简介

“Cypher”是一个描述性的图形查询语言，允许不必编写图形结构的遍历代码对图形存储有表现力和效率的查询。Cypher 还在继续发展和成熟，这也就意味着有可能会出现语法的变化。同时也意味着作为组件没有经历严格的性能测试。

Cypher 设计的目的是一个人类查询语言，适合于开发者和在数据库上做点对点模式（ad-hoc）查询的专业操作人员。它的构建是基于英语单词和灵巧的图解。

Cypher 通过一系列不同的方法和建立于确定的实践为表达查询而激发的。许多关键字如 like 和 order by 是受 SQL 的启发。模式匹配的表达式来自于 SPARQL。正则表达式匹配实现实用 Scala programming language 语言。

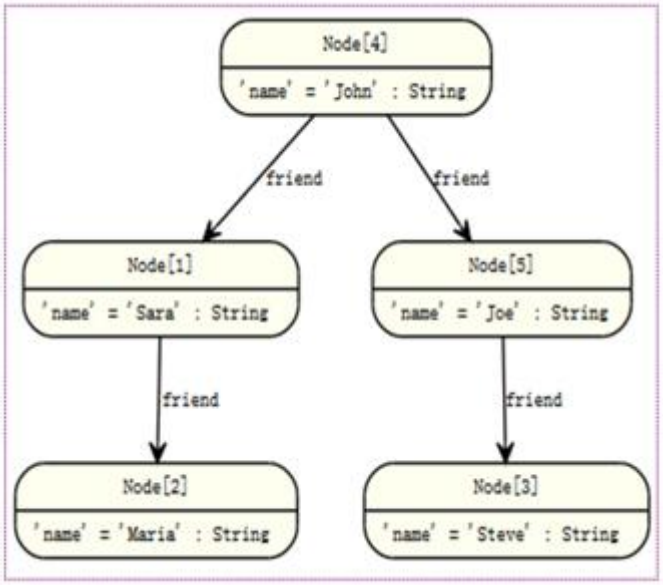
Cypher 是一个申明式的语言。对比命令式语言如 Java 和脚本语言如 Gremlin 和 JRuby，它的焦点在于从图中如何找回（what to retrieve），而不是怎么去做。这使得在不对用户公布的实现细节，关心的是怎么优化查询。

这个查询语言包含以下几个明显的部分：

- START：在图中的开始点，通过元素的 ID 或索引查找获得。
- MATCH：图形的匹配模式，束缚于开始点。
- WHERE：过滤条件。
- RETURN：返回所需要的。

在下例中看三个关键字

示例图片如下：



如：这个有个查询，通过遍历图，找到索引里一个叫 John 的朋友的朋友（不是他的直接朋友），返回 John 和找到的朋友的朋友。

START john=node:node_auto_index(name = 'John')

```
MATCH john-[:friend]->()-[:friend]->fof
RETURN john, fof
```

返回结果：

john	fof
Node[4]{name->"John"}	Node[2]{name->"Maria"}
Node[4]{name->"John"}	Node[3]{name->"Steve"}
2 rows, 73 ms	

下一步添加过滤：
在下一个例子中，列出一组用户的 id，并遍历图查找这些用户接出 friend 关系线，返回有属性 name 并且其值是以 S 开始的用户。

```
START user=node(5,4,1,2,3)
MATCH user-[:friend]->follower
WHERE follower.name =~ /S.*/
RETURN user, follower.name
```

返回结果：

user	follower.name
Node[5]{name->"Joe"}	"Steve"
Node[4]{name->"John"}	"Sara"
2 rows, 4 ms	

操作符

Cypher 中的操作符有三个不同种类：数学，相等和关系。
数学操作符有+，-，*，/和%。当然只有+对字符有作用。
等于操作符有=，<>，<，>，<=，>=。

因为 Neo4j 是一个模式少的图形数据库，Cypher 有两个特殊的操作符?和!。

有些是用在属性上，有些事用于处理缺少值。对于一个不存在的属性做比较会导致错误。为替代与其他什么做比较时总是检查属性是否存在，在缺失属性时问号将使得比较总是返回 true，感叹号使得比较总是返回 false。

```
WHEREn.prop? = "foo"
```

这个断言在属性缺失情况下将评估为 true。

WHEREn.prop! = "foo"
这个断言在属性缺失情况下将评估为 false。

警告：在同一个比较中混合使用两个符号将导致不可预料的结果。

参数

Cypher 支持带参数的查询。这允许开发者不需要必须构建一个 string 的查询，并且使得 Cypher 的查询计划的缓存更容易。

参数可以在 where 子句，start 子句的索引 key 或索引值，索引查询中作为节点/关系 id 的引用。

以下是几个在 java 中使用参数的示例：

节点 id 参数

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "id", 0 );
ExecutionResult result = engine.execute( "start n=node({id}) return n.name", params );
```

节点对象参数

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "node", andreasNode );
ExecutionResult result = engine.execute( "start n=node({node}) return n.name", params );
```

多节点 id 参数

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "id", Arrays.asList( 0, 1, 2 ) );
```

```
ExecutionResult result = engine.execute( "start n=node({id}) return n.name", params );
```

字符串参数

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "name", "Johan" );
ExecutionResult result = engine.execute( "start n=node(0,1,2) where n.name = {name} return n",
params );
```

索引键值参数

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "key", "name" );
params.put( "value", "Michaela" );
ExecutionResult result = engine.execute( "start n=node:people({key} = {value}) return n", params );
```

索引查询参数

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "query", "name:Andreas" );
ExecutionResult result = engine.execute( "start n=node:people({query}) return n", params );
```

SKIP 与 LIMIT * 的数字参数

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "s", 1 );
params.put( "l", 1 );
ExecutionResult result = engine.execute( "start n=node(0,1,2) return n.name skip {s} limit {l}",
params );
```

正则表达式参数

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "regex", ".*h.*" );
ExecutionResult result = engine.execute( "start n=node(0,1,2) where n.name =~ {regex} return n.name", params );
```

标识符

当你参考部分的模式时，需要通过命名完成。定义的不同的命名部分就被称为标识符。
如下例中：

```
START n=node(1) MATCH n-->b RETURN b
```

标识符为 n 和 b。

标识符可以是大写或小些，可以包含下划线。当需要其他字符时可以使用`符号。对于属性名的规则也是一样。

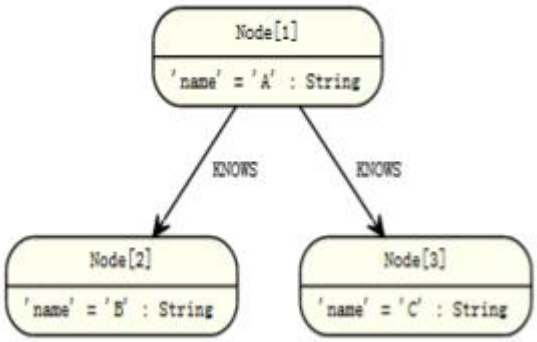
注解

可以在查询语句中使用双斜杠来添加注解。如：

```
START n=node(1) RETURN b //这是行结束注释
START n=node(1) RETURN b
START n=node(1) WHERE n.property = "//这部是一个注释" RETURN b
```

Start

每一个查询都是描述一个图案（模式），在这个图案（模式）中可以有多个限制点。一个限制点是为模式匹配的从开始点出发的一条关系或一个节点。可以通过 id 或索引查询绑定点。



通过 id 绑定点

通过 node(*)函数绑定一个节点作为开始点
查询：

```
START n=node(1)
RETURN n
```

返回引用的节点。

结果：

n
Node[1]{name->"A"}
1 rows, 0 ms

通过 id 绑定关系

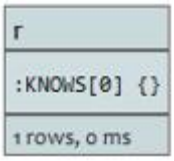
可以通过 relationship()函数绑定一个关系作为开始点。也可以通过缩写 rel()。

查询：

```
START r=relationship(0)
RETURN r
```

Id 为 0 的关系将被返回

结果：

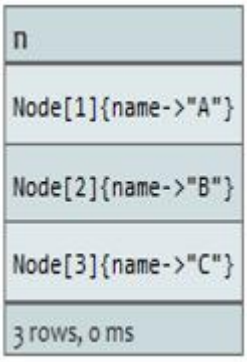


通过 id 绑定多个节点

选择多个节点可以通过逗号分开。
查询：

```
START n=node(1, 2, 3)
RETURN n
```

结果：



所有节点

得到所有节点可以通过星号（*），同样对于关系也适用。

查询：

```
START n=node(*)
RETURN n
```

这个查询将返回图中所有节点。

结果：

n
Node[1]{name->"A"}
Node[2]{name->"B"}
Node[3]{name->"C"}
3 rows, 0 ms

通过索引查询获取节点

如果开始节点可以通过索引查询得到，可以如此来写：
node:index-name(key=”value”)。在此列子中存在一个节点索引叫 nodes。

查询：

```
START n=node:nodes(name = "A")
RETURN n
```

索引中命名为 A 的节点将被返回。

结果：

n
Node[1]{name->"A"}
1 rows, 2 ms

通过索引查询获取关系

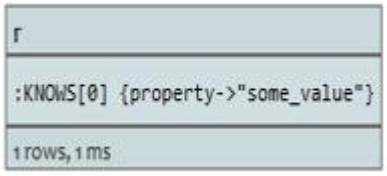
如果开始点可以通过索引查询得到，可以如此做：
Relationship:index-name(key=”value”)。

查询：

```
START r=relationship:rels(property ="some_value")
RETURN r
```

索引中属性名为”some_value”的关系将被返回。

结果：

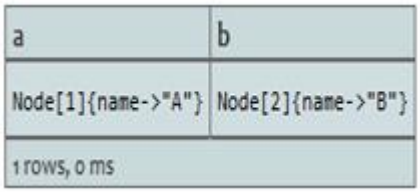


多个开始点

有时需要绑定多个开始点。只需要列出并以逗号分隔开。
查询：

```
START a=node(1), b=node(2)
RETURN a,b
```

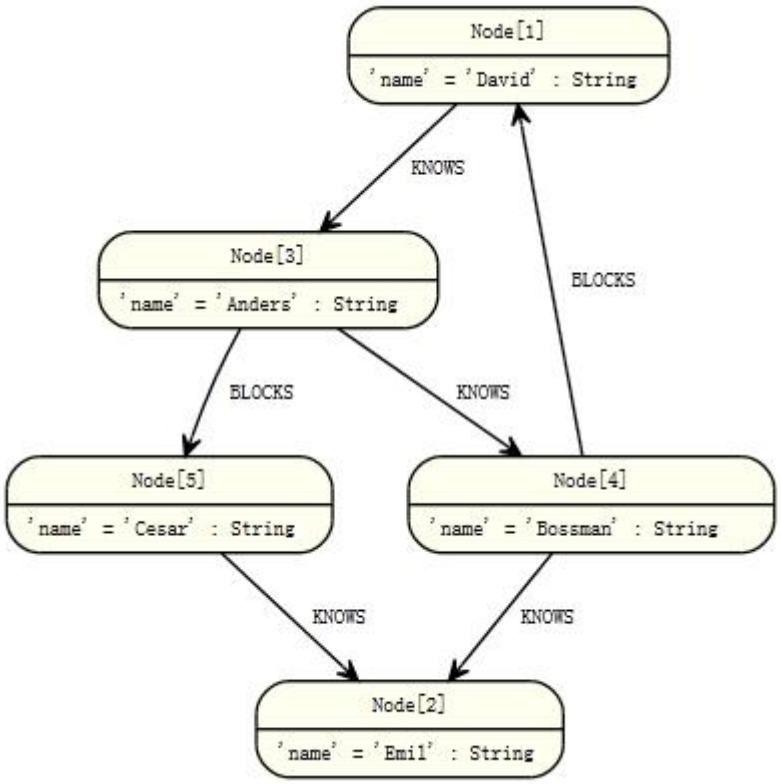
A 和 B 两个节点都将被返回。
结果：



Match

在一个查询的匹配（match）部分申明图形（模式）。模式的申明导致一个或多个以逗号隔开的路径（path）。
节点标识符可以使用或者不是用圆括号。使用圆括号与不使用圆括号完全对等，如：
MATCH(a)-->(b) 与 MATCH a-->b 匹配模式完全相同。
模式的所有部分都直接或者间接地绑定到开始点上。可选关系是一个可选描述模式的方法，但在真正图中可能没有匹配（节点可能没有或者没有此类关系时），将被估值为 null。与 SQL 中的外联结类似，如果 Cypher 发现一个或者多个匹配，将会全部返回。如果没有匹配，Cypher 将返回 null。
如以下例子，b 和 p 都是可选的并都可能包含 null：

```
START a=node(1) MATCH p = a-[?]->b
START a=node(1) MATCH p = a-[*?]->b
START a=node(1) MATCH p = a-[?]->x-->b
START a=node(1), x=node(100) MATCH p = shortestPath( a-[*?]->x )
```



相关节点

符号—意味着相关性，不需要关心方向和类型。

查询：

```
START n=node(3)
MATCH (n)--(x)
RETURN x
```

所有与 A 相关节点都被返回。

结果：

x
Node[4]{name->"Bossman"}
Node[1]{name->"David"}
Node[5]{name->"Cesar"}
3 rows, 1 ms

接出关系（Outgong relationship）

当对关系的方向感兴趣时，可以使用-->或<--符号，如：

查询：
START n=node(3)
MATCH (n)-->(x)
RETURN x

所有 A 的接出关系到达的节点将被返回.
结果：

x
Node[4]{name->"Bossman"}
Node[5]{name->"Cesar"}
2 rows, 1 ms

定向关系和标识符

如果需要关系的标识符，为了过滤关系的属性或为了返回关系，可如下例使用标识符。

查询：
START n=node(3)
MATCH (n)-[r]->()
RETURN r

所有从节点 A 接出的关系将被返回。

结果：

r
:KNOWS[0] {}
:BLOCKS[1] {}
2 rows, 1 ms

通过关系类型匹配

当已知关系类型并想通过关系类型匹配时，可以通过冒号详细描述。
查询：
START n=node(3)
MATCH (n)-[:BLOCKS]->(x)
RETURN x

返回 A 接出关系类型为 BLOCKS 的节点。

结果：

x
Node[5]{name->"Cesar"}
1 rows, 1 ms

通过关系类型匹配和使用标识符

如果既想获得关系又要通过已知的关系类型，那就都添加上，如：

查询：

```
START n=node(3)
MATCH (n)-[r:BLOCKS]->()
RETURN r
```

所有从 A 接出的关系为 BLOCKS 的关系都被返回。

结果：

r
:BLOCKS[1] {}
1 rows, 1 ms

带有特殊字符的关系类型

有时候数据库中有非字母字符类型，或有空格在内时，使用单引号。

查询：

```
START n=node(3)
MATCH (n)-[r:`TYPE WITH SPACE IN IT`]->()
RETURN r
```

返回类型有空格的关系。

结果：

r
:TYPE WITH SPACE IN IT[6] {}
1 rows, 1 ms

多重关系

关系可以通过使用在()—()多个语句来表达，或可以串在一起。如下：

查询:

```
START a=node(3)
MATCH (a)-[:KNOWS]->(b)-[:KNOWS]->(c)
RETURN a,b,c
```

路径中的三个节点。

结果：

a	b	c
Node[3]{name->"Anders"}	Node[4]{name->"Bossman"}	Node[2]{name->"Emil"}
1 rows, 2 ms		

可变长度的关系

可变数量的关系->节点可以使用-[:TYPE*minHops..maxHops]->。

查询：

```
START a=node(3), x=node(2, 4)
MATCH a-[:KNOWS*1..3]->x
RETURN a,x
```

如果在 1 到 3 的关系中存在路径，将返回开始点和结束点。

结果：

a	x
Node[3]{name->"Anders"}	Node[2]{name->"Emil"}
Node[3]{name->"Anders"}	Node[4]{name->"Bossman"}
2 rows, 22 ms	

在可变长度关系的关系标识符

当连接两个节点的长度是可变的不确定的时，可以使用一个关系标识符遍历所有关系。

查询：

```
START a=node(3), x=node(2, 4)
MATCH a-[r:KNOWS*1..3]->x
RETURN r
```

如果在 1 到 3 的关系中存在路径，将返回开始点和结束点。
结果：

r
[:KNOWS[0] {},:KNOWS[3] {}]
[:KNOWS[0] {}]
2 rows, 1 ms

零长度路径

当使用可变长度路径，可能其路径长度为 0，这也就是说两个标识符指向的为同一个节点。
如果两点间的距离为 0，可以确定这是同一个节点。

查询：

```
START a=node(3)
MATCH p1=a-[:KNOWS*0..1]->b, p2=b-[:BLOCKS*0..1]->c
RETURN a,b,c, length(p1), length(p2)
```

这个查询将返回四个路径，其中有些路径长度为 0.

结果：

a	b	c	length(p1)	length(p2)
Node[3] {name->"Anders"}	Node[3] {name->"Anders"}	Node[3] {name->"Anders"}	0	0
Node[3] {name->"Anders"}	Node[3] {name->"Anders"}	Node[5] {name->"Cesar"}	0	1
Node[3] {name->"Anders"}	Node[4] {name->"Bossman"}	Node[4] {name->"Bossman"}	1	0
Node[3] {name->"Anders"}	Node[4] {name->"Bossman"}	Node[1] {name->"David"}	1	1
4 rows, 5 ms				

可选关系

如果关系为可选的，可以使用问号表示。与 SQL 的外连接类似。如果关系存在，将被返回。如果不存在在其位置将以 null 代替。

查询：

```
START a=node(2)
MATCH a-[?]->x
RETURN a,x
```

返回一个节点和一个 null，因为这个节点没有关系。

结果：

a	x
Node[2]{name->"Emil"}	<null>
1 rows, 3 ms	

可选类型和命名关系

通过一个正常的关系，可以决定哪个标识符可以进入，那些关系类型是需要的。

查询：

```
START a=node(3)
MATCH a-[r?:LOVES]->()
```

```
RETURN a,r
```

返回一个节点和一个 null，因为这个节点没有关系。

结果：

a	r
Node[3]{name->"Anders"}	<null>
1 rows, 1 ms	

可选元素的属性

返回可选元素上的属性，null 值将返回 null。

查询：

```
START a=node(2)
MATCH a-[?]->x
RETURN x, x.name
```

元素 x 在查询中为 null，所有其属性 name 为 null。
结果：

x	x.name
<null>	<null>
1 rows, 0 ms	

复杂匹配

在 Cypher 中，可哟通过更多复杂模式来匹配，像一个钻石形状模式。

查询：

```
START a=node(3)
MATCH (a)-[:KNOWS]->(b)-[:KNOWS]->(c),(a)-[:BLOCKS]-(d)-[:KNOWS]-(c)
RETURN a,b,c,d
```

路径中的四个节点。

结果：

a	b	c	d
Node[3] {name->"Anders"}	Node[4] {name->"Bossman"}	Node[2] {name->"Emil"}	Node[5] {name->"Cesar"}
1 rows, 3 ms			

最短路径

使用 shortestPath 函数可以找出一条两个节点间的最短路径，如下。

查询：

```
START d=node(1), e=node(2)
MATCH p = shortestPath( d-[*..15]->e )
RETURN p
```

这意味着：找出两点间的一条最短路径，最大关系长度为 15.圆括号内是一个简单的路径连接，开始节点，连接关系和结束节点。关系的字符描述像关系类型，最大数和方向在寻找最短路径中都将被用到。也可以标识路径为可选。

结果：

p
(1)--[KNOWS,2]-->(3)--[KNOWS,0]-->(4)--[KNOWS,3]-->(2)
1 rows, 11 ms

所有最但路径

找出两节点节点所有的最短路径。

查询：

```
START d=node(1), e=node(2)
MATCH p = allShortestPaths( d-[*..15]->e )
RETURN p
```

这将在节点 d 与 e 中找到两条有方向的路径。
结果：

p
(1)--[KNOWS,2]-->(3)--[KNOWS,0]-->(4)--[KNOWS,3]-->(2)
(1)--[KNOWS,2]-->(3)--[BLOCKS,1]-->(5)--[KNOWS,4]-->(2)
2 rows, 2 ms

命名路径

如果想在模式图上的路径进行过滤或者返回此路径，可以使用命名路径（named path）。

查询：

```
START a=node(3)
MATCH p = a-->b
RETURN p
```

开始节点的两个路径。

结果：

p
[Node[3]{name->"Anders"},:KNOWS[0] {},Node[4]{name->"Bossman"}]
[Node[3]{name->"Anders"},:BLOCKS[1] {},Node[5]{name->"Cesar"}]
2 rows, 5 ms

在绑定关系上的匹配

当模式中包含一个绑定关系时，此关系模式没有明确的方向，Cypher 将尝试着切换连接节点的边匹配关系。

查询：

```
START a=node(3), b=node(2)
MATCH a-[?:KNOWS]-x-[?:KNOWS]-b
RETURN x
```

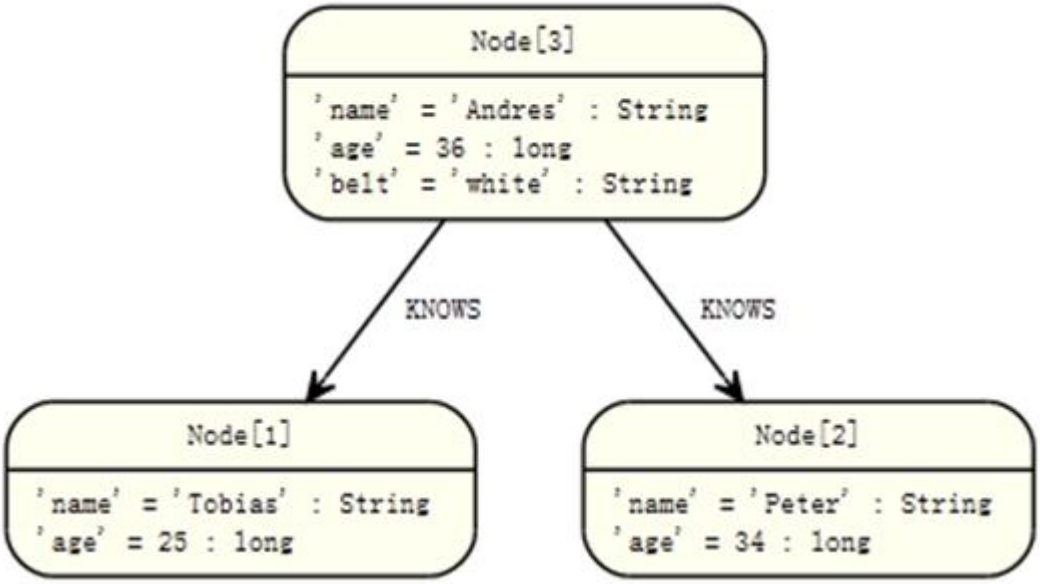
将返回两个连接节点，一次为开始节点，一次为结束节点。

结果：

x
Node[4]{name->"Bossman"}
Node[1]{name->"David"}
Node[5]{name->"Cesar"}
3 rows, 8 ms

Where

如果需要从查找的数据的图中过滤，可以在查询语句中添加 where 子句。
图：



Boolean 操作类型

可以使用 boolean 操作符 and 和 or 或者也可以使用 not（）函数。
查询：

```
START n=node(3, 1)
WHERE (n.age < 30 and n.name = "Tobias") ornot(n.name = "Tobias")
RETURN n
```

返回节点。

结果：

n
Node[3]{name->"Andres",age->36,belt->"white"}
Node[1]{name->"Tobias",age->25}
2 rows, 0 ms

节点属性上的过滤

查询：

```
START n=node(3, 1)
WHERE n.age < 30
RETURN n
```

结果：

n
Node[1]{name->"Tobias",age->25}
1 rows, 0 ms

正则表达式

可以通过使用=~ /regexp/来匹配正在表达式。如下：

查询：

```
START n=node(3, 1)
WHERE n.name =~ /Tob.*/
RETURN n
```

返回名叫 Tobias 的节点。

结果：

n
Node[1]{name->"Tobias",age->25}
1 rows, 1 ms

转义正则表达式

如果在正则表达式中需要有斜杠时可以通过转义实现。

查询：

```
START n=node(3, 1)
WHERE n.name =~ /Some\/thing/
RETURN n
```

没有匹配的节点返回。

结果：

n
(empty result)
0 rows, 0 ms

不分大小些正则表达式

在正则表达式前加上?i，整个正则表达式将会忽略大小写。

查询：

```
START n=node(3, 1)
WHERE n.name =~ /(?i)ANDR.*/
RETURN n
```

属性 name 为 Andres 的节点将返回

结果：

n
Node[3]{name->"Andres",age->36,belt->"white"}
1 rows, 0 ms

关系类型上的过滤

可以 match 模式中通过添加具体的关系类型，但有时需要针对类型的更加高级的过滤。可以使用明确的 type 属性来对比，查询对关系类型名作一个正则比较。

查询：

```
START n=node(3)
MATCH (n)-[r]->()
WHERE type(r) =~ /K.*/
RETURN r
```

关系整个以 K 开始的类型名都将返回。

结果：

r
:KNOWS[0] {}
:KNOWS[1] {}
2 rows, 0 ms

属性存在性

查询：

```
START n=node(3, 1)
WHERE n.belt
RETURN n
```

结果：

n
Node[3]{name->"Andres",age->36,belt->"white"}
1 rows, 0 ms

如果缺失属性默认为 true

仅当属性存在时，比较一个图的元素的此属性，使用允许空属性的语法。

查询：

```
START n=node(3, 1)
WHERE n.belt? = 'white'
RETURN n
```

所有节点即使没有 belt 属性的 都将返回。此类比较返回为 true。

结果：

n
Node[3]{name->"Andres",age->36,belt->"white"}
Node[1]{name->"Tobias",age->25}
2 rows, 0 ms

如果缺失属性默认为 false

需要在缺失属性时为 false，即不想返回此属性不存在的节点时。使用感叹号。

查询：

```
START n=node(3, 1)
WHERE n.belt! = 'white'
RETURN n
```

结果：

n
Node[3]{name->"Andres",age->36,belt->"white"}
1 rows, 1 ms

空置 null 过滤

有时候需要测试值或者标识符是否为 null。与 sql 类似使用 is null 或 not（is null x）也能起作用。

查询：

```
START a=node(1), b=node(3, 2)
MATCH a<-[r?]-b
WHERE r is null
RETURN b
```

Tobias 节点没有链接上。

结果：

b
Node[2]{name->"Peter",age->34}
1 rows, 2 ms

关系过滤

为过滤两点间基于关系的子图，在 match 子句中使用限制部分。可以描述带方向的关系和可能的类型。这些都是有效的表达：WHERE a-→b WHERE a←-b WHERE a←[:KNOWS]-bWHERE a-[:KNOWS]-b

查询：

```
START a=node(1), b=node(3, 2)
WHERE a<--b
RETURN b
```

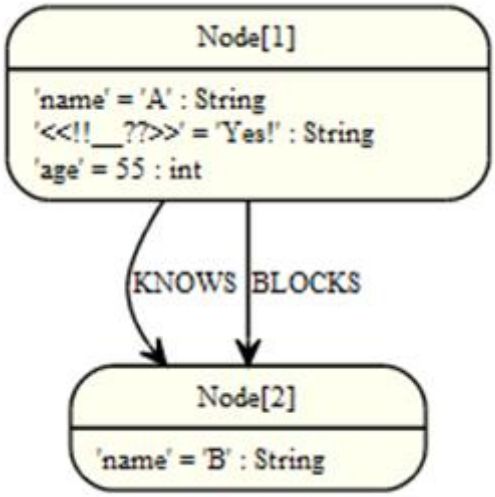
Tobias 节点没有链接
结果：

b
Node[3]{name->"Andres",age->36,belt->"white"}
1 rows, 1 ms

Return

查询中的返回部分，返回途中定义的感兴趣的部分。可以为节点、关系或其上的属性。

图



返回节点

返回一个节点，在返回语句中列出即可。
查询：

```
START n=node(2)
RETURN n
```

结果：

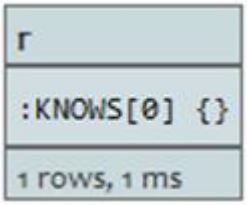
n
Node[2]{name->"B"}
1 rows, 0 ms

返回关系

查询：

```
START n=node(1)
MATCH (n)-[r:KNOWS]->(c)
RETURN r
```

结果：

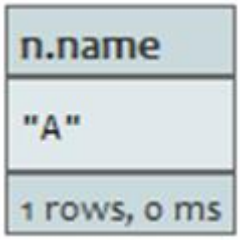


返回属性

查询：

```
START n=node(1)
RETURN n.name
```

结果：



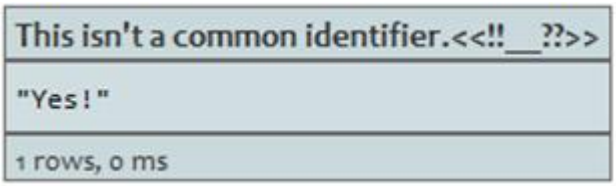
带特殊字符的标识符

使用不在英语字符表中的字符，可以使用'单引号。

查询：

```
START `This isn't a commonidentifier`=node(1)
RETURN `This isn't a commonidentifier`.`<<!!__??>>`
```

结果：



列的别名

可以给展示出来的列名起别名。

查询：

```
START a=node(1)
RETURN a.age AS SomethingTotallyDifferent
```

返回节点的 age 属性，但重命名列名。

结果：

SomethingTotallyDifferent
55
1 rows, 0 ms

可选属性

属性在节点上可能存在也可能不存在，可以使用问号来标识标识符即可。

查询：

```
START n=node(1, 2)
RETURN n.age?
```

如果存在 age 属性，则返回，不存在则返回 null。

结果：

n.age
55
<null>
2 rows, 0 ms

特别的结果

DISTINCT 仅检索特别的行，基于选择输出的列。

查询：

```
START a=node(1)
MATCH (a)-->(b)
```

RETURN distinct b

返回 name 为 B 的节点，但仅为一次。

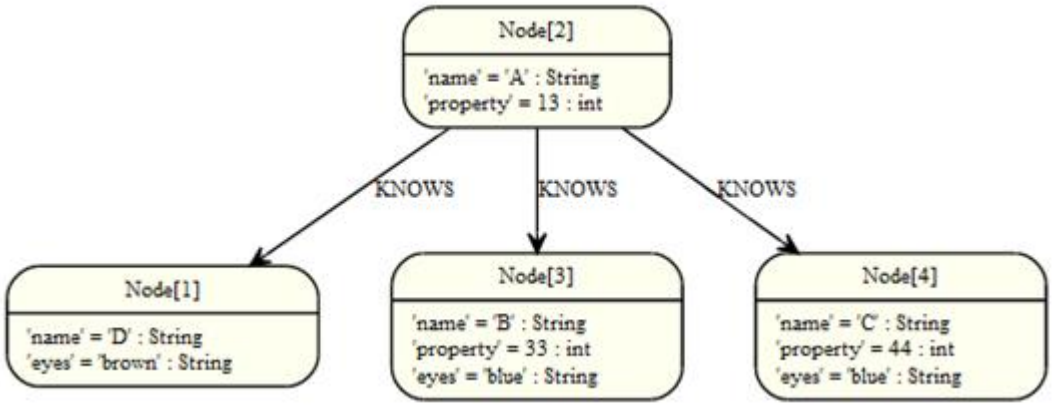
结果：

b
Node[2]{name->"B"}
1 rows, 1 ms

聚合（Aggregation）

为集合计算数据，Cypher 提供聚类功能，与 SQL 的 group by 类似。在 return 语句中发现的任何聚类函数，所有没有聚类函数的列将作为聚合 key 使用。

图：



计数

计数（count）使用来计算行数。Count 有两种使用方法。Count（*）计算匹配的行的行数，count（<标识符>）计算标识符中非空值数。

计算节点数

计算链接到一个节点的节点数，可以使用 count（*）。

查询：

```
START n=node(2)
MATCH (n)-->(x)
RETURN n, count(*)
```

返回开始节点和相关节点节点数。

结果：

n	count(*)
Node[2]{name->"A",property->13}	3
1 rows, 1 ms	

分组计算关系类型

计算分组了得关系类型，返回关系类型并使用 count（*）计算。

查询：

```
START n=node(2)
MATCH (n)-[r]->()
RETURN type(r), count(*)
```

返回关系类型和其分组数。

结果：

TYPE(r)	count(*)
"KNOWS"	3
1 rows, 1 ms	

计算实体数

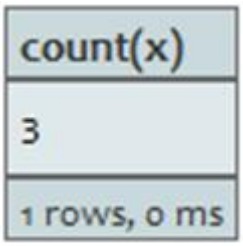
相比使用 count（*），可能计算标识符更实在。

查询：

```
START n=node(2)
MATCH (n)-->(x)
RETURN count(x)
```

返回链接到开始节点上的节点数

结果：

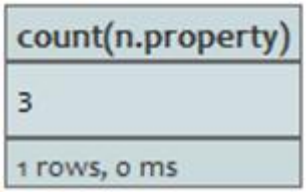


计算非空可以值数

查询：

```
START n=node(2,3,4,1)
RETURN count(n.property?)
```

结果：



求和（sum）

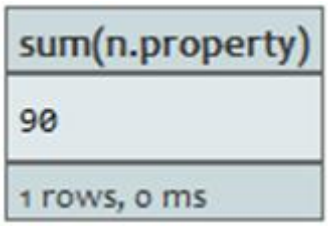
Sum 集合简单计算数值类型的值。Null 值将自动去掉。如下：

查询：

```
START n=node(2,3,4)
RETURN sum(n.property)
```

计算所有节点属性值之和。

结果：



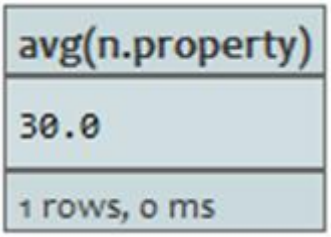
平均值（avg）

Avg 计算数量列的平均值

查询：

```
START n=node(2,3,4)
RETURN avg(n.property)
```

结果：



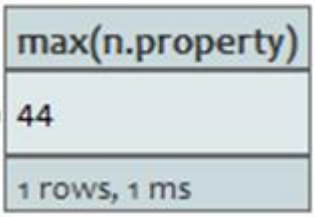
最大值（max）

Max 查找数字列中的最大值。

查询：

```
START n=node(2,3,4)
RETURN max(n.property)
```

结果：



最小值（min）

Min 使用数字属性作为输入，并返回在列中最小的值。

查询：

```
START n=node(2,3,4)
RETURN min(n.property)
```

结果：

min(n.property)
13
1 rows, 1 ms

聚类（COLLECT）

Collect 将所有值收集到一个集合 list 中。

查询：

```
START n=node(2,3,4)
RETURN collect(n.property)
```

返回一个带有所有属性值的简单列。

结果：

collect(n.property)
List(13, 33, 44)
1 rows, 0 ms

相异（DISTINCT）

聚合函数中使用 distinct 来去掉值中重复的数据。

查询：

```
START a=node(2)
MATCH a-->b
```

RETURN count(distinct b.eyes)

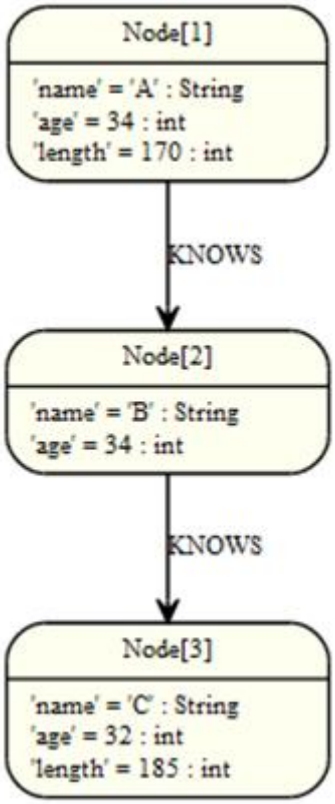
结果：

count(distinct b.eyes)
2
1 rows, 1 ms

排序（Order by）

输出结果排序可以使用 order by 子句。注意，不能使用节点或者关系排序，仅仅只针对其属性有效。

图：



通过节点属性排序节点

查询：

START n=node(3,1,2)

```
RETURN n
ORDER BY n.name
```

结果：

n
Node[1]{name->"A",age->34,length->170}
Node[2]{name->"B",age->34}
Node[3]{name->"C",age->32,length->185}
3 rows, 0 ms

通过多节点属性排序节点

在 order by 子句中可以通过多个属性来排序每个标识符。Cypher 首先将通过第一个标识符排序，如果第一个标识符或属性相等，则在 order by 中检查下一个属性，依次类推。

查询：

```
START n=node(3,1,2)
RETURN n
ORDER BY n.age, n.name
```

首先通过 age 排序，然后再通过 name 排序。

结果：

n
Node[3]{name->"C",age->32,length->185}
Node[1]{name->"A",age->34,length->170}
Node[2]{name->"B",age->34}
3 rows, 1 ms

倒序排列节点

可以在标识符后添加 desc 或 asc 来进行倒序排列或顺序排列。
查询：

```
START n=node(3,1,2)
RETURN n
```

ORDER BY n.name DESC

结果：

n
Node[3]{name->"C",age->32,length->185}
Node[2]{name->"B",age->34}
Node[1]{name->"A",age->34,length->170}
3 rows, 0 ms

空值排序

当排列结果集时，在顺序排列中 null 将永远放在最后，而在倒序排列中放最前面。

查询：

START n=node(3,1,2)
RETURN n.length?, n
ORDER BY n.length?

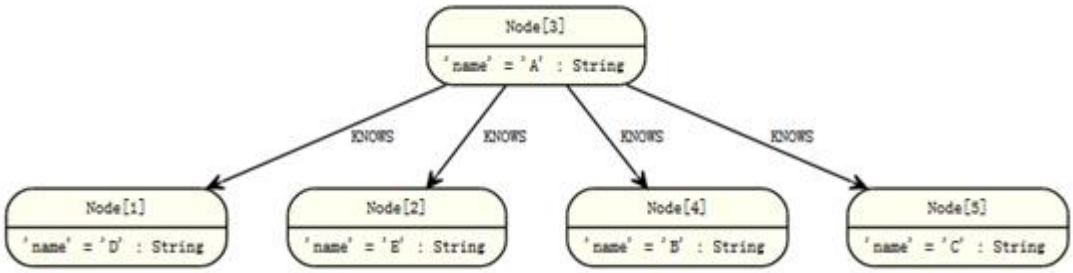
结果：

n.length	n
170	Node[1]{name->"A",age->34,length->170}
185	Node[3]{name->"C",age->32,length->185}
<null>	Node[2]{name->"B",age->34}
3 rows, 0 ms	

Skip

Skip 允许返回总结果集中的一个子集。此不保证排序，除非使用了 order by’子句。

图：



跳过前三个

返回结果中一个子集，从第三个结果开始，语法如下：
查询：

```
START n=node(3, 4, 5, 1, 2)
RETURN n
ORDER BY n.name
SKIP 3
```

前三个节点将略过，最后两个节点将被返回。

结果：

n
Node[1]{name->"D"}
Node[2]{name->"E"}
2 rows, 0 ms

返回中间两个

查询：

```
START n=node(3, 4, 5, 1, 2)
RETURN n
ORDER BY n.name
SKIP 1
LIMIT 2
```

中间两个节点将被返回。

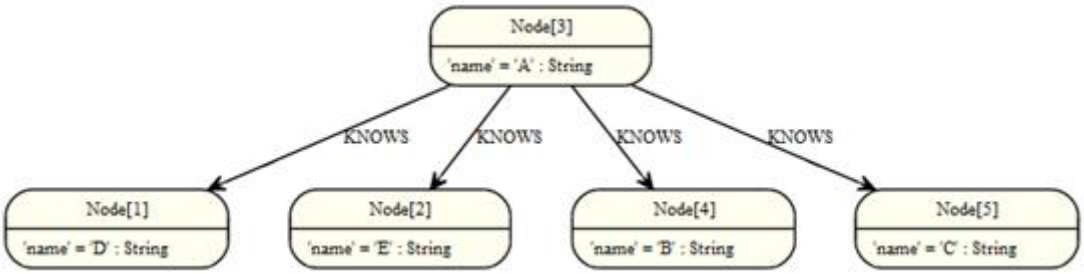
结果：

n
Node[4]{name->"B"}
Node[5]{name->"C"}
2 rows, 0 ms

Limit

Limit 允许返回结果集中的一个子集。

图：



返回第一部分

查询：

```
START n=node(3, 4, 5, 1, 2)
RETURN n
LIMIT 3
```

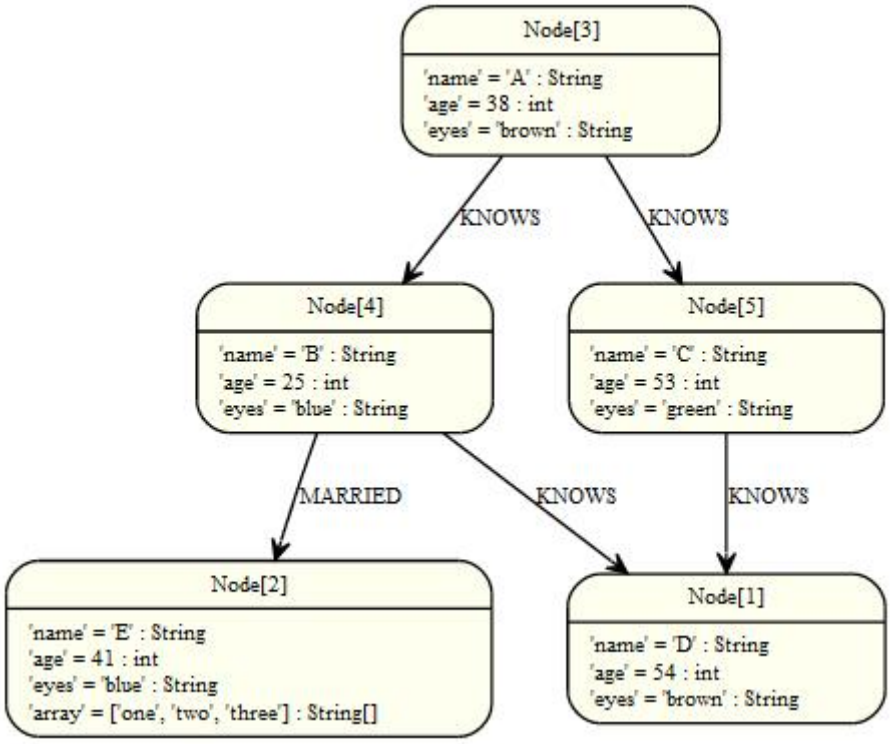
结果：

n
Node[3]{name->"A"}
Node[4]{name->"B"}
Node[5]{name->"C"}
3 rows, 0 ms

函数（Functions）

在 Cypher 中有一组函数，可分为三类不同类型：判断、标量函数和聚类函数。

图：



判断

判断为 boolean 函数，对给出的输入集合做判断并返回 true 或者 false。常用在 where 子句中过滤子集。

All

迭代测试集合中所有元素的判断。

语法：

All（标识符 in iterable where 判断）

参数：

Ø iterable ：一个集合属性，或者可迭代的元素，或一个迭代函数。

Ø 标识符：可用于判断比较的标识符。

Ø 判断：一个测试所有迭代器中元素的判断。

查询：

```
START a=node(3), b=node(1)
MATCH p=a-[*1..3]->b
WHERE all(x in nodes(p) WHERE x.age > 30)
RETURN p
```

过滤包含 age（30 的节点的路径，返回符合条件路径中所有节点。

结果：

p
(3)--[KNOWS,1]-->(5)--[KNOWS,3]-->(1)
1 rows, 2 ms

Any

语法：ANY(identifierin iterable WHERE predicate)

参数：

Ø Iterable（迭代器）：一个集合属性，或者可迭代的元素，或一个迭代函数。

Ø Identifier（标识符）：可用于判断比较的标识符。

Ø Predicate（判断）：一个测试所有迭代器中元素的判断。

查询：

```
START a=node(2)
WHERE any(x in a.array WHERE x = "one")
RETURN a
```

结果：

a
Node[2]{name->"E",age->41,eyes->"blue",array-> [Ljava.lang.String;@42101da9}
1 rows, 1 ms

None

在迭代器中没有元素判断将返回 true。

语法：NONE(identifierin iterable WHERE predicate)

Ø Iterable（迭代器）：一个集合属性，或者可迭代的元素，或一个迭代函数。

Ø Identifier（标识符）：可用于判断比较的标识符。

Ø Predicate（判断）：一个测试所有迭代器中元素的判断。

查询：

```
START n=node(3)
MATCH p=n-[*1..3]->b
WHERE NONE(x in nodes(p) WHERE x.age = 25)
RETURN p
```

结果：

p
(3)--[KNOWS,1]-->(5)
(3)--[KNOWS,1]-->(5)--[KNOWS,3]-->(1)
2 rows, 2 ms

Single

如果迭代器中仅有一个元素则返回 true。

语法：SINGLE(identifierin iterable WHERE predicate)

参数：

Ø Iterable（迭代器）：一个集合属性，或者可迭代的元素，或一个迭代函数。

Ø Identifier（标识符）：可用于判断比较的标识符。

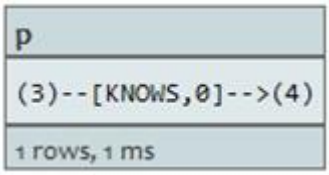
Ø Predicate（判断）：一个测试所有迭代器中元素的判断。

查询：

```
START n=node(3)
MATCH p=n-->b
WHERE SINGLE(var in nodes(p) WHERE var.eyes = "blue")
```

RETURN p

结果：



Scalar 函数

标量函数返回单个值。
Length
使用详细的 length 属性，返回或过滤路径的长度。

语法：LENGTH(iterable)

参数：

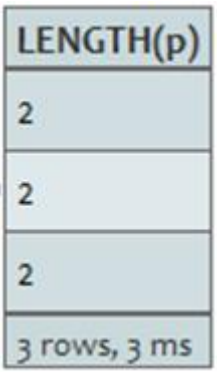
Ø Iterable（迭代器）：一个集合属性，或者可迭代的元素，或一个迭代函数。

查询：

START a=node(3)
MATCH p=a-->b-->c
RETURN length(p)

返回路径的长度。

结果：



Type

返回关系类型的字符串值。

语法：TYPE(relationship)

参数：

Ø Relationship：一条关系。

查询：

```
START n=node(3)
MATCH (n)-[r]->()
RETURN type(r)
```

返回关系 r 的类型。

结果：

TYPE(r)
"KNOWS"
"KNOWS"
2 rows, 1 ms

Id

返回关系或者节点的 id

语法：ID(property-container)

参数：

Ø Property-container：一个节点或者一条关系。

查询：

```
START a=node(3, 4, 5)
RETURN ID(a)
```

返回这三个节点的 id。

结果：

ID(a)
3
4
5
3 rows, 0 ms

Coalesce

返回表达式中第一个非空值。

语法：COALESCE(expression [, expression]*)

参数：

Ø Expression：可能返回 null 的表达式。

查询：

```
START a=node(3)
RETURN coalesce(a.hairColour?,a.eyes?)
```

结果：

COALESCE(a.hairColour,a.eyes)
"brown"
1 rows, 0 ms

Iterable 函数
迭代器函数返回一个事物的迭代器---在路径中的节点等等。

Nodes

返回一个路径中的所有节点。
语法：NODES(path)

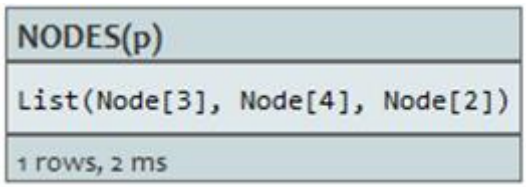
参数：

Ø Path：路径

查询：

START a=node(3), c=node(2)
MATCH p=a-->b-->c
RETURN NODES(p)

结果：



Relationships

返回一条路径中的所有关系。

语法：RELATIONSHIPS(path)

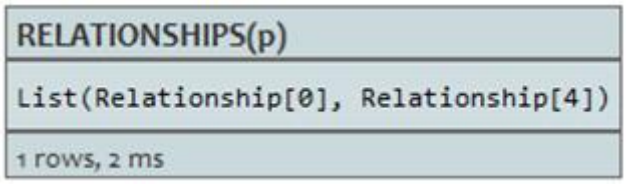
参数：

Ø Path：路径

查询：

START a=node(3), c=node(2)
MATCH p=a-->b-->c
RETURN RELATIONSHIPS(p)

结果：



Extract

可以使用 extract 单个属性，或从关系或节点集合迭代一个函数的值。将遍历迭代器中所有的节点并运行表达式返回结果。

语法：EXTRACT(identifier in iterable : expression)

Ø Iterable（迭代器）：一个集合属性，或者可迭代的元素，或一个迭代函数。

Ø Identifier（标识符）：闭包中表述内容的标识符，这决定哪个标识符将用到。

Ø expression（表达式）：这个表达式将对于迭代器中每个值运行一次，并生成一个结果迭代器。

查询：

```
START a=node(3), b=node(4),c=node(1)
MATCH p=a-->b-->c
RETURN extract(n in nodes(p) : n.age)
```

返回路径中所有 age 属性值。

结果：

extract(n in NODES(p) : n.age)
List(38, 25, 54)
1 rows, 2 ms