ABSTRACT

Query response time often influences user experience in the real world. However, it possibly takes more time to answer a query with all its exact solutions, especially when it contains the OPT operations since the OPT operation is the least conventional operator in SPARQL. So it becomes essential to make a trade-off between the query response time and the accuracy of their solutions. In this paper, based on the depth of the OPT operation occurring in a query, we propose an approach to obtain all approximate queries with less depth of the OPT operation. This paper mainly discusses those queries with well-designed patterns since the OPT operation in a well-designed pattern is really “optional”. Firstly, we transform a well-designed pattern in OPT normal form into a well-designed tree, whose inner nodes are labeled by OPT operation and leaf nodes are labeled by patterns containing other operations such as the AND operation and the FILTER operation. Secondly, based on this well-designed tree, we remove “optional” well-designed subtrees with less depth of the OPT operation and then obtain approximate queries with different depths of the OPT operation. Finally, we evaluate the approximate query efficiency with the degree of approximation.

CCS Concepts

•Information systems → Query languages; Query optimization;

Keywords

Semantic Web, RDF, SPARQL, Well-designed patterns, Approximate queries

1. INTRODUCTION

Currently, there is renewed interest in the classical topic of graph databases [11][16][11]. Much of this interest has been sparked by SPARQL: the query language for RDF. Resource Description Framework (RDF) [7] is the standard data model in the Semantic Web. RDF describes the relationships of entities or resources using directed label graph. RDF has a broad range of applications in the Semantic Web, social network, bio-informatics, geographical data, etc [3]. An example in Table 1 has been given to describe the entities of Jon Smith and Liz Ben. For example, in the first line, it describes that the person Jon smith works for Semantic University. SPARQL [13] recommended by W3C has become the standard language for querying RDF data since 2008 by inheriting classical relational languages such as SQL.

| Jon Smith | workFor | Semantic University |
|-----------|---------|--------------------|
| Jon Smith | teachOf | Liz Ben             |
| Jon Smith | rdf:type| professor          |
| Liz Ben   | rdf:type| master             |
| Liz Ben   | advisor | Jon Smith           |
| Liz Ben   | takesCourse | Ontology |

In the process of information retrieval, users’ tolerable waiting time is limited [10]. Users also might have tolerable waiting time for querying RDF data. For a SPARQL query, if it contains the OPT operation, it will take much time to query optional pattern in SPARQL since OPT is the least conventional operator in AND, OPT, FILTER, SELECT and UNION [15]. It has been shown in [12][15] that the complexity of SPARQL query evaluation raises from PTIME-membership for the conjunctive fragment to PSPACE-completeness when OPT operation is considered. So it is important to make a trade-off between query response time and accuracy of solutions, which is a traditional topic in databases [2]. Since it is hard to obtain all exact solutions of a SPARQL query in a fixed time, a natural idea to reduce the response time of SPARQL query is by removing some “optional” parts of this query (i.e., occurrences of the OPT operator). Moreover, we still expect to preserve its “non-optional” part of this query. For instance, consider a pattern \( Q \) as follows:

\[
Q = (\langle x, \text{rdf:type}, \text{professor} \rangle \text{OPT} (\langle x, \text{workFor}, y \rangle \text{OPT} (\langle x, \text{teachOf}, z \rangle)))
\]

Here \( \langle x, \text{rdf:type}, \text{professor} \rangle \) is a “non-optional” pattern in this query while both \( \langle x, \text{workFor}, y \rangle \) and \( \langle x, \text{teachOf}, z \rangle \) are “optional” patterns. Based on this natural idea, there are three possible new patterns with less OPT operators as follows:
Clearly, we can find that $Q_1$ and $Q_2$ are ideal candidates which contain less optional patterns with protecting “non-
opt{Q_2}$ will be referred to as a notion of approximation (for short, BPS’s maximal answer in this paper can be summarized as follows: 

- Any triple from $(I \cup U \cup V) \times (I \cup V) \times (I \cup U \cup V)$ is a pattern (called a triple pattern). A Basic Graph Pattern (BGP) is a set of triple patterns.
- If $P_1$ and $P_2$ are patterns, then so are the following: $P_1 \cup P_2$, $P_1 \cap P_2$, $P_1 \cap P_2$ and $P_1 \cup P_2$.
- If $P$ is a pattern and $S$ is a finite set of variables then $\text{SELECT}_S(P)$ is a pattern.
- If $P$ is a pattern and $C$ is a constraint (defined next), then $P \text{ FILTER } C$ is a pattern; we call $C$ the filter condition. Here, a constraint is a boolean combination of atomic constraints.

The semantics of patterns is defined in terms of sets of so-called mappings, which are simply total functions $\mu: S \to U$ on some finite set $S$ of variables. We denote the domain $S$ of $\mu$ by $\text{dom}(\mu)$.

Now given a graph $G$ and a pattern $P$, we define the semantics of $P$ on $G$, denoted by $[P]_G$, as a set of mappings, in the following manner.

- If $P$ is a triple pattern $(u, v, w)$, then $[P]_G = \{ \mu: \{u, v, w\} \cap V \to U \mid (\mu(u), \mu(v), \mu(w)) \in G\}$.
- If $P$ is of the form $P_1 \cup P_2$, then $[P]_G = [P_1]_G \cup [P_2]_G$.
- If $P$ is of the form $P_1 \cap P_2$, then $[P]_G = [P_1]_G \cap [P_2]_G$, where, for any two sets of mappings $\Omega_1$ and $\Omega_2$, we define $\Omega_1 \cap \Omega_2 = \{ \mu_1 \cap \mu_2 \mid \mu_1 \in \Omega_1 \text{ and } \mu_2 \in \Omega_2 \}$ and $\mu_1 \cap \mu_2$ are called compatible, denoted by $\mu_1 \sim \mu_2$, if they agree on the intersection of their domains, i.e., for every variable $?x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$, we have $\mu_1(?x) = \mu_2(?x)$. Note that when $\mu_1$ and $\mu_2$ are compatible, their union $\mu_1 \cup \mu_2$ is a well-defined mapping; this property is used in the formal definition above.
- If $P$ is of the form $P_1 \cap P_2$, then $[P]_G = ([P_1]_G \cap [P_2]_G) \cup ([P_1]_G \cap [P_2]_G)$, where, for any two sets of mappings $\Omega_1$ and $\Omega_2$, we define $\Omega_1 \times \Omega_2 = \{ (\mu_1 \cap \mu_2) \mid \mu_1 \in \Omega_1 \text{ and } \mu_2 \in \Omega_2 \}$ and $\mu_1 \cap \mu_2$ are called compatible, denoted by $\mu_1 \sim \mu_2$, if they agree on the intersection of their domains.
- If $P$ is of the form $P_1 \cap P_2$, then $[P]_G = \{ \mu | \mu \in [P_1]_G \cap \Omega \}$.
- If $P$ is of the form $P_1 \text{ FILTER } C$, then $[P]_G = \{ \mu | \mu(C) = \text{true} \}$.

The rest of this paper is organized as follows: Section 2 briefly introduces the SPARQL and conception of well-designed patterns. Section 3 defines the $k$-approximation queries. Section 4 presents the well-designed tree to capture $k$-approximation queries and Section 5 evaluates experimental results. Finally, Section 6 summarizes the paper.

2. PRELIMINARIES

In this section, we introduce the syntax and semantics of SPARQL 1.0 and well-designed patterns [12].

2.1 RDF

Let $I$, $B$ and $L$ be infinite sets of IRIs, blank nodes and literals, respectively. These three sets are pairwise disjoint. We denote the union $I \cup B \cup L$ by $U$, and elements of $I \cup L$ will be referred to as constants.

A triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called an RDF triple. An RDF graph is a finite set of RDF triples.

2.2 The Syntax and Semantics of SPARQL

Assume furthermore an infinite set $V$ of variables, disjoint from $U$. The convention is to write variables starting with the character ‘?’.” SPARQL patterns are inductively defined as follows:

- Any triple from $(I \cup U \cup V) \times (I \cup V) \times (I \cup U \cup V)$ is a pattern (called a triple pattern). A Basic Graph Pattern (BGP) is a set of triple patterns.
- If $P_1$ and $P_2$ are patterns, then so are the following: $P_1 \cup P_2$, $P_1 \cap P_2$ and $P_1 \cap P_2$.
- If $P$ is a pattern and $S$ is a finite set of variables then $\text{SELECT}_S(P)$ is a pattern.
- If $P$ is a pattern and $C$ is a constraint (defined next), then $P \text{ FILTER } C$ is a pattern; we call $C$ the filter condition. Here, a constraint is a boolean combination of atomic constraints.

The semantics of patterns is defined in terms of sets of so-called mappings, which are simply total functions $\mu: S \to U$ on some finite set $S$ of variables. We denote the domain $S$ of $\mu$ by $\text{dom}(\mu)$.

Now given a graph $G$ and a pattern $P$, we define the semantics of $P$ on $G$, denoted by $[P]_G$, as a set of mappings, in the following manner.

- If $P$ is a triple pattern $(u, v, w)$, then $[P]_G = \{ \mu: \{u, v, w\} \cap V \to U \mid (\mu(u), \mu(v), \mu(w)) \in G\}$.
- If $P$ is of the form $P_1 \cup P_2$, then $[P]_G = [P_1]_G \cup [P_2]_G$.
- If $P$ is of the form $P_1 \cap P_2$, then $[P]_G = [P_1]_G \cap [P_2]_G$.
- If $P$ is of the form $P_1 \cap P_2$, then $[P]_G = [P_1]_G \cap [P_2]_G$, where, for any two sets of mappings $\Omega_1$ and $\Omega_2$, we define $\Omega_1 \cap \Omega_2 = \{ \mu_1 \cap \mu_2 \mid \mu_1 \in \Omega_1 \text{ and } \mu_2 \in \Omega_2 \}$ and $\mu_1 \cap \mu_2$ are called compatible, denoted by $\mu_1 \sim \mu_2$, if they agree on the intersection of their domains.
- If $P$ is of the form $P_1 \cap P_2$, then $[P]_G = [P_1]_G \cap [P_2]_G$.
- If $P$ is of the form $P_1 \text{ FILTER } C$, then $[P]_G = \{ \mu | \mu(C) = \text{true} \}$.

The rest of this paper is organized as follows: Section 2 briefly introduces the SPARQL and conception of well-designed patterns. Section 3 defines the $k$-approximation queries. Section 4 presents the well-designed tree to capture $k$-approximation queries and Section 5 evaluates experimental results. Finally, Section 6 summarizes the paper.

2.3 Well-designed Patterns

The notion of well-designed patterns is introduced to characterize the weak monotonicity [12].

A UNION-free pattern $P$ is well-designed if the following holds:

- $P$ is safe, that is, each subpattern of the form $Q \text{ FILTER } C$ of $P$ holds the condition: $\text{var}(C) \subseteq \text{var}(Q)$.
- For every subpattern $P' = (P_1 \text{ OPT } P_2)$ of $P$ and for every variable $?x$ occurring in $P$, the following condition hold: If $?x$ occurs both inside $P_2$ and outside $P'$,
3. APPROXIMATE QUERIES

In this section, we introduce our approximation method in the \textit{OPT normal form}.

3.1 OPT Normal Form

A UNION-free pattern $P$ is in \textit{OPT normal form} \cite{12} if $P$ meets one of the following two conditions:

- $P$ is constructed by using only the AND and FILTER operators;
- $P = (P_1 \text{ OPT } P_2)$ where $P_1$ and $P_2$ patterns are in OPT normal form.

For instance, the pattern $Q$ stated in Section \ref{sec:1} is in OPT normal form. However, consider the pattern (((?x, p, ?y) OPT (?y, q, ?z)) AND (?r, r, ?z)) is not in OPT normal form.

Note that all patterns in OPT normal form have the following form:

$$P_0 \text{ OPT } P_1 \text{ OPT } \ldots \text{ OPT } P_m$$  \hspace{1cm} (1)

where $P_0$ is an OPT-free pattern, that is, $P_0$ contains only AND and FILTER operations (called \textit{AF-pattern}). In this sense, we use BGP($P$) to denote $P_0$ and $O(P)$ to denote \{P_1, \ldots, P_m\}, i.e., the collection of optional patterns occurring in $P$.

**Proposition 3.1.** \cite[Theorem 4.11]{12} For every UNION-free well-designed pattern $P$, there exists a pattern $Q$ in OPT normal form such that $P$ and $Q$ are equivalent.

In the proof of Proposition 3.1, we apply three rewriting rules based on the following equations: let $P, Q, R$ be patterns and $C$ a constraint,

- $(P \text{ OPT } R) \text{ FILTER } C \equiv (P \text{ FILTER } C) \text{ OPT } R$;
- $(P \text{ OPT } R)$ AND $Q \equiv (P \text{ AND } Q) \text{ OPT } R$;
- $P$ AND $(Q \text{ OPT } R) \equiv (P \text{ AND } Q) \text{ OPT } R$.

Since each UNION-free well-designed pattern is equivalent to a pattern in OPT normal form by Proposition 3.1, we mainly consider all well-designed patterns in OPT normal form in the following.

To further observe some features of patterns in OPT normal form, we consider a complicated pattern $P$, where the OPT operation is deeply nested, as follows:

$$P = (t_1 \text{ OPT } (t_2 \text{ OPT } t_3)) \text{ OPT } (t_4 \text{ OPT } t_5).$$  \hspace{1cm} (2)

Note that, in $P$, $t_1$ is non-optional while $t_2, t_3, t_4$ and $t_5$ are optional. Furthermore, if we consider the subpattern ($t_2$ OPT $t_3$), $t_2$ is non-optional while $t_3$ is still optional. Analogously, if we consider the subpattern ($t_4$ OPT $t_5$), $t_4$ is non-optional while $t_5$ is still optional. Now, if we observe the figure of $P$ shown in Figure \ref{fig:1} $t_2$ and $t_4$ are on top of $t_3$ and $t_4$, respectively.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure1.png}
\caption{The figure of OPT normal form}
\end{figure}

3.2 OPT-depth in OPT Normal Form

To characterize the different levels of optional patterns, we define \textit{OPT-depth} of patterns in OPT normal form.

**Definition 3.1.** (OPT-DEPTH). Let $P$ be a pattern in OPT normal form. We use $\text{dep}(P)$ to denote its OPT-depth as follows:

- $\text{dep}(P) = 0$ if $P$ is an AF-pattern;
- $\text{dep}(P) = \max\{\text{dep}(P_1), \ldots, \text{dep}(P_m)\} + 1$ if $O(P) = \{P_1, \ldots, P_m\}$.

For instance, the OPT-depth of the pattern $Q$ stated in Section \ref{sec:1} and the pattern $P$ in Equation \ref{eq:2} are 2.

3.3 Approximate Queries

To define our approximate queries, we introduce an important notion called reduction \cite{12}.

We say that a pattern $P'$ is a \textit{reduction} of a pattern $P$, if $P'$ can be obtained from $P$ by replacing subpattern ($P_1$ OPT $P_2$) with $P_1$, that is, $P'$ is obtained by deleting some optional parts of $P$. The reflexive and transitive closure of the reduction relation is denoted by $\preceq$. In this sense, for a pattern, its reductions can be taken as “inexact” patterns, which can be obtained by reducing the OPT operation. For instance, in Section \ref{sec:1} $Q_1$ and $Q_2$ are reductions of $Q$.

Inspired from the notion of reduction, we introduce our $k$-approximate patterns.

**Definition 3.2.** (K-APPROXIMATION). Let $P$ be a pattern in OPT normal form ($P_0$ OPT $P_1$ OPT $\ldots$ OPT $P_m$) and $k$ be a natural number. The $k$-approximate pattern of $P$ (written as $P^{(k)}$) can be obtained in the following inductive way:

- $P^{(0)} = \text{BGP}(P)$ if $k = 0$;
- $P^{(k)} = P_0$ OPT $P_1^{(k-1)}$ OPT $\ldots$ OPT $P_m^{(k-1)}$ if $1 \leq k \leq \text{dep}(P) - 1$;
- $P^{(k)} = P$ if $k \geq \text{dep}(P)$.

Intuitively, approximate patterns are subpatterns obtained by reducing their OPT-depths. In this sense, our approximation generalizes reduction \cite{13} in a fine-grained way. Since there exists the unique OPT-depth for each OPT in OPT normal form, we have the following proposition:

**Proposition 3.2.** Let $P$ be a pattern in OPT normal form and $k$ be a natural number. $P^{(k)}$ exists and $P^{(k)}$ is unique.

For instance, in Section \ref{sec:1} $Q^{(0)} = Q_1$ and $Q^{(1)} = Q_2$. In Equation \ref{eq:2}, $P^{(0)} = t_1$ and $P^{(1)} = ((t_1$ OPT $t_2$ OPT $t_4$). $Q^{(0)}$ and $Q^{(1)}$ are the reductions of $Q$ Analagously, $P^{(0)}$ and $P^{(1)}$ are the reductions of $P$. 

\footnote{We abbreviate $(P_0$ OPT $P_1$ OPT $\ldots$ OPT $P_m$) as $P_0$ OPT $P_1$ OPT $\ldots$ OPT $P_m$.}
4. K-APPROXIMATION COMPUTATION

In this section, we propose a method to compute all approximate patterns based on a redesigned parse tree called well-designed tree.

Now, we introduce the notion of well-designed tree.

**Definition 4.1 (well-designed tree).** Let \( P \) be a well-designed pattern in OPT normal form. A well-designed tree \( T \) based on \( P \) is a redesigned parse tree, which can be defined as follows:

- All inner nodes in \( T \) are labeled by OPT operations and leaf nodes are labeled by AF-patterns.
- For each subpattern \( (P_1 \ AND \ P_2) \) of \( P \), the well-designed tree \( T_1 \) of \( P_1 \) and the well-designed tree \( T_2 \) of \( P_2 \) have the same parent node.

For instance, given a pattern \( P \) in OPT normal form,

\[ P = (((t_1 \ AND \ t_3) \ FILTER \ C) \ AND \ t_2) \ AND \ ((t_4 \ AND \ t_5) \ FILTER \ C) \ FILTER \ C). \]

We write \( ((t_1 \ AND \ t_3) \ FILTER \ C) \) as \( p_0 \) for short, which is the non-optional part of \( P \). The well-designed tree \( T \) is shown in Figure 2.

**Figure 2: Well-designed Tree**

![Well-designed Tree](image)

Some pruning strategies can be applied to the well-designed tree to achieve \( k \)-approximation. After removing optional subtrees from the well-designed tree, we get a \( k \)-approximation spanning tree (KST for short) which is also a well-designed tree. We denote a \( k \)-approximation spanning tree from well-designed tree \( T \) as \( KST_T^k \). In order to obtain \( KST_T^k \), we define a special traversal method for the well-designed tree based on the conception of OPT-depth, called Left-Deep Level Traversal. Before defining Left-Deep Level Traversal, we provide a partial traversal approach called Leftmost Traversal.

For a well-designed tree, Leftmost Traversal of this tree is by only traversing the left subtree after visiting root node. For instance, consider \( T \) in Figure 2, the leftmost traversal of \( T \) is denoted by \( LT(T) = \{p_0, OPT_2, p_1\} \). Left-Deep Level Traversal of the well-designed tree is proposed as follows:

**Definition 4.2 (left-deep level traversal).** Let \( T \) be a well-designed tree. Left-Deep Level Traversal denoted by \( LD(T) \) is composed of levels. \( level(i) \) can be obtained by leftmost traversing each node’s right children node (called candidate) in \( level(i - 1) \). Especially, \( level(0) = LT(T) \).

For each subtree \( t \) in the well-designed tree, the leftmost leaf node written as \( LM(t) \) is the non-optional part of \( t \). For instance, for the well-designed tree \( T \) in Figure 2, \( LM(T) = \{p_0\} \). We construct \( KST_T^0 \) by removing the subtrees below \( level(k - 1) \) from \( T \). Particularly, \( KST_T^0 \) can be built by returning \( LM(T) \).

In the process of building \( KST_T^k \), firstly we compute each node’s candidate in \( level(k - 1) \). Secondly we obtain the \( LM(n) \) for each OPT node \( n \) in \( level(k - 1) \). Finally \( KST_T^k \) can be constructed by replacing the leftmost nodes with corresponding OPT nodes in \( T \). We obtain the \( k \)-approximation query through traversing on \( KST_T^k \). The process of building \( KST_T^k \) is described in Algorithm 4.3.

**Example 4.1.** Consider the well-designed tree \( T \) in Figure 2 from pattern \( P \). The LD(T) with candidates and leftmost list can be described as follows:

| Level | Traversal List | Candidates | Leftmost |
|-------|----------------|------------|----------|
| 0     | OPT_1, OPT_2, p_0 | OPT_3, t_4, t_5 | t_4, x   |
| 1     | OPT_3, OPT_4, t_4, t_2 | OPT_5, t_5 | t_6, x   |
| 2     | OPT_5, t_6, t_5 | t_7 | x         |

In \( KST_T^0 \), \( p_0 \) is set as the root node without any child node. If we want to obtain \( KST_T^1 \), we can replace \( t_4 \) with OPT_3 in \( T \) based on level(0). Analogously, \( KST_T^2 \) can be obtained by replacing \( t_6 \) with OPT_5 in \( T \) based on level(1). Since \( \text{dep}(P) = 3 \), \( KST_T^3 \) is regarded as \( T \) itself. Both \( KST_T^1 \) and \( KST_T^2 \) are shown in Figure 3.

**Figure 3: Approximation Spanning Tree**

![Approximation Spanning Tree](image)

(a) 1-approximation  (b) 2-approximation

\([P]^1\) and \([P]^2\) are shown as follows:

\([P]^1 = (((t_1 \ AND \ t_3) \ FILTER \ C) \ AND \ t_2) \ AND \ ((t_4 \ AND \ t_5) \ FILTER \ C) \ FILTER \ C)\)

and

\([P]^2 = (((t_1 \ AND \ t_3) \ FILTER \ C) \ AND \ t_2) \ AND \ ((t_4 \ AND \ t_5) \ FILTER \ C)\)

5. EXPERIMENTS AND EVALUATIONS

This section presents our experiments. The purpose of the experiments is to evaluate (1) the performance improvement of approximate well-designed SPARQL queries, and (2) the appropriate \( k \) to reduce the users’ waiting time for solutions.

5.1 Experiments

**Implementations and running environment.**

All experiments were carried out on a machine running Linux, which has one CPU with four cores of 2.40GHz, 32GB

\(\text{We use } \times \text{ to denote that for each non-OPT node } n \text{ in candidates, there exist no corresponding } LM(n) \text{ in leftmost list.}\)
Algorithm 1 K-approximation Spanning Tree

Input: Well-designed tree $T$ from pattern $P$, Leftmost list \textit{leftmost}, and $k$-approximation with $k$

Initialize Candidate $\textit{candidate}$ with $T$, $i \leftarrow 0$

Output: $K$-approximation Spanning Tree

1: if $k = 0$ then
2:   return $LM(T)$
3: else if $k \geq \text{dep}(P)$ then
4:   return $T$
5: else
6:   while $i \neq k$ do
7:      level($i$) $\leftarrow LT(\textit{candidate})$
8:      $\textit{candidate} \leftarrow \text{GetCandidate}(\text{level}(i))$.
9:      for each node in $\textit{candidate}$ do
10:         if node is OPT then
11:            \textit{leftmost} $\leftarrow LM($\textit{node}$)$
12:       end if
13:    end for
14:   end while
15: Replace the nodes in \textit{leftmost} with corresponding OPT nodes in $T$.
16: return $T$
17: end if

memory and 500GB disk storage. All of the algorithms were implemented in JAVA with Eclipse as our compiler. Jena\(^6\) (Jena-3.0.1) and Sesame\(^5\) (Sesame-4.1.1) are used as the underlying query engines of approximate queries.

Dataset.

We used LUBM\(^4\) as the dataset in our experiments to look for the relationship between approximate query efficiency and $k$. LUBM, which features an ontology for the university domain, is a standard benchmark to evaluate the performance of Semantic Web repositories. In our experiments, we used LUBM1, LUBM5 and LUBM10 as query datasets shown in Table 2.

Table 2: Profiles of datasets

| Dataset | Number of Triples | NT File Size(bytes) |
|---------|-------------------|---------------------|
| LUBM1   | 103,104           | 14,497,954          |
| LUBM5   | 645,836           | 90,960,405          |
| LUBM10  | 1,316,701         | 185,474,846         |

SPARQL queries.

The queries over LUBM were designed as 4 forms in Table 3. Obviously, OPT nesting in $Q_4$ is the most complex among 4 forms. Furthermore, we built AND and FILTER operations in each query. All of query patterns have $k$ ranging from 0 to 4. Specially, since $\text{dep}(Q_2)$ is 1, we regard $k$-approximate query as $Q_2$ itself when $k > 1$.

Table 3: Queries on LUBM

| Query | Well-designed tree | Amount of OPT |
|-------|--------------------|--------------|
| $Q_1$ | zigzag tree        | 9            |
| $Q_2$ | left-deep tree     | 4            |
| $Q_3$ | right-deep tree    | 4            |
| $Q_4$ | full tree          | 15           |

The amount of OPT after approximation.

The amount of OPT with different $k$ is shown in Table 4. Clearly, the amount of OPT is decreasing after approximation since our approximation method can reduce OPT-
depth. Note that when $k$ is 4, query is itself without any approximation.

Table 4: Amount of OPT after approximation

| $k$ | $k=0$ | $k=1$ | $k=2$ | $k=3$ | $k=4$ |
|-----|-------|-------|-------|-------|-------|
| $Q_1$ | 0 | 2 | 5 | 8 | 9 |
| $Q_2$ | 0 | 4 | 4 | 4 | 4 |
| $Q_3$ | 0 | 1 | 2 | 3 | 4 |
| $Q_4$ | 0 | 4 | 10 | 14 | 15 |

5.2 Efficiency of Approximate Queries

For a well-designed query $Q$ and its $k$-approximation query $Q^{(k)}$, $Q^{(4)}$ is more closed to $Q$ with higher value of $k$. The variation tendencies of query response time shown in Figure 4 and Figure 5 are similar. Query efficiency is promoted with lower response time when $k$ is decreasing (approximation degree becomes larger). Furthermore, there has been a significant increase in query efficiency when the dataset scale grows up. For instance, we observe $Q_4$, which corresponds to a full well-designed tree. When the dataset is LUBM10, its query response time is more than an hour implemented by Jena and Sesame without any approximation ($Q_4^{(4)}$). However, the response time of $Q_4^{(1)}$ is less than a minute. Furthermore, comparing $Q_4^{(3)}$ with $Q_4^{(4)}$ implemented by Jena and Sesame, an approximately decrease of 25% in the query response time has shown in both Figure 4 and Figure 5. $Q_2$ and $Q_3$ has less time than $Q_1$ and $Q_4$ since $Q_2$ and $Q_3$ have less OPT amounts and simpler OPT nestings.

Approximate queries can efficiently reduce the query response time and users’ waiting time. An appropriate $k$ can be determined to reduce users’ waiting time for solutions since users’ tolerable waiting time is limited. We assume that $Q_4$ on LUBM10 comes from users, and it takes more than an hour to answer $Q_4^{(4)}$ by Jena and Sesame if users want to obtain all exact solutions, which might lead to bad user experience. In this scene, it can be approximated as $Q_4^{(1)}$ to improve user experience within a minute waiting time.

More results of $k$-approximation can be found in the online demo website: [http://123.56.79.184/approximate.html](http://123.56.79.184/approximate.html)

6. CONCLUSION

In this paper, we have presented the approximation of well-designed SPARQL patterns in OPT normal form based on the depth of OPT operation. Theoretically, our proposal $k$-approximation generalizes reductions of patterns in a fine-grained way. The $k$-approximation provides rich and various approximate queries to answer user’s query within a fixed time. Our experimental results show that our approximation on the depth of OPT operation is reasonable and useful.

In the future, we are going to handle other non-well-designed patterns and deal with more operations such as UNION. Besides, we will extend the approximation method to obtain other approximation queries.

7. ACKNOWLEDGMENTS

This work is supported by the program of the National Natural Science Foundation of China (NSFC) under 61502336, 61572353, 61373935 and the National High-tech R&D Program of China (863 Program) under 2013AA013204. Xiaowang Zhang is supported by the project-sponsored by School of Computer Science and Technology in Tianjin University.

8. REFERENCES

[1] R. Angles and C. Gutierrez. Survey of graph database models. ACM Computing Surveys, 40(1)(2008): article 1.
[2] S. Abiteboul, H. Richard, and V. Vianu. Foundations of databases. Addison Wesley, page 9:369-56, 1995.
[3] S. Abiteboul, P. Buneman, and D. Suciu. Data on the Web: from relations to semistructured data and XML. Morgan Kaufmann, 2000.
[4] P. Barcelo, R. Pichler, and S. Skritek. Efficient evaluation and approximation of well-designed pattern trees. In Proc. of PODS 2015, pages 131–144. ACM, 2015.
[5] J. Broekstra, A. Kampman, and F. V. Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. Springer Berlin Heidelberg, 2002.
[6] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In Proc. of WWW 2004, pages 74–83, 2004.
[7] G. Klyne, C. J. Jeremy, and B. McBride. Resource description framework (RDF): Concepts and abstract syntax. W3C Recommendation, 2004.
[8] G.H.L. Fletcher, M. Gyssens, D. Leinders, J. Van den Bussche, D. Van Gucht, S. Vansummeren, and Y. Wu. Relative expressive power of navigational querying on graphs. In Proc. of ICDT 2011, pp.197–207, 2011.
[9] A. Letelier, J. Pérez, R. Pichler, and S. Skritek. Static analysis and optimization of semantic web queries. In Proc. of PODS 2012, 38(4):84–87, 2012.
[10] F. H. Nah. A study on tolerable waiting time: How long are web users willing to wait? Behaviour and Information Technology, 23(3), 2003.
[11] J. Hellings, B. Kuijpers, J. Van den Bussche, and X. Zhang. Walk logic as a framework for path query languages on graph databases. In Proc. of ICDT 2013, pp.117–128, 2011.
[12] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. ACM Transactions on Database Systems, 34(3):30–43, 2009.
[13] E. Prud’hommeaux and A. Seaborne. SPARQL query language for RDF. W3C Recommendation, 2008.
[14] M. Saleem, M. I. Ali, A. Hogan, Q. Mehmoood, and A.-C. N. Ngomo. LSQ: The linked SPARQL queries dataset. In Proc. of ISWC 2015, pages 261–269. Springer, 2015.
[15] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In Proc. of ICDT’10, pp. 4–33, 2010.
[16] P. Wood. Query languages for graph databases, SIGMOD Record, 41(1) (2012): 50–60.
[17] X. Zhang and J. Van den. Bussche. On the satisfiability problem for SPARQL patterns. arXiv:1406.1404, 2014.
[18] X. Zhang and J. Van den. Bussche. On the primitivity of operators in SPARQL. Information Processing Letters, 114(9):480–485, 2014.