1

RDF（Resource Description Framework）采用一种称为Turtle的语法来表示陈述。Turtle是一种简洁和易读的语法，它使用三元组（Triple）的形式来表示关系型数据。

一个RDF陈述包含以下几个要素：

1. 主题（Subject）：表示资源或实体，通常使用统一资源标识符（URI）来唯一标识
2. 谓词（Predicate）：表示主题与陈述之间的关系，也使用URI来标识
3. 宾语（Object）：表示与主题相关的值或目标，可以是一个具体的值（字面量）或另一个资源的URI

RDFS（RDF Schema）的目的是要解决RDF的语义表达能力有限的问题

OWL（Web Ontology Language）则在RDFS之后进一步解决了更丰富的语义建模和推理问题

OWL语言的基础包括三个子语言：

1. OWL Lite：提供了一些基本的建模能力，适用于简单的知识表示和推理。
2. OWL DL（Description Logic）：提供了更丰富的语义表达能力和推理能力，支持复杂的逻辑推理。
3. OWL Full：提供了最大的语义表达能力，但通常不具备完备的推理算法。

OWL具有以下对知识处理的能力：

1. 分类和层次关系：OWL允许定义类与子类之间的层次关系，支持丰富的分类和子类推理。
2. 属性关系和约束：OWL允许定义属性之间的关系，如属性的子属性和属性的限制条件，以及进行属性推理。
3. 实例关系：OWL允许定义实例之间的关系，如实例的类型、属性值等，以支持实例级别的推理。
4. 逻辑推理：OWL提供了基于逻辑的推理能力，可以从已知的知识中推导出新的知识，支持一些常见的推理任务如实例检索、一致性检查等。

2

1

1. 单向性规则（Unidirectional Rule）：如果两个节点在同构匹配中的次序与它们在图中的次序相同，并且它们在同构匹配中的出度与它们在图中的出度相同，则可以进行匹配。
2. 拓扑关系规则（Topology Rule）：这条规则可以分为以下三个部分：
    - 新节点添加规则：当一个节点在同构匹配中被添加时，它必须与已匹配的节点有相同数量的出边和入边。
    - 候选节点规则：当一个节点在同构匹配中没有添加时，它不能与已匹配节点之间有边相连。
    - 部分同构规则：当一个节点在同构匹配中被添加时，它与已匹配节点之间的边的关系必须与图中的拓扑关系一致。
3. 标签规则（Label Rule）：两个节点在同构匹配中必须具有相同的标签。
4. 匹配数规则（Match Count Rule）：每个节点在同构匹配中的入度和出度必须与图中的入度和出度相同。

5. 共享子图规则（Shared Subgraph Rule）：两个节点在同构匹配中如果有相同的邻居节点，那么这些邻居节点也必须在同构匹配中相互匹配。
6. 最大匹配规则（Maximum Match Rule）：对于每个节点，在同构匹配中，其出度必须小于等于其在图中的出度，并且入度必须小于等于其在图中的入度。
7. 反向边规则（Backward Edge Rule）：如果两个节点在同构匹配中有一条边相连，那么在图中也必须有一条相应的边相连。

2

1. 建立超大规模索引
2. 对每个节点执行查询图分解的操作
3. 查询出所有的匹配结果后，按照Stwig的排列顺序执行结果之间的join操作，得到当前节点的结果
4. 对所有节点进行汇总

3

1.
   - V = {N1 N2 N3 N4}
   - L = {人 出演 电影 }
   - E = {E1 E2 E3 E4}
   - A = {姓名 性别 角色 来源 名称}
   - N1.L = N3.L = N4.L = {人} N2.L = {电影}
   - N1.A= N3.A = N4.A = {性别 姓名} N2.A={名称}
   - E1.L=E3.L=E4.L={出演} E2.L={导演}
   - E1.A=E3.A=E4.A={角色 来源}4
2.
```
START s=node(1),e=node(3)
MATCH p=shortestPath(s-[*..10]->e)
RETURN p
```

3.
```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name = '张三' AND p = m
RETURN p.name, m.title
```

4

1.
```
void Compute(MessageIterator* msgs){
    // 遍历由顶点入边传入的消息列表
    for(;!msgs->Done();msgs->Next())
        doSomething();
    // 生成新的顶点值
    *MutableVertexValue() = ...;
    // 生成沿顶点出边发送的消息
    SendMessageToAllNeighbors(...);
}
```

2.
```
# 定义图节点类
class Node:
    def __init__(self, id):
        self.id = id
        self.neighbors = []
        self.two_hop_neighbors = set()
```

```

# 初始化图节点
nodes = [Node(0), Node(1), Node(2), ...]

# 迭代轮次数
iterations = 2

# Gather阶段
def gather(node):
    for neighbor in node.neighbors:
        # 收集邻居节点的邻居节点（一跳邻居）
        for one_hop_neighbor in neighbor.neighbors:
            # 添加到当前节点的二跳邻居集合中
            node.two_hop_neighbors.add(one_hop_neighbor)

# Apply阶段
def apply(node):
    # 更新节点属性为二跳邻居数
    node.neighbors_count = len(node.two_hop_neighbors)

# Scatter阶段
def scatter(node):
    # 向所有邻居节点发送消息
    for neighbor in node.neighbors:
        neighbor.receive_message(node.two_hop_neighbors)

# 图计算迭代
for iteration in range(iterations):
    # Gather阶段
    for node in nodes:
        gather(node)

    # Apply阶段
    for node in nodes:
        apply(node)

    # Scatter阶段
    for node in nodes:
        scatter(node)

# 输出每个节点的二跳邻居数
for node in nodes:
    print("Node", node.id, "Two-hop neighbors:", node.neighbors_count)
```