI_Informatics;
            ex:PhD_from ex:TU_Ilmenau.
ex:Martin   ex:teaches      ex:Databases;
            ex:works_for ex:MPI_Informatics;
            ex:PhD_from ex:Saarland_University.
ex:Ralf     ex:teaches      ex:Information_Retrieval;
            ex:PhD_from ex:Saarland_University;
            ex:works_for ex:Saarland_University,
                           ex:MPI_Informatics.
```

Property Tables

Group entities with similar predicates into a relational table (for example using RDF types or a clustering algorithm).

```
ex:Katja    ex:teaches      ex:Databases;
            ex:works_for ex:MPI_Informatics;
            ex:PhD_from ex:TU_Ilmenau.
ex:Martin   ex:teaches      ex:Databases;
            ex:works_for ex:MPI_Informatics;
            ex:PhD_from ex:Saarland_University.
ex:Ralf     ex:teaches      ex:Information_Retrieval;
            ex:PhD_from ex:Saarland_University;
            ex:works_for ex:Saarland_University,
                         ex:MPI_Informatics.
```

subject	teaches	PhD_from
ex:Katja	ex:Databases	ex:TU_Ilmenau
ex:Martin	ex:Databases	ex:Saarland_University
ex:Ralf	ex:IR	ex:Saarland_University

subject	predicate	object
ex:Katja	ex:works_for	ex:MPI_Informatics
ex:Martin	ex:works_for	ex:MPI_Informatics
ex:Ralf	ex:works_for	ex:Saarland_University
ex:Ralf	ex:works_for	ex:MPI_Informatics

"Leftover triples"

Property Tables: Pros and Cons

Advantages:

- More in the spirit of existing relational systems
- Saves many self-joins over triple tables etc.

Property Tables: Pros and Cons

Advantages:

- More in the spirit of existing relational systems
- Saves many self-joins over triple tables etc.

Disadvantages:

- Potentially many NULL values
- Multi-value attributes problematic
- Query mapping depends on schema
- Schema changes very expensive

Even More Systems…

- Store RDF data as **sparse matrix** with bit-vector compression [BitMat, Hendler at al.: ISWC'09]
- Convert **RDF into XML** and use **XML methods** (XPath, XQuery, …)
- Store RDF data in **graph databases** and perform bi-simulation [Fletcher at al.: ESWC'12] or employ specialized graph index structures [gStore, Zou et al.: PVLDB'11]
- And many more …

Challenges and Opportunities

- SPARQL with **different entailment regimes**
- New **SPARQL 1.1 features**
  (grouping, aggregation, updates)
- User-oriented **ranking of query results**
  - Efficient top-*k* operators
  - Effective scoring methods for structured queries
- What are the **limits of a centralized RDF engine**?
- Dealing with **uncertain RDF data** –
  what is the most likely query answer?
  - Triples with probabilities → probabilistic databases

谢谢大家！

本课程课件由OpenKG提供支持

联系我们

小象学院：互联网新技术在线教育领航者

- 微信公众号：小象
- 新浪微博：ChinaHadoop