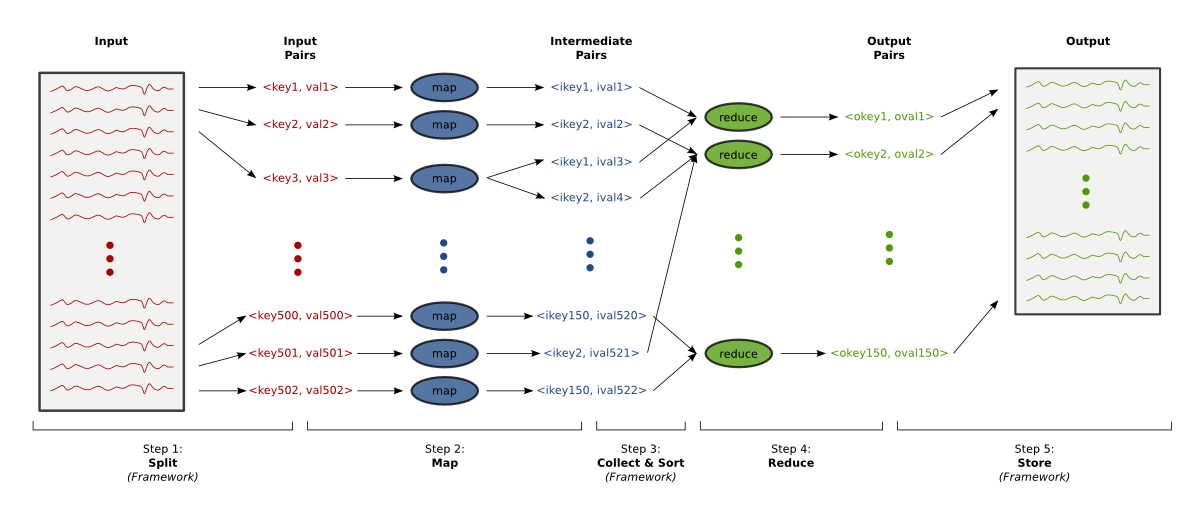
**《大数据管理》课程综合作业**

**1. 画出map-reduce计算模型的的程序框架示意图，并阐述其编程思想。**

Diagram

Description automatically generated

**图1** map-reduce程序框架示意图（讲义）

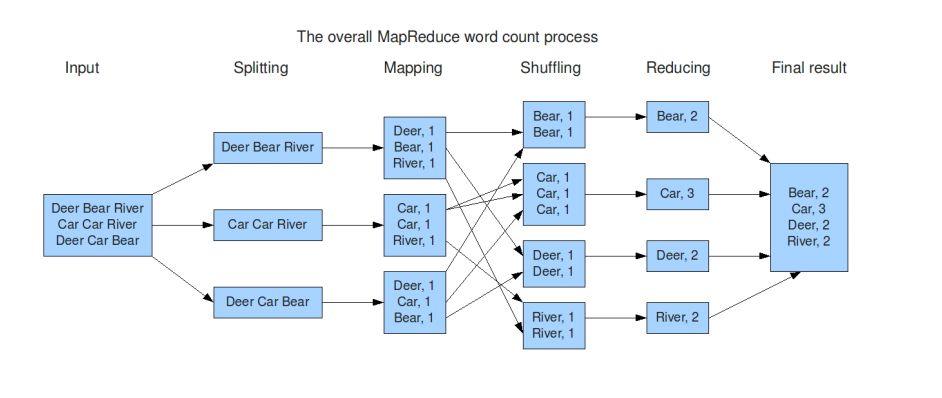


**图2** Map-Reduce编程框架示意图

**Map-Reduce 编程思路：**

1. 输入文件（input file）被拆分为键值对。 然后将这些键值对映射到Mapper（图2 的step 2），对应图1 的worker输入。
2. Mapper单独处理每个键值对并输出一个或多个中间键值对。
3. 中间文件中的键值对都按key来合并/收集（collect）、排序（sort）和分组（grouped by key），对应图2中的collect and sort。
4. 对于每个唯一的键，reducer 会接收键以及与之关联的所有值的列表。reducer 以某种方式聚合（aggregate）这些值（将它们相加、取平均值或找到最大值等）并输出一个或多个输出键值对。
5. 最后输出的对（pair）被收集并存储在输出文件中。

**例：word count**



数据分片阶段：上图中数据被分成3份，并分配给3个Mapper处理。

Map阶段：key，value pairs

Collect 阶段：Map阶段后，可以对每个mapper的中间结果做一个简单的合并。

Shuffle & partition阶段：排序，分区

Reduce阶段：汇总

**MapReduce特点**

* MapReduce更适合非结构化数据的ETL处理类操作，且其可扩展性及容错性好，但是单机处理效率较低。
* 在系统从传统项目转换为MapReduce项目时，所有的数据操作逻辑都必须转换为Mapper和Reducer函数。尽管MapReduce提供了简洁的编程接口及完善的处理机制，使得大规模并发处理海量数据成为可能，但从发展趋势看，相对复杂的任务转换为MapReduce任务的开发效率还是不够高，所以其有逐步被封装到下层的趋势，即在上层系统提供更为简洁方便的开发接口，在底层由系统自动转换为大量MapReduce任务。
* MapReduce本质上是由Map和Reduce序列两阶段构成。尽管Map和Reduce都支持大规模并发，但是由于在Map完成后有任务同步过程(例如 shuffle和partition)，因此只有所有Map任务执行完成后才能开始Reduce过程。MapReduce对子任务间复杂的交互和依赖关系缺乏表达能力。

**2. 结合教材第2.2.2.1节关于SQL语言对于JSON的扩展支持，简要阐述其基本思想，并对例2-4的代码进行注释说明。**

**基本思想：**OPENJSON将*key-value*分离出来，把JSON内容转换成以资料栏的方式呈现，相比用字符串存储获取数据更方便。

Text, letter

Description automatically generated

**步骤：**

将解析出来的JSON数据插入到region\_json里面，OPENJSON将JSON的值转化成WITH短语定义的数据类型。

第一行把所有的数据SELECT到region\_json里面，第二行表示用OPENJSON，最后WITH后面加的就是添加进region\_json的数据架构。

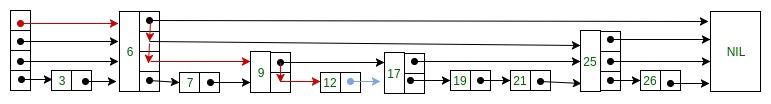
**3. 通过查阅资料，整理并写出Redis系统中使用的skip-list数据结构的特征，以及其数据插入、删除、查询操作的基本流程。**

**特征：**

1. skip-list是可以实现binary search的ordered-linked list；
2. 每个元素插入时随机生成它的level；
3. 最底层是一条含有所有element的linked- list
4. 如果一个元素出现在第x层，那么它肯定出现在x以下的level中,
5. 每个节点包含两个pointer，一个向下，一个向右,
6. Skip-list查询、插入、删除的时间复杂度为O(log n)，与balanced binary search tree接近；
7. 随机算法（random函数）对时间复杂度 和skip-list稳定度有影响
8. 维护所占空间相对而言很大

**Search operation：**

1. 下一个节点的key小于搜索的key，那么我们继续在同一level上前进。
2. 下一个节点的key大于要搜索的key，然后我们将指向当前节点 i 的指针存储在 update[i] 并向下移动一级并继续我们的搜索。
3. 在最低level (0)，如果最右边元素 (update[0]) 旁边的元素的 key 等于搜索的key，那么我们已经找到 key，否则失败。



以下为几个简单的例子：

**例：查找key=8**

Diagram, schematic

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram, schematic

Description automatically generated

Diagram

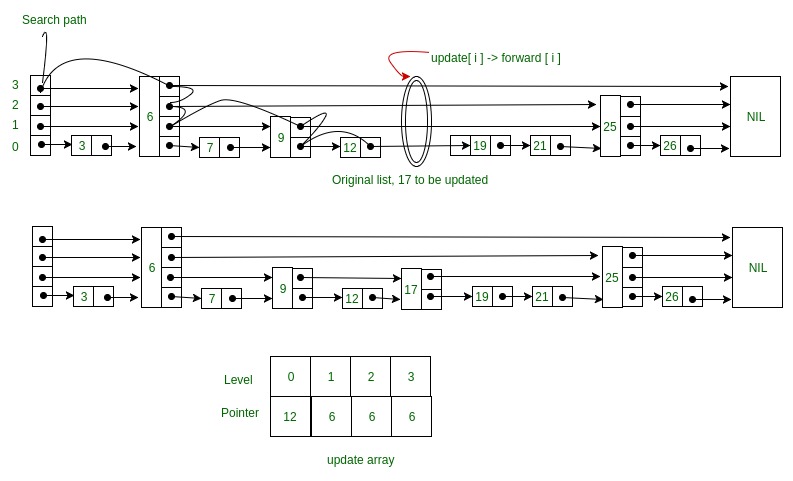
Description automatically generated

**Steps：**从L3开始找，key<13往下一层（L2），key>2，往右接着找，key在2～13之间，往下一层（L1），2～10之间，往下一层（L0），找到

如果key不能找到，查找方式同上。

**Insert operation：**

1. 下一个节点的key小于要插入的key，我们在同一level继续前进
2. 下一个节点的key大于要插入的key，我们将指向当前节点 i的pointer存储在 update[i] 并向下移动一个level并继续我们的搜索。



**例：insert key=9**

Diagram, schematic

Description automatically generated

Table

Description automatically generatedGraphical user interface, application

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

**Steps：**先做search，记录每条path起点（表格所示），新node高度随机（random），最后从上到下（L3～L0）把pointer更新即可。

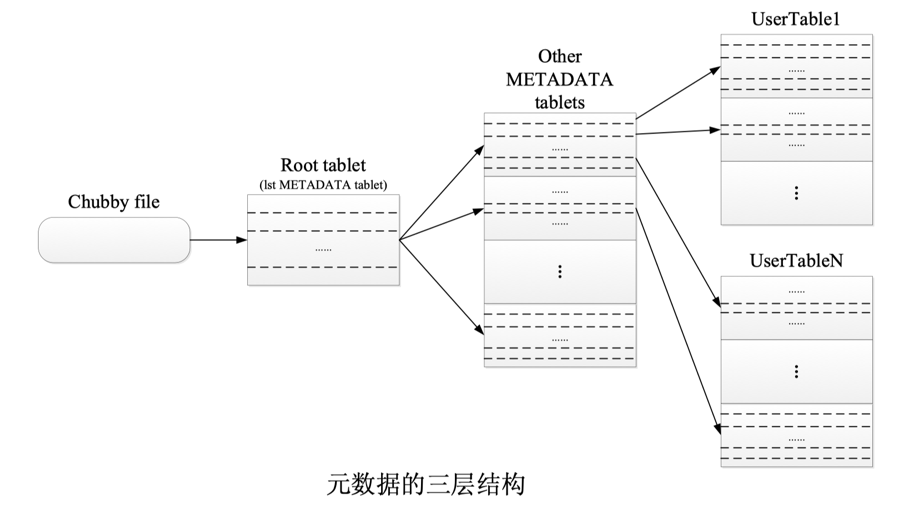
**Delete operation**

**Steps:**在skip-list中查找到这个元素的位置，如果未找到，则退出（还是先search）, 将该元素所在整列从表中删除, 最后将多余的“空链”删除（如下图）

A picture containing table

Description automatically generated

**4. 通过查阅资料，简要说明google的Bigtable系统的基于roottablet-metadata-usertablet的数据组织策略，给出相应的示意图，并结合该图简述其寻址机制。**



**数据组织策略以及操作：**

组成Bigtable主要分为三部分，链接可以到每个客户端的库，很多tablets还有一个主服务器也就是root table，这样的架构相对比较好的适应大量数据工作负载的变化，大表装小表，小表装小小表这样的一个结构。

a. 第一层（从左到右）

第一层是一个包含 Root Tablet 的位置信息、存储在 Chubby 中的文件。Root Tablet 包含了一个特殊的 METADATA 表里所有的 Tablet 的位置信息。METADATA 表的每个 Tablet 包含了一个用户 Tablet 的集合。

特性：Root Tablet 实际上是 METADATA 表的第一个 Tablet，由于 Root Tablet 永远不会被分割，保证了 Tablet 的位置信息存储结构不会超过三层。

b. 第二层

在 METADATA 表里面，每个 Tablet 的位置信息都存放在一个行关键字下面，而这个行关键字是由 Tablet 所在的表的标识符和 Tablet 的最后一行编码而成的。

特性：METADATA 的每一行都存储了大约 1Kb 的内存数据。在 128MB 的 METADATA Tablet 中，三层结构的存储模式共标识 234 个 Tablet 的地址。

c. 第三层：具体的 UserTable

说明：

I. 客户程序使用的库会缓存 Tablet 的位置信息。

II. 若无缓存或信息不正确，客户程序就在树状的存储结构中递归查询 Tablet 位置信息；

III. 如果客户端缓存是空，寻址算法需要通过三次网络来回通信寻址，其中包括一次 Chubby读操作；

IV. 如果客户端缓存的地址信息过期，寻址算法可能需要最多６次网络来回通信才能更新数据，因为只有在缓存中没有查到数据的时候才发现数据过期。

**5. 整理并简要归纳文档数据库MongoDB的数据查询的基本语法。**

**1.基本查询**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | db.集合名称.find({条件文档})   db.集合名称.findOne():   db.集合名称.find({条件文档}).pretty() |

**1） 比较运算符**

* 等于，直接用属性 + :
* 小于：$lt
* 小于等于:$lte
* 大于: $gt
* 大于等于: $gte
* 不等于: ne

|  |  |
| --- | --- |
| 1 | db.mask.find({count:{$gt:5}}) |

**2）逻辑运算符**

* and： 不需要写什么，逗号就是了
* or：使用$or
* and和or结合使用

|  |  |
| --- | --- |
| 1 2 3 | db.mask.find({age:{$gte:18}, gender:1}) db.mask.find({$or:[{age:{$gte:18},gender:1}]}) db.mask.find({$or:[{age:{$gte:18},gender:1}],name:'gj'}) |

**3）范围运算符**

* 使用$in,$nin判断是否在某个范围内

|  |  |
| --- | --- |
| 1 | db.mask.find({age:{$in:[18,28]}}) |

**4）支持正则表达式**

* 使用//或者$regex编写正则表达式

|  |  |
| --- | --- |
| 1 | db.mask.find({name:{$regex:'^黄'}}) |

**2. 自定义函数查询**

* 使用$where定义函数查询
* 函数跟JS函数定义是一样的

|  |  |
| --- | --- |
| 1 | db.mask.find($where:function(){return this.age>20}) |

**3.查询的扩展功能**

**1）Limit + Skip**

* limit：用于读取指定数量的文档，表示获取几条数据
* skip: 跳过指定数量的文档，从第几条开始看

|  |  |
| --- | --- |
| 1 2 | db.mask.find().limit(2)  db.mask.find().skip(2) |

**2）投影**

* 在查询返回的结果中，只选择必要的字段，而不是选择一个文档的整个字段—-投影
* 在find的第二个参数中去写

|  |  |
| --- | --- |
| 1 2 3 | db.集合名称.find({},{字段名称:1,...})   db.mask.find({},{\_id:0, name:1,gender:1}) |

**3）排序**

* sort方法

|  |  |
| --- | --- |
| 1 | db.mask.find().sort({age:-1}) |

**4）统计个数**

* count()方法

|  |  |
| --- | --- |
| 1 2 3 | db.集合名称.count({条件})  db.mask.count({age:{gt:20}}) |

**5）消除重复**

* distinct() 去重复

|  |  |
| --- | --- |
| 1 2 3 | db.集合名称.distinct('去重字段'，{条件})  db.mask.distinct(gender,{age:{$gt:20}}) |

**4.MongoDB的高级操作**

主要讲解这个数据库的高级操作，包括聚合、主从复制、分片、备份和恢复、MR、与python的交互

**4.1. 聚合aggregate**

* 用于计算数据，类型sum()和avg（）

|  |  |
| --- | --- |
| 1 | db.集合名称.aggregate([{管道:{表达式}}]) |

1. **管道的含义：** 将前一次操作的输出结果作为下一个命令的输入

* 常用管道有：
  + $group:将集合中的文档分组，用于统计结果
  + $match:过滤数据，只输出符合条件的文档
  + $project:修改输入文档的结果
  + $sort:将输入文档排序后输出
  + $limit:限制聚合管道返回的文档数
  + $skip:跳过指定数量的文档
  + $unwind:将数组类型的字段进行拆分

1. **表达式：** 处理输入文档并输出

* 常用表达式为：
  + $sum: 计算总数
  + $avg: 计算平均值
  + $min: 计算最小值
  + $max: 计算最大值
  + $push:在结果文档中插入值到另外一个数组中
  + $first:根据资源文档的排序获取第一个文档数据
  + $last:根据资源文档的排序获取最后一个文档数据

**4.1.1. $group**

* 将集合中的文档分组，用于统计结果
* \_id表示分组的依据，使用某个字段的格式为’$字段’

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | db.mask.aggregate([  {$group:{\_id:'$gender',counter:{$sum:1}}}  {$group:{\_id:'$gender',counter:{$avg:'age'}}}   {$group:{\_id:'$gender',counter:{$frist:'age'}}}     {$group:{\_id:'$gender',counter:{$push:'age'}}}   {$group:{\_id:'$gender',counter:{$push:'$$ROOT'}}}   ]) |

**Group by null** 将这个集合数据分成一组

|  |  |
| --- | --- |
| 1 2 3 | db.mask.aggregate(  {$group:{\_id:null,counter:{$sum:1}}} ) |

**4.1.2. $match**

* 用于过滤数据

**4.1.3.$project**

* 将结果集中的一部分拿出来显示

**4.1.4. $sort**

* 用于排序

**4.1.5. $limit+skip**

**4.1.6. $unwind**

* 将文档中的某个数组类型字段拆分成多条，每条包含数组中的一个值

|  |  |
| --- | --- |
| 1 | db.集合名称.aggregate([{$unwind:'$字段名称'}]) |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | db.mask.aggregate([]    {$match:{age:{$gt:20}}},     {$group:{\_id:'$gender',counter:{$sum:1}}},    {$project:{\_id:0,name:1,age:1}},    {$sort:{\_id:1}},    {$skip:1},  {$limit:1}, ]) |

**4.2. 索引**

* 在大量数据中快速查找

**步骤一：创造大量数据**

|  |  |
| --- | --- |
| 1 2 3 | for(i=0;i<100000,i++){  db.mask.insert(name:'test' + i, age:i) } |

**步骤二：数据查找性能分析**

* 查找姓名为test10000的文档,并通过explain()来进行性能分析

|  |  |
| --- | --- |
| 1 | db.mask.find({name:'test10000'}).explain('executionStats') |

**步骤三：创建索引**

|  |  |
| --- | --- |
| 1 2 3 4 | db.集合.ensureIndex({属性:1})   db.mask.ensureIndex({"name":1}) db.mask.find({name:'test10000'}).explain('executionStats') |

**索引命令**

* 建立单一索引

|  |  |
| --- | --- |
| 1 | db.mask.ensureIndex({"name":1},{"unique": true}) |

* 建立多个索引

|  |  |
| --- | --- |
| 1 | db.mask.ensureIndex({"name":1，"age":1}) |

* 查看所有索引

|  |  |
| --- | --- |
| 1 2 3 | db.mask.getIndexes()  db.mask.dropIndexes(‘索引名称’) |

**6. 整理并简要归纳图数据库Neo4j数据查询Cypher的基本查询语法。**

**基本语法**

（）表示节点

(n) 用来给节点赋予一个名字，可以在需要时引用这个节点  
(n:label) 用来匹配有某个标签的节点  
(n: label:label1:label2) 用来匹配有多个标签的节点  
(n: label:label1{key:value}) 用对象字面量的方式，通过节点的属性对节点进行筛选

**关系语法**

用方括号 [] 来表示关系

关系与节点一样，可以赋予一个名字以便在别处引用  
[ r ] 表示一个关系被赋予变量 r ，可以在别处引用  
[ r:label ] 匹配有某个标签的关系

注：关系只能有一个标签，节点可以有多个

[ r:label1 | :label2 ] 用来匹配多个关系中的一种  
[ n:label1 | :label2 {key : value }] 用对象字面量的方式，通过关系的属性对关系进行筛选  
() - [\* 3..5] ->() 用来表示变长关系, 即两个节点之间通过3-5个关系连接  
() - [\*3..] -> () 关系长度大于3  
() - [\*..5] -> () 关系长度小于5  
() - [\*] -> () 任意边界长度

注：变长关系不能用于create和merge语句中

**关系与节点的连接（单个横杠）**

- 表示双向连接  
-> 或者 <- 表示单向连接

例如：

()-[]-()

()-[]->()

()<-[]-()

**节点与节点直接连接（两个横杠）**

如果不在意节点之间通过何种关系连接，可以用 --、-->、<-- 来直接连接节点与节点

()--()-->()

()-->()<--()

**节点的 id**

节点的 id 是整数类型，由数据库内部自行维护，在查询语句中可以使用 id（n）函数来获取。

**语句可以分为3类**

读语句：match, optional match, where, start, aggregation, load csv

写语句：create, merge, set, delete, remove, foreach, create unique

通用语句：return, order by, limit, skip, with, unwind, union, call

**以下为讲义上的例子：**

例：查找并返回所有顶点

MATCH(n)

RETURN n

例：查找带有某一类标签的顶点，并返回该类型顶点的投影信息

//查找带有movie标签的顶点，返回这些顶点的名称

MATCH(n:movie)

RETURN n.名称

WHERE子句

例：查找片名为喜剧之王的电影的上映年份

MATCH(n:movie) //节点名:标签{属性名:属性值}

WHERE n.名称=‘喜剧之王’

RETURN n.上映年份

正则表达式：可以通过使用=~ /regexp/来匹配正则表达式。

例：查询名字以‘Tom’字符串开头的演员

MATCH(n:actor)

WHERE n.name =~ ‘Tom.\*’

RETURN n

**转义正则表达式**：正则表达式中需要字符本身，通过转义“\”转移实现

例：在正则表达式中需要有斜杠’/’

MATCH(n:actor)

WHERE n.name =~ ‘Tom\/.\*’

RETURN n

**不区分大小写的正则表达式**：在正则表达式前加上?i，整个正则表达式将会忽略大小写。

例：MATCH(n:actor)

WHERE n.name =~ ‘(?i)tom.\*’

RETURN n //Tom、Tomas…

关系类型上的过滤：可以先通过match模匹配相应的关系，再使用明确的type属性对关系类型名作一个正则比较。

例：查询以node(3)顶点为起点的边，且边的类型名以K开头

MATCH (n) WHERE id(n) = 3

MATCH (n)-[r]->( )

WHERE type(r) =~‘K.\*’

RETURN r

图匹配查询的基本思路是通过MATCH（可多个）、WHERE、RETURN的组合语句完成，MATCH从属性图中获取有关点、边或属性的数据，WHERE子句过滤MATCH查询的结果，RETURN返回符合匹配条件数据的信息。

例：找出所有自导自演过电影的影星以及对应电影的名称。

MATCH (n1)-[:导演]->(n2:电影)

MATCH (n1)-[:出演]->(n2:电影)

RETURN n1.姓名, n2.名称

Cypher可以实现使用查询路径P的图导航查询，包括使用反向边表达。

例：使用查询路径例：，查询曾经出演过同一部电影的合作影星，条件的正则表达式中包含一条出演的反向边。

MATCH (n1)-[:出演]->(:电影)<-[:出演]-(n2)

RETURN distinct n1.姓名，n2.姓名

例：通过和“吴孟达”共同出演过电影的演员为**桥梁**，找出和这些演员有过合作的其他演员。

导航路径思路：actor-co\_actors-co\_co\_actors

MATCH (t:Person {姓名：’吴孟达’})-[:出演]->(m1)<-[:出演]-(coActors),

(coActors)-[:出演]->(m2)<-[:出演]-(co\_co\_Actors)

//排除演员t的直接共同演出演员以及t本人

WHERE NOT (t)-[:出演]->()<-[:出演]-(co\_co\_Actors)

AND t<>cocoActors

RETURN co\_co\_Actors.姓名

例：列出一组用户（5、2、3）的id，遍历图查找这些用户指向的friend，并筛选保留name是以S开头的friends，返回用户、friend的name属性。

MATCH (user) WHERE id(user) IN [5,2,3]

MATCH user-[:friend]->follower

WHERE follower.name =~‘S.\*’

RETURN user, follower.name

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

例：判断定点3到2、4之间是否存在边标签为KNOWS且长度不超过3的路径，如果存在则返回相应路径的起点和重点。

MATCH (a) WHERE id(a)=3

MATCH (x) WHERE id(x) IN [2,4]

MATCH a-[:KNOWS\*1..3]->x

RETURN a, x

**任意的关系**：使用\*

MATCH p=(n)-[\*]->(m) RETURN p

n到m的任意关系，任意深度节点

**可选关系**

如果某个关系是可选的，可使用OPTINAL MATCH，类似SQL中outer join的工作方式。如果关系存在就返回，否则在相应的地方返回null。

例：返回电影The Matrix这个节点的外向关系。

MATCH (a:Movie { title: 'The Matrix' })

OPTIONAL MATCH (a)-->(x)

RETURN x

例：MATCH (a) where id(a) = 3

MATCH (a)-[\*0..]->() return a;

**可选关系类型**

可在查询中指定可选的关系类型。

MATCH (a:Movie { title: 'The Matrix' })

OPTIONAL MATCH (a)<-[r: ACTED\_IN]-()

RETURN r

**最短路径**： shortestPath函数可以找出一条两个节点间的最短路径

MATCH (s) WHERE id(s) = 1

MATCH (e) WHERE id(e) = 2 //4.0版本放弃START

MATCH p = shortestPath( (s)-[\*..15]->(e))

RETURN p

所有**最短路径**： allShortestPaths，找出两顶点间所有的最短路径。

例： MATCH (s) WHERE id(s) = 1

MATCH (e) WHERE id(e) = 2

MATCH p = allShortestPaths( s-[\*..15]->e )

RETURN p

**图匹配与图导航的复合查询**

例：与冯小刚有过合作演出，且与吴孟达也有过合作演出的演员

MATCH (n1:人 {姓名：’冯小刚’})-[:出演]->(电影)<-[:出演]-(n2),

(n3:人 {姓名：’吴孟达’})-[:出演]->(电影)<-[:出演]-(n2),

RETURN n2.姓名

例：查询与周星驰有关联的且在四跳关联之内的所有电影和演员

MATCH (n:人{名字:’周星驰’})-[\*1..4]-(n1)

RETURN DISTINCT n1

例：查询顶点node（2）的相关顶点数

MATCH (n) WHERE id(n) = 2

MATCH (n)-->(x)

RETURN n, count(\*)

例：按照关系类型分组，返回关系类型及其出现数量

MATCH (n) WHERE id(n) = 2

MATCH (n)-[r]->()

RETURN type(r), count(\*)

例：计算实体数——连接到顶点node（2）上的顶点个数

MATCH (n) WHERE id(n) = 2

MATCH (n)-->(x)

RETURN count(x)

例：计算非空可选值数

MATCH (n) WHERE id(n) IN [2,3,4,1]

RETURN count(n.property)

//或者RETURN count(distinct n.property)

例：**求和/平均/最大/最小**

Sum集合简单计算数值类型的值（Null值将自动去掉）

MATCH (n) WHERE id(n) IN [2,3,4]

RETURN sum(n.property)

类似的：RETURN avg(n.property)、RETURN max(n.property)、RETURN min(n.property)

例：MATCH (n) WHERE id(n) IN [2,3,4]

RETURN avg(n.property) ，max(n.property)，min(n.property)

例：**聚类**（COLLECT），Collect将所有值收集到一个集合list中。

MATCH (n) WHERE id(n) IN [2,3,4]

RETURN collect(n.property)

返回一个带有所有属性值的简单列表。

***With*从句**：with从句可以连接多个查询的结果，将上一个查询的结果用作下一个查询的开始。可以有两种用法：  
（1）对聚合值过滤；

（2）通过使用order by 和limit，with可以限制传入下一个match子查询语句的实体数目。

例：以with o,count(r) as count\_r为结果集，在此基础上筛选出count\_r大于8的

MATCH (n:Person)-[r:ACTED\_IN]->(o:Movie)

with o, count(r) as count\_r

where count\_r > 8

RETURN o

例：对聚集结果保留计数值大于1的

MATCH (david { name: "David" })--(otherPerson)-->()

WITH otherPerson, count(\*) AS foaf

WHERE foaf > 1

RETURN otherPerson

例：在 collect 前排序,  空值将被忽略

match (n)  
  with n order by n.name desc limit 3  
 return collect(co)

例：**在路径搜索的时候限制分支数**。从”Anders”出发，找到关联（无向）的所有人的集合m，保留集合中按照姓名降序排列的第一个顶点，返回该顶点关联（无向）的所有人的name。

MATCH (n { name: "Anders" })--(m)

WITH m

ORDER BY m.name DESC LIMIT 1

MATCH (m)--(o)

RETURN o.name

COLLECT(distinct n.name)

**7. 结合适当的原理图，简述图计算模型GAS的原理及其查询过程。**

GraphLab将数据抽象成Graph结构，将基于顶点切分的算法的执行过程抽象成Gather、Apply、Scatter三个步骤。如下：需要完成对V0邻接顶点的求和计算，将顶点V0进行切分，边关系以及邻接点部署在两台处理器上，各台机器上并行进行求和运算，然后通过master（蓝色）顶点和mirror（橘红色）顶点的通信完成最终的计算。

A picture containing text, watch

Description automatically generated

1.Gather阶段，工作顶点的边从连接顶点和自身收集数据。

2.Apply阶段，mirror将gather阶段计算的结果发送给master顶点，master进行汇总并结合上一步的顶点数据，进行进一步的计算，然后更新master的顶点数据，并同步给mirror。

3.Scatter阶段，工作顶点更新完成之后，更新边上的数据，并通知对其有依赖的邻结顶点更新状态。

由于gather/scatter函数是以单条边为操作粒度，所以对于一个顶点的众多邻边，可以分别由相应的节点独立调用gather/scatter函数，从而避免Pregel模型的问题。