Efficient Top-k Path Search in Large Knowledge Bases

Huiyan Cao, Mingda Li

1 Motivation

Top-k keyword search in large knowledge bases is always a very important problem in keyword search area. When it comes to keyword search on large scale knowledge bases, we want to propose an algorithm which is not only effective but also efficient. Efficiency plays such an important role in this research domain which inspired us to consider to us Spark to do parallel computing in the table join phase.

2 Challenges

Knowledge bases are often prohibitively large. We want to use Yago which is a large public knowledge base nowadays. Existing algorithm rarely support keyword search on a knowledge base of this size. The main challenge here is how to deal with the scale of the knowledge base and in the meantime come up with an algorithm which has high efficacy.

3 Problem formulation

3.1 Input

In the project, the input are two target entities’ named $N_1$ and $N_2$, a knowledge base $KB$ with necessary indexes, and $K$ top results need to be returned.

3.1.1 Knowledge base

The knowledge base $KB$ is defined as a collection of (subject, predicate, object) triples $KB = \{s, p, o\}$. Following is the definition.

- Subjects or objects represent real-world entities. For example, $Tom\_Cruise$ and $Nicole\_Kidman$ are two well-known actor entities.

- Each predicate refers to the relationship between a subject and an object. Each triple is defined as a fact. For instance, $(Tom\_Cruise, wasMarriedTo, Nicole\_Kidman)$ indicates the fact that $Tom\_Cruise$ was once married to $Nicole\_Kidman$. 
• Predicates with their domains are defined as the schema of the knowledge base. There are mainly two types of relations in knowledge base. The first one is the mapping from entity name to entity labels. The second is the triples to present facts. The data structure for them are label table and fact table. Label table are saved the in form of the pairs. The first element of the pair is the entity name, the second is the entity label. Note that one entity name can map to a set of labels. For example, in table 1, the “Microsoft” entity can map to a set of labels \{Microsoft, Microsoft_Entourage, Microsoft_Outlook\}. Facts are saved as rows in the predicate table. There are three columns in the predicate table. The first column is the label of subject entity, the second column is the name of the predicate, and the last one is the object label. Table 2 shows some facts in the fact table.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tom Cruise</td>
<td>Tom_Cruise</td>
</tr>
<tr>
<td>2</td>
<td>Nicole Kidman</td>
<td>Nicole_Kidman</td>
</tr>
<tr>
<td>3</td>
<td>Microsoft</td>
<td>Microsoft</td>
</tr>
<tr>
<td>4</td>
<td>Microsoft</td>
<td>Microsoft_Entourage</td>
</tr>
<tr>
<td>5</td>
<td>Microsoft</td>
<td>Microsoft_Outlook</td>
</tr>
</tbody>
</table>

Table 1: The label table in knowledge base

<table>
<thead>
<tr>
<th>ID</th>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tom_Cruise</td>
<td>isMarriedTo</td>
<td>Nicole_Kidman</td>
</tr>
<tr>
<td>2</td>
<td>Tom_Cruise</td>
<td>isMarriedTo</td>
<td>Mimi_Rogers</td>
</tr>
<tr>
<td>3</td>
<td>Tom_Cruise</td>
<td>actedIn</td>
<td>Rain_Man</td>
</tr>
<tr>
<td>4</td>
<td>Nicole_Kidman</td>
<td>actedIn</td>
<td>Billy_Bathgate</td>
</tr>
</tbody>
</table>

Table 2: The fact table in knowledge base

3.1.2 Index

Some indexes can be created in order to improve the efficiency of the whole search operation. Based on our algorithm which will be discussed later in section 4, we first create one index (shown in table 3) on subject and another (shown in table 4) on object in the fact table. We also create index (shown in table 5) on entity name in label table.

<table>
<thead>
<tr>
<th>Subject</th>
<th>PredicateAndObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tom_Cruise</td>
<td>{isMarriedTo Nicole_Kidman, isMarriedTo Mimi_Rogers, \ldots}</td>
</tr>
<tr>
<td>Microsoft</td>
<td>{created Microsoft_Entourage, created Microsoft_Outlook, \ldots}</td>
</tr>
</tbody>
</table>

Table 3: index on subject in fact table
### 3.2 Output

The output is the top-k results which record the paths between those two given entities. Note that the path length may vary from result to result. Thus we decide to use strings to store those results. Table 6 illustrates three possible output results when given $N_1 = \text{Tom Cruise}$ and $N_2 = \text{Nicole Kidman}$. Also note that in the path we are looking for, entity can be used both as subject and object.

<table>
<thead>
<tr>
<th>Ranking</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\text{Tom Cruise} \rightarrow \text{wasMarriedTo} \rightarrow \text{Nicole Kidman}$</td>
</tr>
<tr>
<td>2</td>
<td>$\text{Tom Cruise} \rightarrow \text{actIn} \rightarrow \text{Eyes Wide Shut} \leftarrow \text{actIn} \leftarrow \text{Nicole Kidman}$</td>
</tr>
<tr>
<td>3</td>
<td>$\text{Tom Cruise} \rightarrow \text{actIn} \rightarrow \text{Rain Man} \leftarrow \text{actIn} \leftarrow \text{Dustin Hoffman} \rightarrow \text{actIn}$ $\rightarrow \text{Billy Bathgate} \leftarrow \text{actIn} \leftarrow \text{Nicole Kidman}$</td>
</tr>
</tbody>
</table>

Table 6: Possible results between “TomCruise” and “NicoleKidman”

Note that in this example, the longer a path of is, the lower ranking that result gains. Which may not hold true in all occasions. In this example, the first result is the shortest path from $\text{Tom Cruise}$ to $\text{Nicole Kidman}$ which is their once marriage relationship. While, the second result is the path between those two actors via the movie named $\text{Eyes Wide Shut}$ which they co-starred in. The longest path is the third one, in which those two actors are connected by another actor who stars in two different movies with them. Inspired by this example, our baseline ranking method is based on the number of hops on the selected path. The larger this number is, the less important the path is.

### 3.3 Goal

Inspired by [1], which is doing parallel join, we use Spark to do parallel join in the multiple table joining phase, note that in our work, we increased the parallelism. We propose an
top-k path finding algorithm that can efficiently processes top-k relationship search between two given entities over large knowledge bases.

4 Proposed algorithm

Algorithm 1 summarize the major steps of the top-k path finding algorithm. First, by using the nameToLabels index, the labels mapping to \( N_1 \) and \( N_2 \) will be collect and stored in \( L_1 \) and \( L_2 \). Since we use index which will only cost in \( O(1) \), we don’t need to incorporate parallel computing in this step. Second, Algorithm 2 will be used to collect all the related facts which contain one label in \( L_1 \) as either subject or object into a list of sets \( F \). Then it will iteratively collect results by Algorithm 3 and update the list of fact sets \( F \) by Algorithm 4. Details are shown below.

Algorithm 1 Top-k path finding algorithm

**Input:** Two entity name \( N_1 \) and \( N_2 \), knowledge base \( KB \), the index create on entity name of label table nameToLabels, the index create on subject of the fact table subToFacts, the index create on object of the fact table objToFacts, and the number of the paths to return \( K \)

**Output:** Top \( K \) paths between \( N_1 \) and \( N_2 \)

\[
\begin{align*}
\text{ResultQueue} & \leftarrow \emptyset \\
L_1 & = KB[\text{nameToLabels}[N_1]] \\
L_2 & = KB[\text{nameToLabels}[N_2]] \\
F & = \text{GetRelatedFacts}(L_1, \text{subToFacts}, \text{objToFacts}, KB) \\
\textbf{while} \quad \text{ResultQueue.size} < K \quad \textbf{do} \\
& \quad \text{CollectResults}(L_2, F, \text{ResultQueue}) \\
& \quad F = \text{IncOneHop}(F, \text{subToFacts}, \text{objToFacts}, KB) \\
\textbf{return} \quad \text{ResultQueue}
\end{align*}
\]

4.1 Get related facts

Our strategy of generating paths is to start from one entity and increase the path hop by hop. So first of all, we collect all the facts start from one of the given entities. Suppose we choose the first entity and all the labels mapping to the provided entity name have been collected to \( L_1 \). There are two types of fact regarding to each label \( l \) in \( L_1 \). The first type of fact contains \( l \) as subject and the second one contains \( l \) as object. Then processing these two types of fact in parallel will improve the performance of both collecting facts and updating facts. In this algorithm, for each \( l \) in \( L_1 \), we collect the first type of fact in \( F_1 \) and the second type of fact in \( F_2 \) in parallel. Finally, we return the list \( F \) with \( F_1 \) as the first element and \( F_2 \) as the second element.
Algorithm 2  Get related facts of the first entity
Input: The label sets of the first given entity $L_1$, knowledge base $KB$, the index create on subject of the fact table $\text{subToFacts}$, the index create on object of the fact table $\text{objToFacts}$
Output: The fact set $F$ in form of triple $(e_1, p, e_2)$ where $F_1$ stores the facts with the first entity as subject and $F_2$ stores the fact with the first entity as object.

$\text{GetRelatedFacts}(L_1, \text{subToFacts}, \text{objToFacts}, KB)$

\[
F_1, F_2 \leftarrow \emptyset \\
\text{for } i \leftarrow 1, 2 \text{ in parallel do} \\
\quad \text{for } \ell \in L_1 \text{ do} \\
\quad \quad \text{if } i = 1 \text{ then} \\
\quad \quad \quad \text{for } (p, e) \in KB[\text{subToFacts}][\ell] \text{ do} \\
\quad \quad \quad \quad \text{add } (\ell, p, e) \text{ to } F_i \\
\quad \quad \text{else} \\
\quad \quad \quad \text{for } (p, e) \in KB[\text{objToFacts}][\ell] \text{ do} \\
\quad \quad \quad \quad \text{add } (\ell, p, e) \text{ to } F_i
\]

Algorithm 3 Collect results
Input: The label sets of the second given entity $L_2$, the fact set $F$ in form of triple $(e_1, p, e_2)$ where $F_1$ stores the facts with the first entity as subject and $F_2$ stores the fact with the first entity as object, Top-K result queue $\text{ResultQueue}$.

$\text{CollectResults}(L_2, F, \text{ResultQueue})$

\[
\text{for } i \leftarrow 1, 2 \text{ do} \\
\quad \text{for } (\ell, p, e) \in F_i \text{ do} \\
\quad \quad \text{if } e \in L_2 \text{ then} \\
\quad \quad \quad \text{if } i = 1 \text{ then} \\
\quad \quad \quad \quad \text{add } "\ell \leftarrow p \rightarrow e" \text{ to } \text{ResultQueue with Ranking Function} \\
\quad \quad \text{else} \\
\quad \quad \quad \quad \text{add } "\ell \rightarrow p \leftarrow e" \text{ to } \text{ResultQueue with Ranking Function}
\]

4.2 Collect results

Once those two types of facts ($F$) contains one of the labels ($L_1$) mapping to the first entity are collected or updated by increasing one hop, we can collect result paths by checking whether those facts contains one of the labels ($L_2$) mapping to the second entity. Note that parallel computing is not suitable in this process since there is only one variable to collect results. If we use two variables for collecting results of each type of facts, we may waste a lot of time to merge those two result queues regarding to the ranking score.
4.3 Update fact sets

By leveraging the idea of dynamic programming, the prefix path starts from one of the labels \(L_1\) of the first entity will be saved in form of a string as the first element of triple in fact sets. In the update process, we design two layers of parallelism regarding parallel computing. In the outer layer, we process two type of facts in parallel. In the inner layer, we adapt the group join algorithm [1] to increase each fact by one hop. For each fact triple \((e_1, p, e_2)\) in each type of fact set \(F_i\), it will only join with the group of facts with subject or object equals to \(e_2\). The updated list of two set of facts will replace \(F\) in the next iteration.

Algorithm 4 Update the fact set by increasing one hop

**Input:** The fact set \(F\) in form of triple \((e_1, p, e_2)\) where \(F_1\) stores the facts with the first entity as subject and \(F_2\) stores the fact with the first entity as object, knowledge base \(KB\), the index create on subject of the fact table \textbf{subToFacts}, the index create on object of the fact table \textbf{objToFacts}

**Output:** The updated fact set \(newF\) by increasing one hop

\[
\text{IncOneHop}(F, \text{subToFacts}, \text{objToFacts}, KB) = (newF_1, newF_2) \leftarrow (\emptyset)
\]

\[
\text{for } i \leftarrow 1, 2 \text{ in parallel do} \\
\quad \text{for } (e_1, p_1, e_2) \in F_i \text{ in parallel do} \\
\quad \\
\quad \quad \text{for } (p_2, e_3) \in KB[\text{subToFacts}[e_2]] \text{ do} \\
\quad \\
\quad \quad \quad \text{if } i = 1 \text{ then} \\
\quad \quad \quad \quad \quad \text{add } (e_1 - p_1 \rightarrow e_2, p_2, e_3) \text{ to } newF_i \\
\quad \quad \quad \text{else} \\
\quad \quad \quad \quad \quad \text{add } (e_1 \leftarrow p_1 - e_2, p_2, e_3) \text{ to } newF_i \\
\quad \quad \text{for } (p_3, e_4) \in KB[\text{objToFacts}[e_2]] \text{ do} \\
\quad \quad \quad \text{if } i = 1 \text{ then} \\
\quad \quad \quad \quad \quad \text{add } (e_1 - p_1 \rightarrow e_2, p_3, e_4) \text{ to } newF_i \\
\quad \quad \quad \text{else} \\
\quad \quad \quad \quad \quad \text{add } (e_1 \leftarrow p_1 - e_2, p_3, e_4) \text{ to } newF_i \\
\]

\[
\text{return } \{newF_1, newF_2\}
\]

4.4 Ranking the results

There are mainly four type of ranking functions in comparison in our algorithm. The first one is the baseline method which only compares the number of hops of paths. The longer the path is, the less important the relationship might be. So the greater \(Score_1(p)\) in Equation 1 is, the higher the result path \(p\) ranks.

\[
Score_1(p) = \frac{1}{\#\text{of Hops}(p)} \tag{1}
\]

The second one is to compare the mean page rank [2] weight of all the nodes on the path.
It follows the rule that the greater $\text{Score}_2(p)$ in Equation 2 is, the higher path $p$ ranks. Note that each entity name may map to several labels, so we consider both ends when calculating the mean weight.

$$
\text{Score}_2(p) = \frac{\sum_{\text{node } i \text{ on } p} \text{PageRankWeight}(i)}{\# \text{ofNodes}(p)}
$$

(2)

The third method is to compare the mean frequency of the edges on the result path. The intuition is that the more frequent one type of predicate appears in the knowledge base, the less important the path is. So the larger $\text{Score}_3(p)$ in Equation 3 is, the higher path $p$ ranks.

$$
\text{Score}_3(p) = \frac{\# \text{ofEdges}(p)}{\sum_{\text{edge } j \text{ on } p} \text{FrequencyOfThePredicateOf}(j)}
$$

(3)

The last method is the combination of the second and the third method. The measurement is simply the sum of those two scores as shown in Equation 4. The larger it is, the higher the path ranks.

$$
\text{Score}_4(p) = \text{Score}_2(p) + \text{Score}_3(p)
$$

(4)

The implementation of ranking function is achieved by leveraging the data structure of heap. Each path result will be pushed into the heap regarding to its ranking score calculated by one of the equations above. When top k results are collected, top-k paths will be returned by popping k elements from the heap.

5 Experiments

We validate our algorithm by testing it on the Yago knowledge base. Following is the details of the experiment setup.

**Yago.** Yago is a well-known knowledge base which is derived from Wikipedia, WordNet, and GeoName. The version we used in our experiment is Yago2. There are 15372301 labels and 9186527 facts.

**Settings.** We conduct the experiment on MacBook Pro with processor at 2.2 GHz, 16 GB memory and 251 GB disk. The top-k path finding algorithm is implemented in Spark and Scala, respectively, running on Spark 1.6.3, Scala 2.10.5. The Spark is built on top of Hadoop 2.6.0. The knowledge base is stored in MySQL 5.6.35 database. There are two tables created in the database to present the knowledge base. The schema is shown in table 1 and 2. The size of label table is 898MB and the size of fact table is 669MB. Installation and setup details can be found in [3].

---

1 http://resources.mpi-inf.mpg.de/yago-naga/yago1_yago2/download/yago2/
2 http://d3kbcqa49mib13.cloudfront.net/spark-1.6.3-bin-hadoop2.6.tgz
3 https://downloads.lightbend.com/scala/2.10.5/scala-2.10.5.tgz
4 https://archive.apache.org/dist/hadoop/core/hadoop-2.6.0/hadoop-2.6.0.tar.gz
5 https://dev.mysql.com/downloads/file/?id=468992
5.1 Efficiency experiment

To test the efficiency of our algorithm, we compare the running time of the algorithm only adapt the group join algorithm in [1] and our algorithm. We use the queries shown in table 7 and record the average running time of each query which has run three times. The results is shown in Figure 1. Our algorithm is 2 to 6 times faster than the method only use group join.

<table>
<thead>
<tr>
<th>Query Index</th>
<th>$N_1$</th>
<th>$N_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tom Cruise</td>
<td>Nicole Kidman</td>
</tr>
<tr>
<td>2</td>
<td>Tom Cruise</td>
<td>Rain Man</td>
</tr>
<tr>
<td>3</td>
<td>Tom Hanks</td>
<td>Forrest Gump</td>
</tr>
<tr>
<td>4</td>
<td>The Mask</td>
<td>Jim Carrey</td>
</tr>
<tr>
<td>5</td>
<td>Eyes Wide Shut</td>
<td>Nicole Kidman</td>
</tr>
<tr>
<td>6</td>
<td>Robert de Niro</td>
<td>Al Pacino</td>
</tr>
<tr>
<td>7</td>
<td>Sleepless in Seattle</td>
<td>Tom Hanks</td>
</tr>
<tr>
<td>8</td>
<td>Billy Bathgate</td>
<td>Nicole Kidman</td>
</tr>
<tr>
<td>9</td>
<td>Bill Gates</td>
<td>Microsoft</td>
</tr>
<tr>
<td>10</td>
<td>Microsoft</td>
<td>IBM</td>
</tr>
</tbody>
</table>

Table 7: The queries to test top-k path finding algorithm
5.2 Effectiveness experiment

We also conduct experiments to test effectiveness of the ranking functions. For each query in table 7, we collect the top 5 results of each query and do a user study on the ranking results. Users are asked to score each ranking result produced by different measurements on different queries. The highest score is 5 and the lowest is 1. We only manage to collect 6 surveys. Figure 2 shows the results. The baseline method beats the others and the performance of the other three are very close to each other.
Figure 2: Effectiveness test

References

