5、索引和限制

Index and Constraints

图数据库需要索引吗?

- Neo4j是“本地图数据库”(Native Graph Database)，也就是说Neo4j以节点、关系和属性的形式物理地保存数据。与此相对比，关系型数据库按照表、字段和行保存数据。
- 关系型数据库在对邻接表的主键和外键进行连接(JOIN)时，通常需要建立索引，以提高键值匹配的效率。
- Neo4j在保存数据的同时也保存数据之间的关系。因此它实现的一个重要突破叫做“无需索引的邻接跳转”(Index-free Adjacency)。简单地说，在从一个节点到另一个节点、沿着关系的方向查询、或者说遍历图时，只需要读取保存的起始节点的地址、然后进行跳转就可以了，无须做连接(JOIN)。这也是Neo4j在处理复杂和深度关系遍历型查询时性能远远超过任何关系型数据库的关键所在。

图数据库需要索引吗？(续)

- 因此说，在Neo4j“本地图数据库”中，是没有连接、主键、外键的概念的，因为不需要！
- 但是，Neo4j中仍然有索引(Index)！
- Neo4j中可以对属性建立索引，以实现快速的节点或关系查询。例如节点‘人物’拥有属性‘姓名’，那么对‘姓名’建立索引可以快速地根据姓名找到相应的‘人物’节点。
- 另外，Neo4j自动对节点的标签，以及关系的类型建立了索引。

索引的类型

- Neo4j中可以定义下面三类索引

1) Legacy Index / 遗留索引，又称Explicit/明确索引：

在3.0版本前使用的、基于Lucene的索引，目前已经不再推荐使用。

2) Schema(Native) Index / 数据库模式(或称本地)索引：

与关系型数据库实现机制类似、由数据库管理的索引。数据库模式索引支持节点属性和基本数据类型，包括字符串、数字、布尔、浮点、空间坐标等。索引不能在列表型数据上建立。

3) External Index / 外部索引：

基于Lucene的索引，使用**APOC过程**创建和维护，支持节点和关系属性、以及全文索引。

创建数据库模式索引

- 数据库模式索引只能定义在节点属性上：

```
CREATE INDEX ON :`人物`(name);
```

- 数据库模式索引支持基本数据类型，如字符串、数字、布尔、浮点、空间坐标等；不能对列表类型的节点创建索引。
- 数据库模式索引在<database-name>/schema/index目录下。
- 在Neo4j浏览器中，使用下面的命令查看所有模式索引：

```
CALL db.indexes      // 显示所有索引
:schema              // 显示索引和限制
```

- 模式索引由数据库自动维护，包括更新、重建、备份、恢复。

创建数据库模式索引(续)

- 数据库模式索引支持复合索引，即可包含多个节点属性：

```
CREATE INDEX ON :`人物`(name,title);
```

注意：我们现有的节点中并没有title属性。Neo4j是Schema-Lite(轻型模式数据库)，可以在属性还不存在的时候就为其定义索引。

- 查询时，Cypher查询生成器会自动寻找相关索引并应用到对属性值的搜索中。
- 对于单属性索引，WHERE子句中使用=、<>、>、<、IN、STARTS WITH、ENDS WITH、CONTAINS、exists()时都会自动应用索引；
- 对于复合索引，WHERE子句中使用=、IN时会自动应用索引。

更新数据库模式索引

- 数据库模式索引在创建后会自动更新；
- 在数据量较大时，索引的创建会需要一定的时间，这时查询可能会出现“索引未准备好”的错误；
- 可以调用以下过程来等待索引建立完成：

```
CALL db.awaitIndex(":`人物`(name)");    // 等待一个索引完成
CALL db.awaitIndexes();                 // 等待所有索引完成
```

注意：这是企业版才有的特性。

创建和使用外部索引

- 外部索引又称“手工索引”(Manual Index)，是APOC实现的基于Apache Lucene的全文索引；
- 外部索引同时支持节点和关系的属性；
- 外部索引支持Lucene的模糊查询条件；
- 全文索引不需要特别安装Apache Lucene服务器，相关软件包已经包含在Neo4j和APOC中；
- 可以调用以下过程来建立节点属性的外部索引：

```
MATCH (a:`人物`)
CALL apoc.index.addNode(a,['name'])    // 索引名即标签名
RETURN count(*)
```

创建和使用外部索引(续)

- 使用下面的过程建立关系属性的索引：

```
MATCH (:`人物`)-[r:`关系`]->(:`人物`)
CALL apoc.index.addRelationship(r,['relationship'])    // 索引名即关系名
RETURN count(*);
```

- 查询已经建立的外部索引：

```
CALL apoc.index.list();
```

- 使用指定索引查询节点：

```
CALL apoc.index.nodes('人物','name:刘*');
```

创建和使用外部索引(续)

- 使用指定索引查询关系：

```
CALL apoc.index.relationships('关系','relationship:兄*');
```

- 外部索引在<database_name>/index/lucene目录下；
- 外部索引可以自动更新、也可以手动更新。在neo4j.conf中：

```
apoc.autoIndex.enabled=true   # 即时/同步更新索引
apoc.autoIndex.async=true     # 或者异步更新索引
```

更多外部索引的使用和设置请参见APOC文档：

https://neo4j-contrib.github.io/neo4j-apoc-procedures/#_text_and_lookup_indexes 。

创建和使用遗留索引

- 遗留索引，或者显式索引是Neo4j 3.0以前的索引技术。除非是从以前版本的数据库移植过来的，现在已经不推荐使用。
- 使用db.index.explicit.*过程来管理和维护遗留索引。例如：

```
MATCH (a:`人物`)
CALL db.index.explicit.addNode('Person',a,'name','刘备') YIELD success
RETURN success;
```

- 更多关于遗留索引的使用，请参见Neo4j Developer Manual: https://neo4j.com/docs/developer-manual/current/cypher/schema/index/#db.index.explicit.addNode

定义限制

- Neo4j是“轻型模式”数据库(Schema-Lite)，除了索引，限制(Constraint)是另外一个属于数据库模式的概念；
- 节点的限制包括以下类型：
  - 属性唯一性限制
  - 属性存在性限制
  - 复合键唯一性限制
- 关系的限制包括以下类型：
  - 关系属性存在性限制

定义节点属性限制

- 节点属性的唯一性限制：

```
CREATE CONSTRAINT ON (person:`人物`) ASSERT person.name IS UNIQUE;
```

- 节点属性存在性限制：

```
CREATE CONSTRAINT ON (person:`人物`) ASSERT exists(person.name);
```

- 节点复合键(多个属性)唯一性限制：

```
CREATE CONSTRAINT ON (person:`人物`)
ASSERT (person.name, person.title) IS NODE KEY;
```

注意：创建唯一性限制时，数据库会同时自动在相关属性上创建索引。如果属性已经有索引，必须先删除索引后再创建限制。

定义关系属性限制

- 关系属性的存在性限制：

```
CREATE CONSTRAINT ON ()-[r:`关系`]->() ASSERT exists(r.relationship);
```

- 不支持对关系属性定义唯一性限制；
- 查看已定义的限制：

```
CALL db.constraints;  // 仅显示限制
:schema               // 显示索引和限制
```

总结

- Neo4j实现了本地图数据库，数据的物理存储、查询和展示都是基于统一的图数据模型。
- Neo4j的“无索引邻接跳转”使得关联数据不需要做连接(JOIN)，因此性能大大提高。
- Neo4j是“轻型模式数据库”(Schema-Lite)，提供对索引和限制的支持。
- Neo4j提供的节点属性索引，目的是为了根据属性值快速定位关系遍历的起始节点；而其提供的关系属性索引，更多的是用来对关系进行过滤。
- 使用APOC创建外部索引，可以实现对存储在属性中的文本内容进行全文检索。

总结(续)

- 使用CREATE INDEX ON创建数据库模式/本地索引；使用DROP INDEX ON删除索引。
- 使用CREATE CONSTRAINT ON创建限制；使用DROP CONSTRAINT ON删除限制。
- Neo4j数据库根据查询自动应用索引，以实现最高的查询效率。在特定情况下，也可以在查询中使用USING INDEX来指定要使用的索引，以获得更好的性能。这会在后续的“查询优化”部分介绍。