目录

[1. 研究意义 2](#_Toc9526649)

[2. 批量RDF Q/A框架 2](#_Toc9526650)

[3. Nc的含义与获取方法 3](#_Toc9526651)

[3.1. 批量N分组 4](#_Toc9526652)

[3.2. 判断语义相似度 4](#_Toc9526653)

[3.3. 在N之间找出相似的语义段 4](#_Toc9526654)

[3.3.1. 穷举方法寻找 4](#_Toc9526655)

[3.3.2. 启发式寻找方法 5](#_Toc9526656)

[3.4. 边界处理 5](#_Toc9526657)

[3.4.1. 红色的部分只有一个单词 5](#_Toc9526658)

[3.4.2. 红色的部分是一个短句 6](#_Toc9526659)

[4. 由Nc生成Qc 7](#_Toc9526660)

[4.1. gAnswer（18TKDE） 7](#_Toc9526661)

[4.2. 语义对齐（Phrase Linking） 8](#_Toc9526662)

[4.2.1. 语义向“用户SPARQL查询日志”对齐 9](#_Toc9526663)

[4.2.2. 互信息语义消歧 10](#_Toc9526664)

[4.3. 隐语义关系以及语义不完全覆盖问题的解决措施 10](#_Toc9526665)

[4.3.1. NLP中判断句子是否有意义 11](#_Toc9526666)

[4.3.2. 隐语义关系挖掘 12](#_Toc9526667)

[4.3.3. 不完整语义推测 13](#_Toc9526668)

[4.4. 条件概率计算 14](#_Toc9526669)

[4.4.1. 语句语义条件概率计算 14](#_Toc9526670)

[4.4.2. SPARQL语义条件概率计算 15](#_Toc9526671)

[4.4.3. 离线阶段任务 15](#_Toc9526672)

[4.4.4. 条件概率优化 15](#_Toc9526673)

[5. SPARQL中triple pattern的核心度度量 16](#_Toc9526674)

[6. RDF Engine 19](#_Toc9526675)

[6.1. RDF存储方式 19](#_Toc9526676)

[6.2. 查询优化 20](#_Toc9526677)

[6.2.1. 引擎处理SQL和SPARQL的流程 20](#_Toc9526678)

[6.2.2. 动态规划算法寻找连接顺序 22](#_Toc9526679)

[6.2.3. 查询代价估计 23](#_Toc9526680)

[6.2.4. Join Method 24](#_Toc9526681)

[7. Extended Work (Parallel Execute Version) 25](#_Toc9526682)

[7.1. 查询代价(时间) 25](#_Toc9526683)

[7.2. PCM中句法分析树分组 26](#_Toc9526684)

[7.3. 组查询执行顺序问题 27](#_Toc9526685)

# 研究意义

Knowledge Graph(KG) has become a very popular way to represent and query world knowledge. RDF and SPARQL are two core concepts in the Knowledge Graph, which are standards for representing and querying Knowledge Graph, recommended by the W3C. Although SPARQL is a standard way to access RDF data, it remains tedious and difficult for users. Hence, qustion/answering(Q/A) based on Knowledge Graph has received wide attention in both natural language processing and database areas.

Generally, there are two significant challenges in RDF Q/A systems（***question understanding, Question -> SPARQL***) and ***Query evaluation***. A great deal of research has been done to address the first challenge. For example, Zou et al proposed two frameworks to build the semantic query graph. To overcome the second challenge, single-machine RDF systems, like RDF-3X, gStore and many distributed SPARQL query engines have been introduced.

However, in some scenarios multiple queries can be processed as a batch. For example, in Q/A system, multiple questions often need to be processed together. Therefore, the third challenge arises in RDF Q/A system: "How can RDF Q/A System handle batch questions efficiently". Some research work has been proposed to efficiently process multiple SPARQL queries. However, efficient processing of multiple SPARQL queries is not equivalent to multiple Question Answer. Because batch processing of SPARQL queries is only part of the second challenge.

**Contribution:**

1. 提出了批量RDF Q/A处理框架，并给出了寻找问题之间公共语义的方法。
2. 将隐马尔可夫模型引入RDF Q/A系统来寻找隐式关系和解决SPARQL语义不完整的问题，以此来提高推荐精度。
3. 使用互信息消歧并提出将phrases Linking由向RDF Graph对齐改为向User Query Log对齐，既能提高精准度又能提高效率。
4. 定义了SPARQL中triple pattern的核心度度量公式，并使用核心度大小参与决定SPARQL引擎执行中的joning order问题。
5. 设计了一种新的单机版高效SPARQL引擎，并在此基础上给出了并行环境下设计批量RDF Q/A系统存在的问题与解决方法。

# 批量RDF Q/A框架

Firstly, We define the following three formulas.



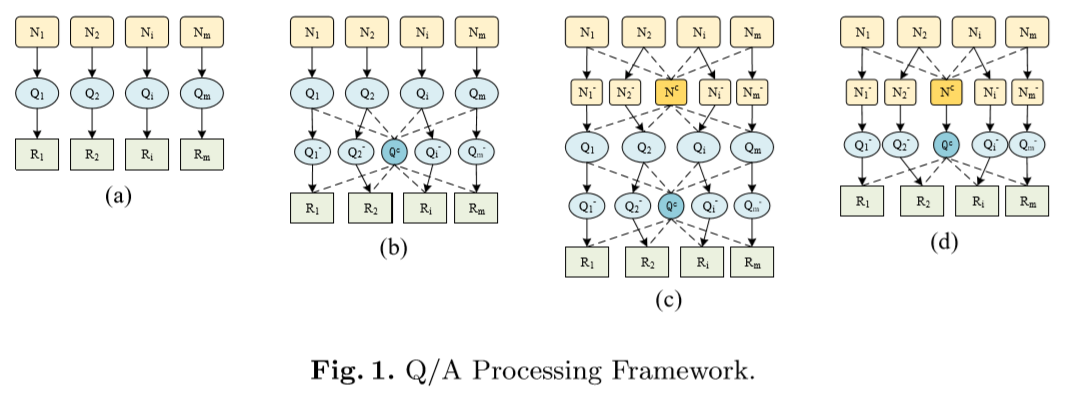
Formula 1 refers to the method of converting signle natural language question ***N*** to a SPARQL query ***Q***, Formula 2 refers to the method of fast matching Q on RDF data graph ***G*** by various query engines, and Formula 3 is used to represent all multiple SPARQL query optimization methods, Which ***Qs*** represent SPARQL Query Set。

As shown in Fig.1 (a), the current RDF Q/A system can only process ***Ni*** in ***NS*** by executing Formula 1 and Formula 2 one by one. Fig. 1(b) shows a better way to execute ***QS***. Since Formula 3 already exists Multi-Query Optimization for SPARQL queries, we can first execute Formula 1 for all ***Ni*** in ***NS***to obtain a ***QS***, formula 3 is then executed for this ***QS***. This approach makes good use of the common subgraph that may exist in Formula 2. However, there may also be a common structure in Formula 1.

Motivated by the above, It's easy to think of adding a Formula 4 that can convert ***NS*** to ***QS*** in batches. Fig.1(c) shows the method, which can make better use of the common structure in both Formula 1 and 2, so it can improve the efficiency. However, there is a ***synchronous action*** in this method. That is, "Formula 3 will not be implemented until all Ni in NS are converted into ***Qi***.



In order to improve the efficiency as much as possible, we give the method in Fig.1 (d).



模型d就是我们最终选定的批量RDF Q/A框架。

|  |  |
| --- | --- |
| Ni | a natual language question |
| Nc | 自然语言问题的公共部分 |
| Ni- | Ni-Nc |
| Qi | 由N转化的SPARQL查询 |
| Qc | SPARQL查询的公共部分 |
| Qi- | Qi-Qc |
| Ri | Qi查询执行返回的结果 |

# Nc的含义与获取方法

Nc指的是公共语义。要想找到两句话中的公共语义，有两个挑战要解决。

**其一**：判断两句话是否是语义相似的。

**其二**：如何在两个Question中找到公共语义部分。

为方便介绍，我们使用N来代表natual language question，用wi代表组成N的单词，使用Q代表由N生成的SPARQL查询。

## 批量N分组

当RDF Q/A system要处理一批自然语句N的时候，为了高效的寻找和最大程度的利用公共结构。第一步要做的就是要将这一批N进行分组，在这里我们使用论文《Multi-Query Optimization for Subgraph Isomorphism Search》中提出的tri-vertix方法对N进行分组。

## 判断语义相似度

目前NLP中判断两句话之间是否相似的方法有很多，我们选择使用google提出的doc2vec模型来判断N之间的相似性。

Doc2vec是word2vec的进阶版本。能很好的判断两句话或者两个文本块之间的相似度。

## 在N之间找出相似的语义段

例如寻找以下两句话之间的公共语义部分：

|  |  |
| --- | --- |
| who is obama’s father? | who is Obama’s dad? |
| **核心想法：**将两个问题N1，N2分解为一系列的子问题的集合N1S和N2S。之后在这两个集合中寻找是否有子问题语义相似。 | |

### 穷举方法寻找

|  |  |  |  |
| --- | --- | --- | --- |
|  | N1 | N2 | 相似度 |
| 1 | who is obama’s father? | who is Obama’s dad? | 0.99 |
| 2 | who is obama’s | who is Obama’s | 1 |
| 3 | is obama’s father? | is Obama’s dad? | 0.99 |
| 4 | who is | who is | 1 |
| 5 | is obama’s | is Obama’s | 1 |
| 6 | obama’s father? | Obama’s dad? | 0.98 |

可以看到任何一个包含m个单词的问题N都可以被分为一系列的子问题，子问题数量由如下公式定义：

事实上N1产生的所有子问题与N2产生的所有子问题之间都要进行一次比对。时间复杂度为O(n2)。

### 启发式寻找方法

通常来说，一个N中wi的数量比较少，因此上面的穷举方法已经足够了。但是仍然不能避免一些用户会输入比较长的N。因此上面的穷举方法就不够高效了，由此我们给出如下启发式方法。我们使用一个例子来解释，

**例子：**

我是一个研究生，我的名字是张三，我就读于天津大学。

我是一个研究生，我的名字是李四，我的性别是男，我就读于天津大学。

**第一步：挑选出可能成为SPARQL中实体（主语、谓语）或者谓词的词语。**

我 研究生，我 名字 张三，我 就读于 天津大学。

我 研究生，我 名字 李四，我 性别 男，我 就读于 天津大学。

**第二步：挑选出两个N之间所有互相相似的单词。(word2vec)**

我 研究生，我 名字 张三，我 就读于 天津大学。 N1

我 研究生，我 名字 李四，我 性别 男，我 就读于 天津大学。 N2

其中绿色单词部分代表在另一句话中存在相似单词，而红色字体则代表没有相似语义单词。为了保证相似语义部分能生成相同的SPARQL查询，那么公共语义部分Nc中就不能存在有不相同语义的单词。

**第三步：具体的算法**

找出较短句子N1中所有连续的相似单词（绿色单词连接在一起的部分）。也就是将红色部分作为分隔符，进而将一句话进行分割。

## 边界处理

主要处理的是红色与绿色单词相邻的区域。

### 红色的部分只有一个单词

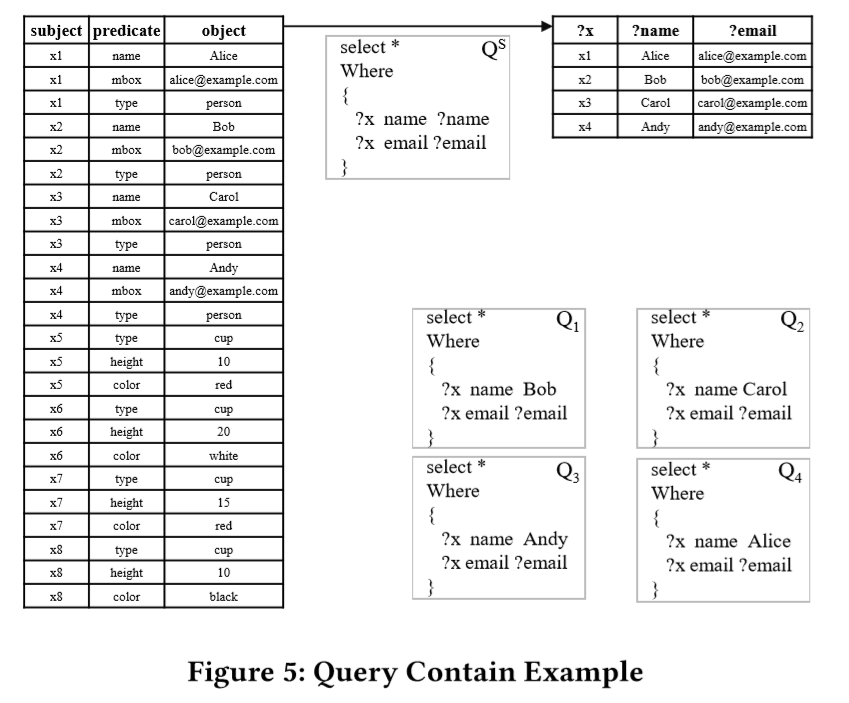
这个红色单词的前后必然存在一部分绿色内容。（从句法解析树上看）如果红色单词就近的绿色内容中包含一个可以做谓语的单词，那么这个红色单词将来很可能会是一个triple pattern的主语或者宾语。此时这个单词会被保存起来作为以后的一个筛选条件。

例如将N1,N2中的两部分子问题（我 研究生，我 名字 张三；我 研究生，我 名字 李四）转化为SPARQL查询：

|  |  |
| --- | --- |
| {  ?x 学位 研究生  ?x name 张三  } | {  ?x 学位 研究生  ?x name 李四  } |

我们可以看到以上这两个SPARQL查询中唯一不同的部分只有第二个triple pattern中的宾语。我们将这类问题统一定为Query Contain问题。

以下使用一个实例来介绍：



假设我们有Q1,Q2,Q3,Q4四个查询。那么正常情况下查询引擎需要执行四次才能获得这四个查询的结果。

但是我们看到有Qs这个查询（Q1,Q2,Q3,Q4四个查询的结果都包含于Qs查询的结果中，这也是命名为Query Contain问题的原因），如果我们首先执行Qs。之后在?name一列中选择包含Bob, Carol, Andy与Alice的行就可以获得Q1,Q2,Q3,Q4四个查询的结果了。这样做可以大大的节省查询执行时间。

我们可以看到，如果红色部分仅仅是一个单词，并且这个单词附近有一个可以做谓词的单词。这种情况下，将子问题进行转化就会产生例子中的这种Query

Contain问题。

**所以我们将这个具体不同的红色部分设置为一个变量，并将原本具有的常量提取出来作为筛选条件。**

如果就近的绿色内容是一个可以作为主语或者宾语的单词，那就说明存在隐式关系需要挖掘。因此这个距离红色单词最近的绿色单词被取出来不再作为生成公共结构的一部分。

### 红色的部分是一个短句

就当做一个子问题处理。将其生成为triple pattern。后面再进一步处理。

# 由Nc生成Qc

## gAnswer（18TKDE）



**定义：**

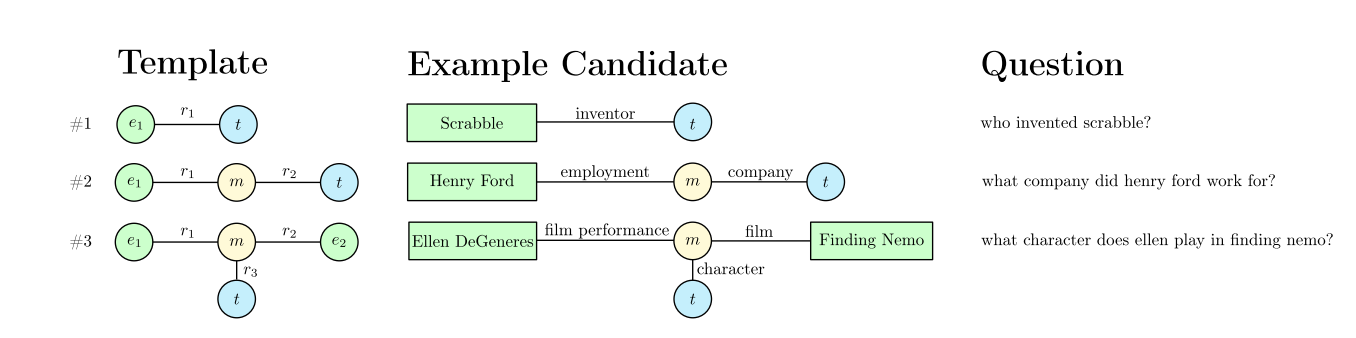
***Semantic Query Graph:***A semantic query graph is a graph, in which each vertex vi is associated with an entity phrase, class phrase or wild-cards in the question sentence N; and each edge vivj is associated with a relation phrase in the question sentence N。

***NodeFirst框架：***gAnswer中提出的构造Semantic Query Graph的一种方法，他首先将句子N中所有可以做subject和object的单词挑选出来，之后再从句子中寻找存在于这些单词之间的可以做predicate的短语或者词语。

***RelationFirst框架***：也是一种构造Semantic Query Graph的方法，他首先将句子N中所有可以做predicate的单词挑选出来，再在predicate的两侧寻找可以做subject和object的词语。

**整体执行流程：**

1. 句子进行简单的分词并标记词性，并根据句子中单词的词性选出可能是主语、谓语、宾语的单词。
2. 根据NodeFirst框架或者Relation First框架（其他论文中有template based的方法）根据单词之间的依赖关系（句法依赖树给出）形成Semantic Query Graph。（Semantic Query graph就相当于template based 方法中的Template）
3. 将Semantic Query Graph中的结点，边对应到具体数据集中的具体表示。比如说句子中说的是”Transformers”而数据集中真正存储的可能是“dbo:transformers”。并且还可能存在这种问题，数据集中可能包含大量与”Transformers”相似的表示。我们还要挑选出其中最可能的对应词语。
4. 之后就形成了真正可用的Query graph



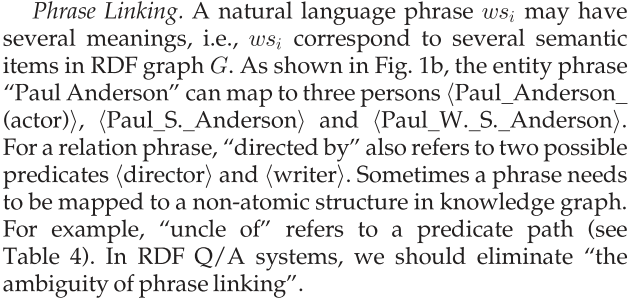
Template based method

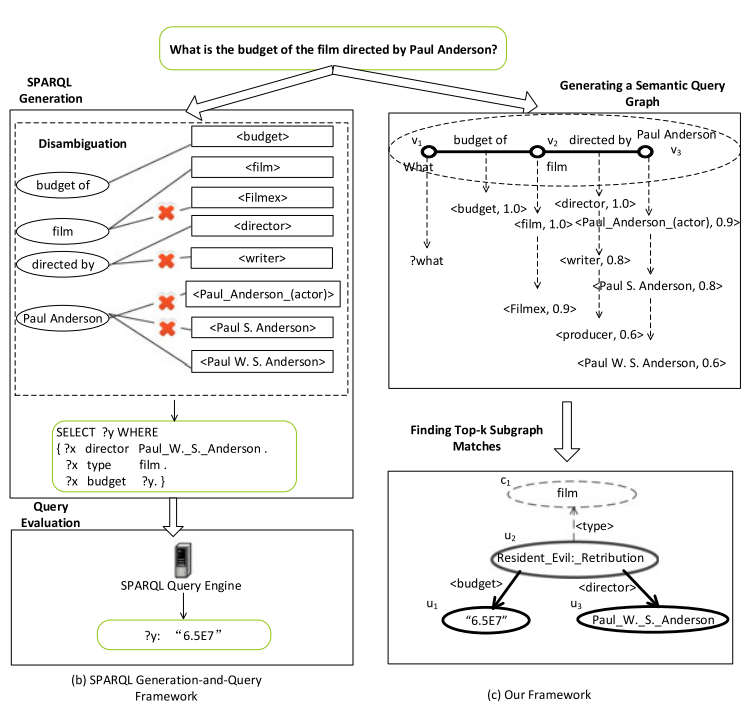
实质上在Query Understanding中，又有两个挑战：**其一是**，生成一个类似于Template的框架（也就是Semantic Query Graph）；**其二是，**正确的建立N中单词与数据集中表示的映射。

其中gAnswer这篇论文很好的解决了Query Understanding中的第一个挑战。即，不再需要事先建立好模板。能够根据句子自动生成一个结构。在我们的论文中也使用gAnswer中提出的方法来解决生成Template这个挑战。

## 语义对齐（Phrase Linking）

**语义对齐**：指的是将自然语言问句中的单词或短语在数据集中找到正确的表示形式。





### 语义向“用户SPARQL查询日志”对齐

目前所有的方法在做语义对齐的时候，都是选择将出现在句子中的短语wi向数据集进行对齐。这样做并没有什么不对，但是还可以更好。

我们知道，知识图谱G本身只是一堆数据的集合。不能反映当前这段时间，或者不同用户之间所关注点有何不同。而且数据集通常来说非常大，从这个数据集中选取N中wi的对应表示是比较耗费时间的。

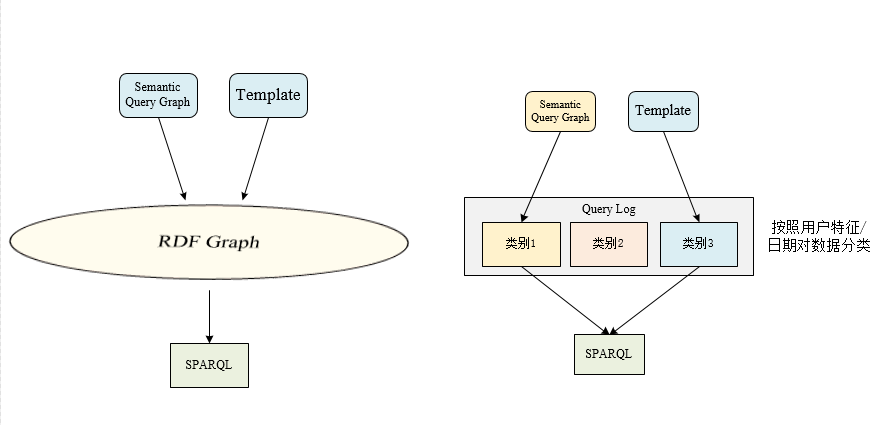
但是针对这个数据集的大量SPARQL查询就不同了。假设我们采用的是DBpedia数据集作为知识库，并且有一个含有非常多SPARQL查询的用户查询日志Ql。

以往在做语义对齐的时候都是将wi向Dbpedia对齐。但是我们选择向Ql对齐。只是一个很小的改变，但是能带来许多的好处：

1. Ql中的数据来自于用户，与用户的距离更近。更能反映用户的需求。毕竟我们现在做的是将Semantic Query Graph转化为SPARQL Query Graph。
2. Q1中的数据相比于知识库小很多，执行语义对齐的速度更快。
3. 我们都知道数据集中的数据是有时效性的。有些存在很久的数据甚至可能都没有意义了。而最近用户的查询日志中反映的就是最近用户们对哪些查询（数据）更加感兴趣。因此一旦将语义对齐转移到向Ql对齐我们就可以进一步缩减对齐所花费的时间。例如，6.5号的查询可能只需要5.30号到6.4号的SPARQL查询日志就足够了。
4. 我们还可以对Ql中的SPARQL查询进行分类，（例如按照用户特征进行分类）。这样，当一个用户进行查询时，我们可以只把这个用户的自然语言问题N，向这个用户所在的那部分SPARQL查询集对齐就可以了。
5. ….

这一块主要是想在提高精确度的基础上又能减少语义对齐的执行时间。具体的效果要通过试验来观察。

经过这一步之后，每一个存在于N中的短语wi，都会有一个候选词列表candidateList。存储着数据集中wi所有可能的表示。



传统做法 本文做法

### 互信息语义消歧

使用互信息消歧。例如我们有一个查询。问的是“张三是哪个足球队的”，当我们对“张三”这个单词做phrases linking的时候，我们很可能可以看到数据集中有非常大量的关于“张三”这个单词的数据。

例如：踢球的张三1，打篮球的张三2，打羽毛球的张三3。根据语义我们其实是可以知道，尽管数据集中有大量的张三，但是事实上只有与“足球队”这个概念相关的“张三1”才是我们真正需要的。

于是我们可以通过收集结点与结点之间边的数量（0-无穷）来对有歧义的词进行筛选。

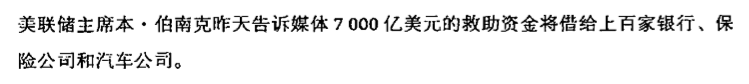
例如在这个例子中很显然，“张三”与“足球”之间有关系。也就是这二者之间应该存在边。那么，如果一个”张三i”与”足球队”这个概念之间没有任何连接边。那么可以直接删除这个”张三i”。于此同时，可以根据边的数量来判断关联性。

这一步的目的是将CandidateList进行一次筛选。留下真正高效可用的数据。

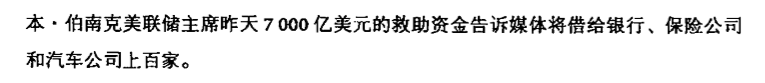
## 隐语义关系以及语义不完全覆盖问题的解决措施

### NLP中判断句子是否有意义

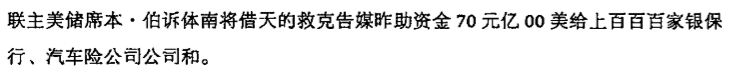
（1）



（2）



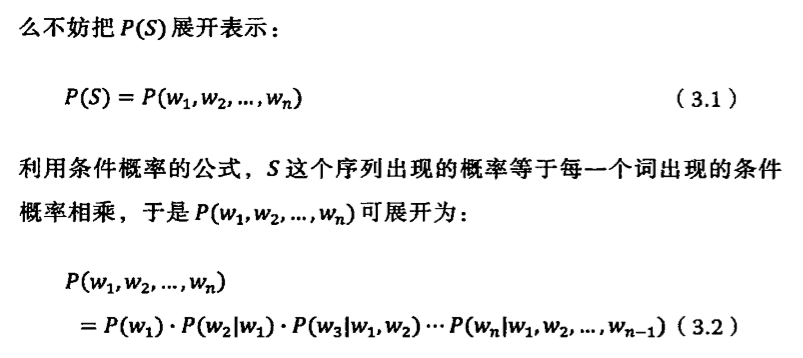
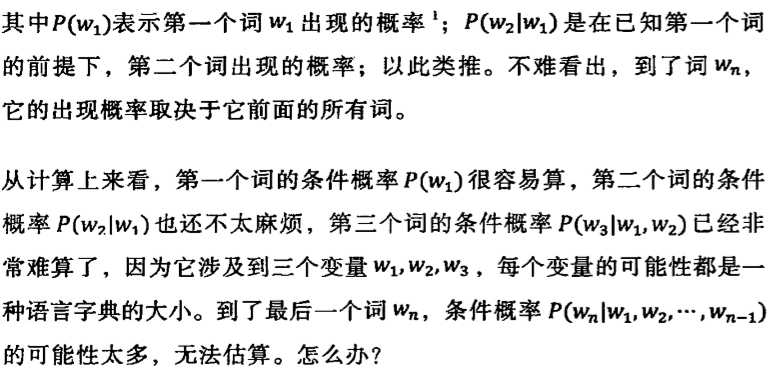
（3）

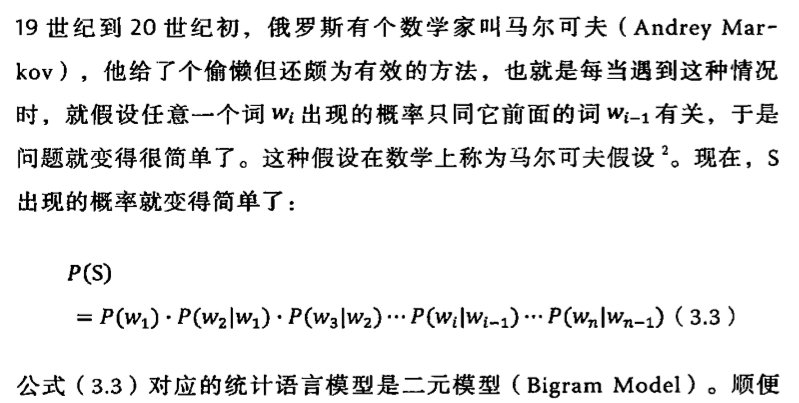


可以看到上面三个句子中，（1）最清晰，也是我们认为最对的。（2）稍微有一些混乱但我们依旧可以理解。（3）就基本不能理解了。

于是我们需要有办法来衡量一句话的语义是否是完整正确的。

假设我们用使用S来代表一个句子，Wi表示组成S的单词。之后我们可以把P（S）表示为如下公式：



到这里，我们就可以使用公式3.3来对一句话的语义是否正确进行判定。我们可以得到任何一句话是正确语义的概率。

**那么很简单，我们将SPARQL查询看做此处的S，而SPARQL中的每一个triple pattern都是S中的一个wi。于是我们就很轻易的可以将这个公式3.3推广到SPARQL预测中去。**

例如，我们已经有SPARQL查询的一部分。之后我们可以利用已有的这一部分{triple pattern}集合来推测下一个最可能出现在这个SPARQL查询中的triple pattern。

于是我们得到如下公式：

### 隐语义关系挖掘

**定义：[隐语义关系]：指N中的一个关系没有显式的出现在两个实体之间。**

**例子：**

“show me all films started by a Chinese actor”.

“show me all films stared by an actor who was born in China”.

Obviously, the latter question has one explicit relation mention “(be) born in”, where the relation in the former one is implicitly mentioned. Therefore, it is difficult to extract these implicit relations.

这个问题在gAnswer这篇论文中给出了定义，并给出了解决方法。解决方法如下：在数据集中进行遍历，看哪个predicate出现在subject与object之间的次数最多就选择哪一个predicate作为这两个实体之间的关系。

我们针对这个问题也给出了新的解决办法。不再遍历数据集，而是遍历Query Log。把匹配从数据集转移到Query Log中，我们认为这更贴合了用户的最近的查询习惯，因而有理由相信这样能提高精度。

与此同时，我们使用上文提出的隐马尔可夫预测模型进行打分。选择更高可能概率的隐式关系组成的triple pattern。

### 不完整语义推测

在这一节中我主要介绍由Nc生成Qc的优点。为什么选择由Nc生成Qc，而不是在SPARQL查询中寻找Qc，除了效率上会有提高以外，这种方式还能提高查询的精度。

**定义[不完全覆盖问题]**：指的是SPARQL不能完全覆盖Nc表达的语义。

我们知道要想完美的使用SPARQL来代替N的语义是很困难的。因此经常会出现不完全覆盖问题。给一个典型的例子。

**例子：**

N1: 原子能的

N2:原子能是什么与原子的应用

N1问题原本是想问，“原子能的应用”。但是由于各种可能的原因会出现如下错误。因此N1可能会变成：

原子能的

原子能的英勇…

当前的N1,只能生成如下SPARQL查询：

{

?x name 原子能

}

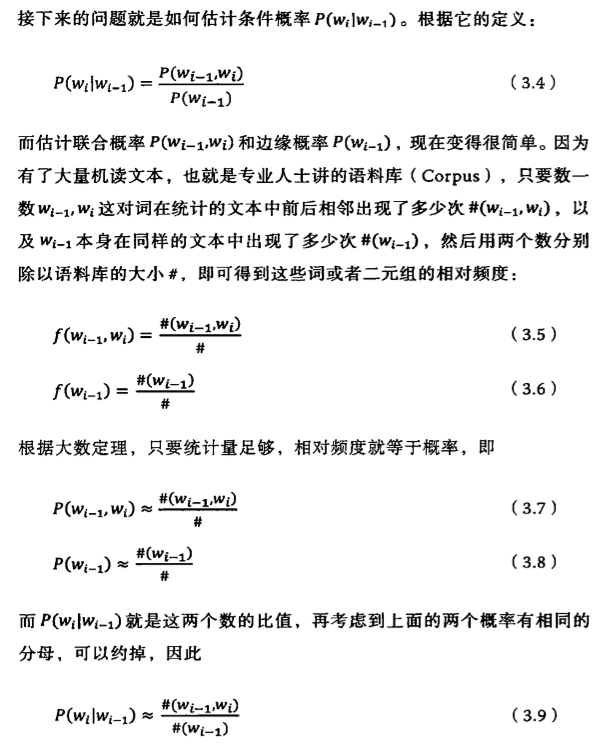
这个SPARQL查询只覆盖了原来的N1中的“原子能”这部分内容。由于无法识别“的应”是什么只能选择抛弃这部分内容。

由于直接在SPARQL中寻找公共结构的时候，这部分信息已经损失掉了。因此我们是无从得知是否存在不完全覆盖问题。因此只能将错就错。而选择使用Nc生成Qc就可以一定程度避免这个问题。

**我们可以使用上文提出的隐马尔可夫模型推测在有当前部分SPARQL的情况下使用哪个triple pattern进行语义补全最合适。**

## 条件概率计算

### 语句语义条件概率计算



事实上在SPARQL中进行预测还与句子中预测一个单词出现的概率稍微有些不同。

在一句话中，单词出现是有次序的。然而SPARQL的triple pattern之间是没有严格的次序之分的。

因此在triple pattern中有：

### SPARQL语义条件概率计算

### 离线阶段任务

离线阶段，统计所有二元关系。即P(Wi|Wi-1)。但是由于SPARQL查询中triple pattern的数量常常少于7个。因此这使得计算多元关系变的可能。所以在真正计算的时候我们会考虑多元关系。

即：我们在SPARQL Log中经过几次遍历后。获得所有可能的P。其中包含二元关系以及多元关系。

但是如果SPARQL Log中SPARQL数量过多，或者用户希望尽量减少数据的预处理时间，那么这种多元的计算方法代价还是比较高。

于是，我们给出由简化版二元关系推出多元关系的公式。假设我们只有二元关系的概率、我们要计算n元关系。

### 条件概率优化

现在把S看做一个完整的SPARQL查询。而Wi就是SPARQL中的triple pattern。于是我们可以通过一样的计算方法，使用隐马尔可夫模型来挑选出在具有的当前triple pattern集合的基础上，哪个triple pattern是最有可能被选出来的。

这个最可能被选出来的triple pattern就是我们选择出用来覆盖N语义的。（语义增）

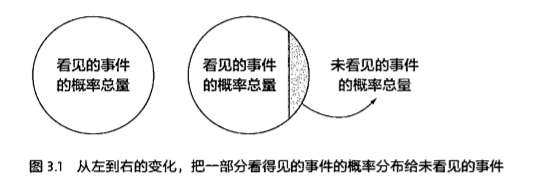
但是由于训练数据量的不足够，可能会出现这样的问题。如果有几千个SPARQL查询用来训练，其中几百个与原子能相关的SPARQL查询反映的语义都是“原子能的应用”。还有几十个SPARQL查询表达的语义是“原子能的制作”。这种情况下我们基本可以认为，有关“原子能的”这个字符串的问题，很可能用户想问的是“原子能的应用”。

但是如果总共只5个与原子能相关的SPARQL查询。其中4个表达的语义是“原子能的应用”，而只有1个SPARQL查询的语义是“原子能的制作”。那么直接说用户的这个问题是原子能的应用的概率是80%显然是不太好的。

因此我们需要解决当样本数量并不是太大的时候，应该怎样进行概率估计这个问题。

**古德-图灵估计：**

对于未知的时间，我们不能认为它发生的概率为0.因此我们要从概率的总量中分很小的比例给予这些没看见的事件。这样一来看的见的那些事件的概率总和就小于1了。具体少多少根据原则“越是不可信的统计折扣越多来进行”。

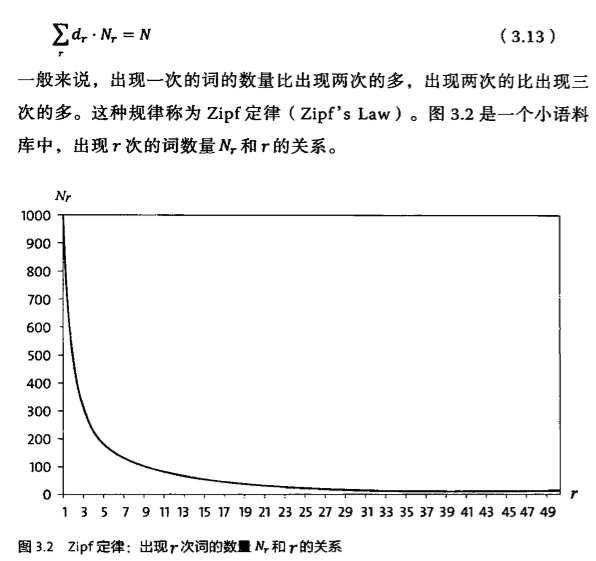


假设SPARQL语料库中，语料库中SPARQL查询的数量为N。

出现r次的SPARQL查询有Nr个，有如下公式：

一般来说，我们直接使用了r/N来表示这个SPARQL的概率了。但是当r比较小的时候，统计信息可能是不可靠的。因此出现r次的那些SPARQL查询在计算他们的概率时要稍微小一些，使用dr来计算。

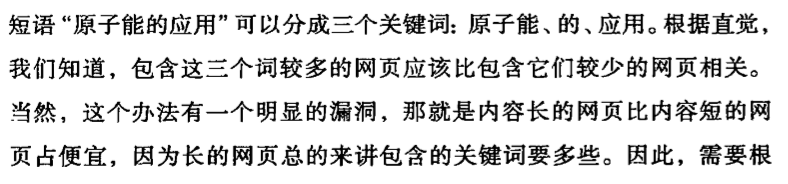


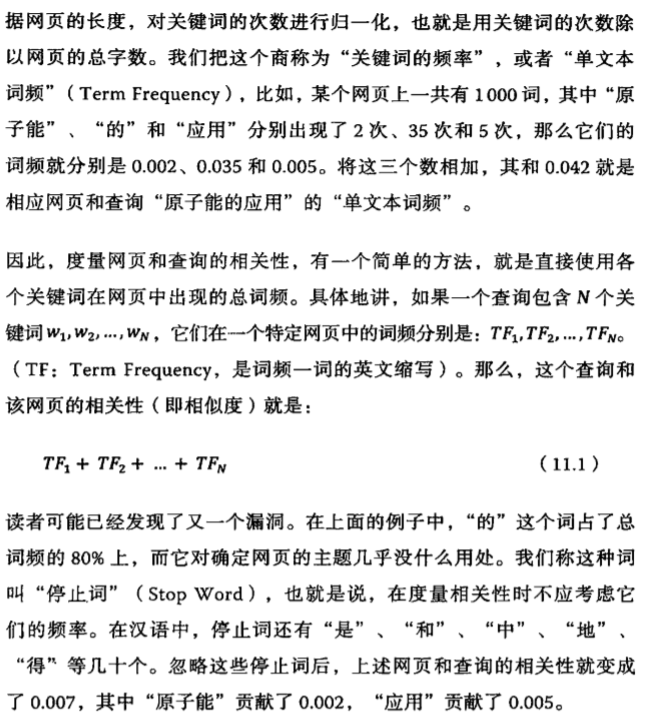


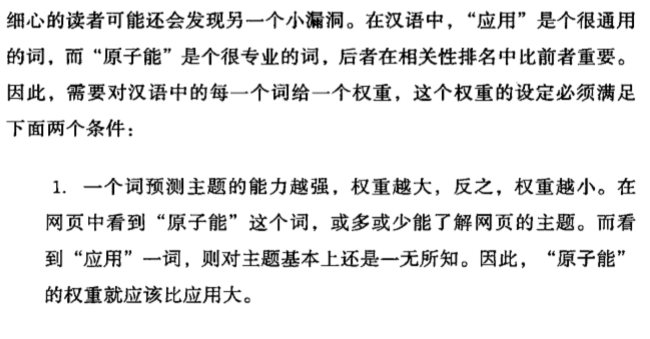
因此使用这种方法做到了对概率的平滑处理。

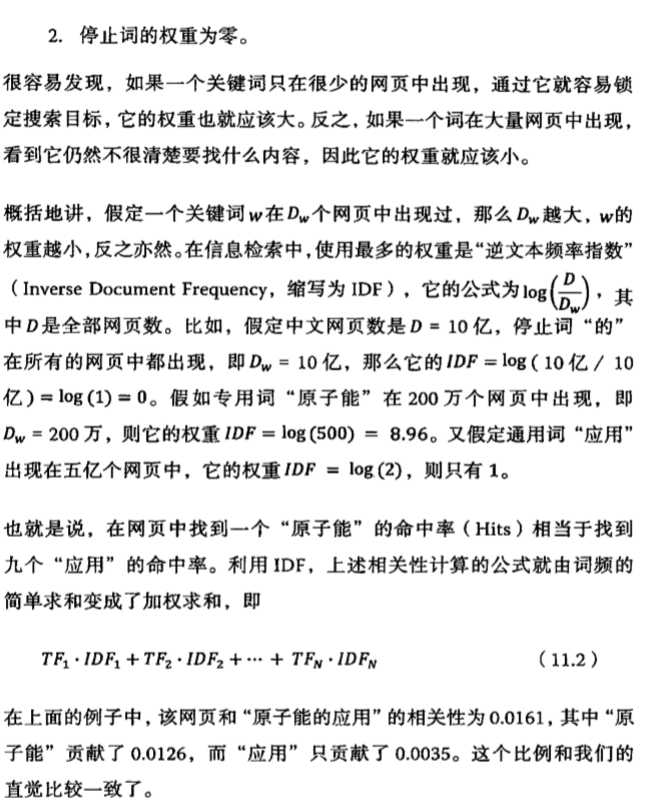
# SPARQL中triple pattern的核心度度量

Sparql中每个triple pattern对SPARQL的限制性是不同的。有的限制性更大，有的限制性更小。引入NLP中的例子：









之后按照这个例子，我们可以对SPARQL中triple pattern的核心度定义如下：

有了核心度定义之后，我们可以对生成的SPARQL查询进行再一次过滤，防止产生冗余的实际价值不大的triple pattern。

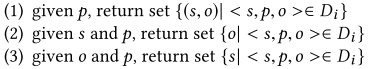
于此同时，triple pattern的核心度还决定了SPARQL查询中的joining order问题。先执行核心度大的triple pattern（因为核心度大代表限制性强，代表结果数量小）。

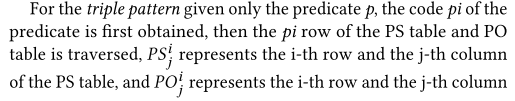
# RDF Engine

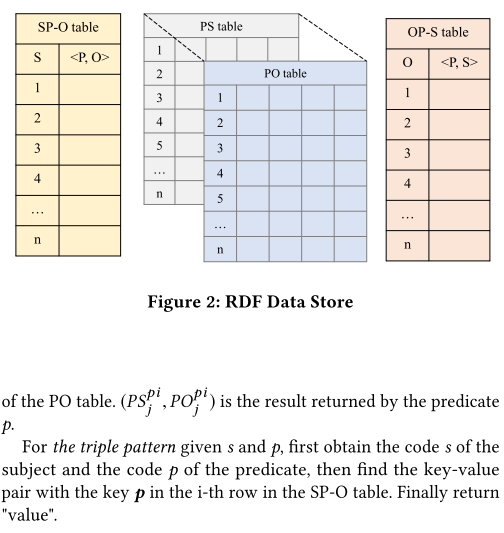
## RDF存储方式

Triple table only uses one table with three columns corresponding to subject, predicate, and object to store RDF data. An index is created on each column to ensure faster join operations. But, this approach of data storage will lead to expensive self joins. Hexastore reduce this cost by using a set of indices that cover all possible permutations of subject, predicate and object. Although this RDF management brings a great improvement, it requires redundant storage of six pieces of data, occupies a large amount of memory. Property table is a wider and flattened representation of RDF data. This representation can’t represent multi-valued attributes. Vertical Partitioning is another representation for RDF data proposed by SW-Store.

In this paper, we propose a novel approach to manage RDF data. Four tables are required on each worker to store RDF data. As shown in the Figure2 these four tables are PS table, PO table, SP-O table, OP-S table. The i-th row in the PS table stores the subject associated with the predicate encoded as i. The i-th row in the PO table stores the object associated with the predicate encoded as i. The i-th row in the SP-O table stores a key-value pair <P, O> associated with the subject encoded as i, where P is a key representing the predicate associated with the subject. O is a collection which stores objects that are related to S and P. Each worker wi uses these four tables to store its local set of triples Di in an in-memory data structure, which supports the following search operations, where s, p and o are subject, predicate and object:





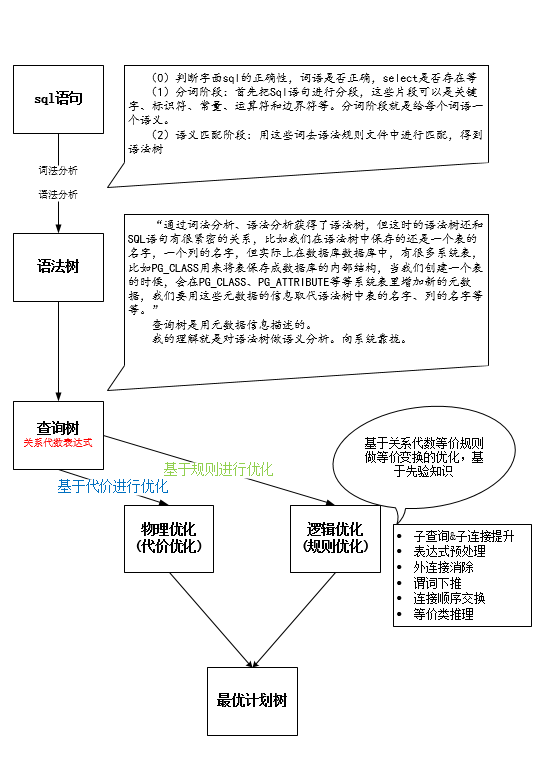


## 查询优化

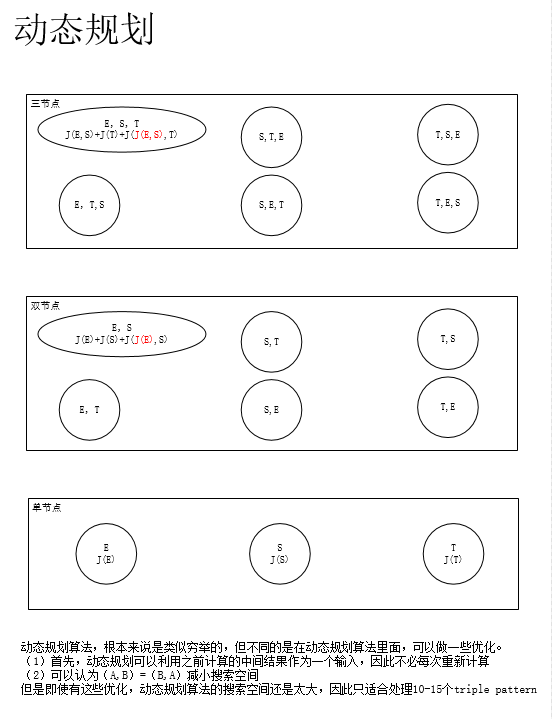
### 引擎处理SQL和SPARQL的流程

要想对查询进行优化处理，首先要了解一下查询引擎对查询的处理流程。其中我们主要是使用动态规划算法来进行物理（代价）优化。

数据库以及常规SPARQL引擎处理一个SQL或者SPARQL的流程如下。



### 动态规划算法寻找连接顺序



其中J()代表估计的COST。这个例子中给出了如何寻找E,S,T三个triple pattern的最佳连接顺序。

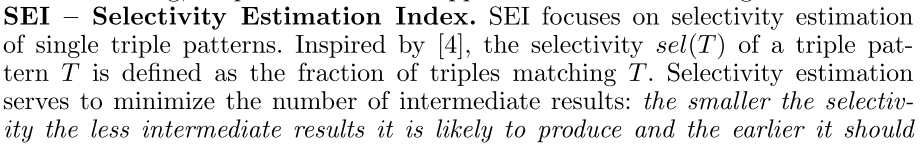
1. 首先在单节点层计算出单个triple pattern的代价
2. 之后在双节点层可以借助单节点层中的代价作为一个输入，因而不必要重新计算。其中J(a,b)代表a,b两个triple pattern连接的代价
3. 之后在三节点层，再借助第二层的结果作为输入，找到E,S,T三个triple pattern之间以哪个顺序执行代价最小。通常来说，SPARQL查询的triple pattern数量不会超过7个。因此动态规划算法就足够了。

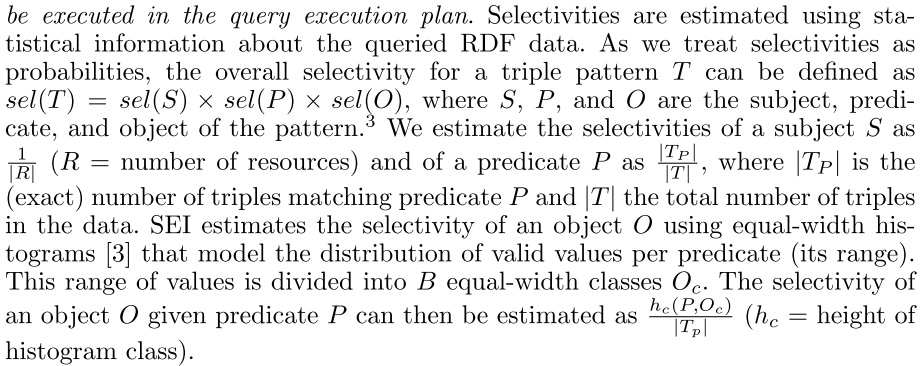
### 查询代价估计

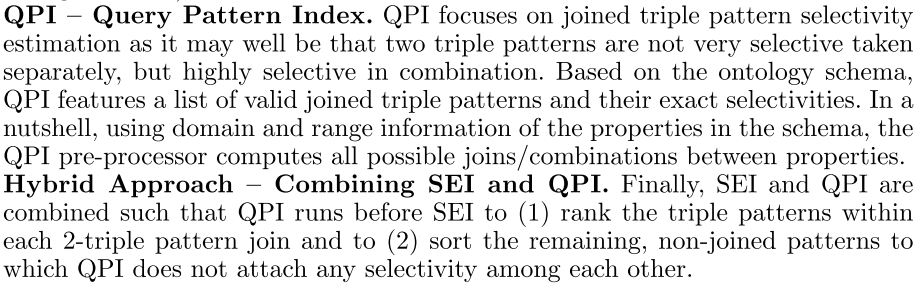
#### Frequent Path（RDF3X）

|  |  |
| --- | --- |
| 描述 | 算法 |
|  |  |

#### 选择度代价计算公式







### Join Method

目前数据库中常用的join方式有三种，分别是loop join, merge Join以及hash Join.

如何合理的选择Join方式也是非常重要的。我们看到一个规律。假设我们有一个SPARQL Join序列。

获得了E的结果R(E)之后,要将R(E)与R(S)结果进行一次Join获得R(E,S)，那么针对这些情况如何选择Join方法呢。

对于SPARQL来说，一般一个triple pattern会具有1个，或者2个变量。通常，只有一个变量的triple pattern中的变量一定已经存在于当前具有的中间结果中，也就是说出现在下一个执行的triple pattern中的变量一定已经存在于当前的结果序列中。所以可以认为只有一个变量的triple pattern是对当前结果的一个筛选，对于这种情况，我们选择hash Join。而具有2个变量的triple pattern通常会给当前现有的中间结果增加一个变量。所以对于这种情况我们选择Merge Join。

当前有一个Join order为：

|  |  |
| --- | --- |
| 1 | ?x p1 ?y. |
| 2 | ?y p2 o1. |
| 3 | ?y p3 ?z. |

1-2之间执行Join，选择Hash Join。因为2只有一个变量，假设1的结果R(1)

如下,第一列是?x的绑定，第二列是?y的绑定：

|  |  |
| --- | --- |
| ?x | ?y |
|  |  |
|  |  |

当我们需要把R(1)与R(2)Join的时候我们看到，R(2)的结果只有一列，也就是?y的绑定。R(2)结果如下：

|  |
| --- |
| ?y |
|  |
|  |

R(1)与R(2)执行Join之后，不会增加列数。反而会导致R(1)结果数量减少，也就是一个结果筛选。

而如果要执行R(1,2)与R(3)则会带来新变量的引入，因此对于这种情况我们使用MergeJoin。

# Extended Work (Parallel Execute Version)

RDF的数据量不断增大，因此将RDF Q/A系统扩展为分布式并行执行非常重要。并且就目前可用于计算的服务器而言，即便集群中只有一台服务器用于计算，其并行度通常大于1，也就是说，即便是单机状态下，仍然可以使用我们的并行执行方法来提高执行效率。

事实上，在我们的RDF Q/A系统中，首先我们对N执行了分组操作，将相类似的N分配到同一组中。

之后我们会获得n组自然语句问题。假设现有的分布式/单机 并行度为m。我们需要将这n组自然语句问题尽可能平均的分为m组。于是这个问题一定会引出下面这个问题，估计查询代价。

## 查询代价(时间)

假设有估计查询代价的函数cost(N)，通过这个函数我们可以估计RDF Q/A系统处理一个N的代价。当然最好有两个代价函数:

(1)

(2)

其中公式1可以给出由N生成的SPARQL查询Q并执行的代价，而公式2可以给出由 生成Qi和Qj的公共查询Qc并执行的代价。于是如果我们设计了多查询优化，公共查询只执行一次，于是就会有如下公式：

（3）

拥有了公式3，就可以计算在有公共结构的情况下，一组N生成SPARQL查询并执行的代价。

目前已经有不少研究工作给出了使用机器学习的方法来预测SPARQL查询的代价的方法，例如14年的论文< A Machine Learning Approach to SPARQL Query Performance Prediction >以及18年WWW的一篇论文都给出了预测SPARQL查询代价的方法。

相比较于单纯的SPARQL查询给出的特征信息，事实上句法分析树中给出的特征信息更多。因此希望可以找到一个比较中肯的方法（机器学习算法）来实现预测由超句法分析树生成的SPARQL查询的代价。

## PCM中句法分析树分组

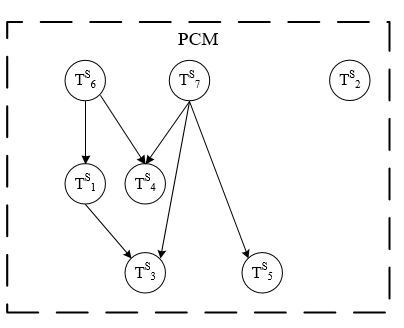


Figure 2 PCM

如图2所示，假设句法分析树之间可以形成这样的连接关系。

**定义：**

**入度**：指向PCM中某一结点TSi的边数量。

**出度**：PCM中某一结点TSi指向其他结点的边数量。

**度**：入度+出度

不难看到，PCM中的所有结点可以分为以下三种情况

* 度为0
* 度为1
* 度>=2

分组的时候，对于度为0和1的结点都很好解决。度为0 的结点例如TS2不和任何其他结点有公共结构，因此自己就是一组。度为1的结点，例如TS5就一定是和指向它的结点TS4绑定在一组。

而针对度大于等于2的结点TSi就比较复杂。TSi可以与所有指向它的结点组合在一组，也可以与它指向的结点组成一组。为了更好的负载均衡。设计如下启发式的算法进行分组。

**算法如下：**

**输入**：PCM ，电脑并行度N

**输出**：将PCM中的所有超句法分析树分为N组

算法分为三种情况：

***PCM中度为1的结点数量>=N***

1. 挑出所有度为1的结点，分为N组。
2. 将所有与度为1的结点相连的结点加入对应组
3. 计算每组的查询代价
4. 将度为大于等于2的结点尝试加入有边与其相连的其他结点所属分组，（要加入查询代价最小的一组）
5. 最后分配度为0的结点。尽可能保持不同组之间代价基本相似。

**PCM中度为1的结点的数量>N/2且<N**

1. 按照度为1的结点的数量m分为m组
2. 将所有与度为1的结点相连的结点加入对应组

由于实际上服务器的并行度为N，而m<N，因此还剩下N-m分组可以进行计算。

1. 因此此步选择将度为0的结点平均分入这N-m个分组中。（但是要保证这N-m个分组中，最大的那一组代价不能超过那m个分组中的最大代价，如果开始超出，则将该超句法分析树分配至前m个分组中代价最小的一组）
2. 处理度大于等于2的结点。每次都选择当前m组中代价最小的那一组加入。（未必要是与该结点连的结点所属的组）

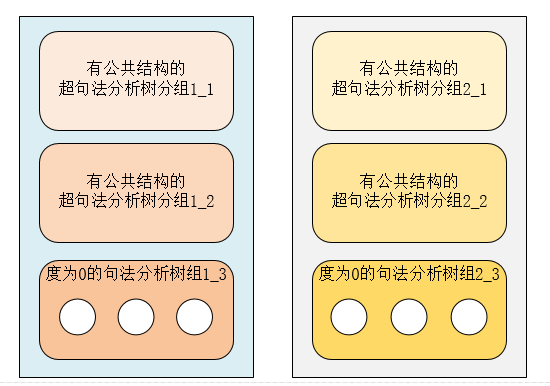
**PCM中度为1的结点数量小于N/2时**

未解决。

N/2是一个设定的参数，可以是N/2,也可以是N/3等等

## 组查询执行顺序问题

经过上面对PCM中句法分析树的分组。给出如下一个例子：



假设计算机的并行度为2，并生成了如上图的两个分组。在超句法分析树生成SPARQL查询Q的部分是没有问题的。这只涉及到计算。计算过程中对存储的压力依赖比较小。

但是当所有的超句法分析树都生成为SPARQL之后，要执行SPARQL查询问题就来了。上图中每个组对内存的耗费情况是不同的。 例如每一组从开始执行到执行完毕这段时间内耗费内存情况如下：

|  |  |  |
| --- | --- | --- |
| 执行顺序 | 组1 | 组2 |
| 1 | 500MB | 500MB |
| 2 | 100MB | 200MB |
| 3 | 3GB | 2GB |

如果只是单纯的要按照上表的执行顺序来执行。如果计算机的内存只有4GB，那么很显然。当执行第三步的时候麻烦就大了。由于内存受限。无法运行。计算机崩溃。但这本来是可以避免的。

**因此在此步应该有两步重要内容需要做。**

1. 如何使用机器学习算法来估测sparql查询占用内存的大小（可以使用每个triple pattern对应的数据所占存储空间大小之和）
2. 设计合理的組执行顺序算法。