Abstract—Recent keyword search techniques on Semantic Web are moving away from shallow, information retrieval-style approaches that merely find “keyword matches” towards more interpretive approaches that attempt to induce structure from keyword queries. The process of query interpretation is usually guided by structures in data, and schema and is often supported by a graph exploration procedure. However, graph exploration-based interpretive techniques are impractical for multi-tenant scenarios for large databases because separate expensive graph exploration states need to be maintained for different user queries. This leads to significant memory overhead in situations of large numbers of concurrent requests. This limitation could negatively impact the possibility of achieving the ultimate goal of personalizing search. In this paper, we propose a light-weight interpretation approach that employs indexing to improve throughput and concurrency with much less memory overhead. It is also more amenable to distributed or partitioned execution. The approach is implemented in a system called “SKI” and an experimental evaluation of SKI’s performance on the DBPedia and Billion Triple Challenge datasets shows orders-of-magnitude performance improvement over existing techniques.

Keywords—Scalability; Concurrency; Keyword Query Interpretation; Personalization; Big RDF data; Semantic Web;

I. INTRODUCTION

There is increasing interest in exploiting the growing amount of structured, semantic data on the Web to improve the quality of keyword search. The idea is to be able to better identify a user’s intended semantics for a query so as to provide the appropriate set of results. To achieve this, techniques must go beyond the IR or “meaning as match” interpretation styles used by traditional search engines. Most techniques proposed for keyword queries over structured/semi-structured data are also based on such interpretation styles with relational databases ([1], [2], [3], [4]) and XML databases ([5]). The limitation of these approaches are well known: i) result lists that are semantically incongruent so that users have to manually filter out results that are irrelevant to the user’s querying context; ii) inability to deal with moderately complex queries. One example is the class of “list queries” that describes a set of entities rather than a specific entity. For example, the query “patents big data management” is likely intended to return a list of patents on topics related to big data management. For example, a query for “patent big data management” may miss patents that do not contain the phrase “big data management” but contain keywords like “scalable”, “MapReduce” and so on. Processing such queries with current IR techniques presents challenges due to the length of such queries. In addition, some keywords in the queries are descriptive, which should not be considered as data explicitly associated with relevant results but as metadata.

To address this limitation, semantic search techniques have been proposed ([6], [7], [8]) where an “interpretation” phase is inserted prior to query evaluation. The goal of interpretation is to map a keyword query to a structured query, thereby giving the query a fixed semantics for which semantically congruent results can then be computed. However, identifying a unique semantics is often difficult because the terse nature of keyword queries makes them ambiguous. Consequently, most techniques address this as a top-K problem producing a list of K structured queries representing the K most likely intended semantics for a given keyword query. The structurization process involves computing connected subgraphs of keyword hits on a summarized representation of schema and data graphs. These graphs are mapped to structured queries by distinguishing the roles of keywords in a query i.e. metadata keywords vs. data keywords. For example, for the query “conferences tutorial big data” depending on the roles assigned to the word “conferences”, the interpretation could be “the list of conferences with tutorials on big data” or could be the “list of tutorials at conferences on big data”.

Heuristics for ranking connected subgraphs typically assign the intended semantics of a query to the probability of the query, defined in terms of the frequency or occurrence of subgraph structures. This is similar in spirit to search engines determining most relevant results based on click patterns in query and click-through logs. However, such techniques are impersonal and fail to customize interpretation of a query to an individual user’s needs. As an example, while the most frequent intention associated with the query “magic sets” may be toys because the shopping context is very general and has a large population of users. However, the same query could be intended to mean the query optimization technique which is well known but in a much smaller community. Unfortunately, such a relatively unpopular context would not be considered even if intended. It is important to note that this is not simply a question of profiling users and determining interests of a user because users have different interests and interact with the Web in different contexts. For example, a database researcher issuing the query “magic sets” may intend the context of toys when shopping for some gifts. Another example is “Jaguar Speed”; a computer science researcher may be investigating the supercomputer, then at a different time be buying a car, and yet another time be helping a middle school child with a science homework. Therefore, the ideal situation is to personalize query interpretation by identifying the intended interpretation for a specific user at a specific time.
In our previous work ([6], [7]), we addressed the problem of “personalizing” keyword query interpretation by capturing and exploiting a user’s “ambient” query context in terms of the queries preceding a query being considered. The underlying hypothesis is that users often pose multiple queries related to a specific need or task so that earlier queries can provide some contextual basis for future queries. The approach proposed modeled this context-aware query interpretation as computing top-K connected subgraphs over a context-aware summary graph. The context-aware summary graph was modeled as a dynamic weighted graph model in which keywords in a query ascribe weights transfer summary graph and the impact. The result is that for a given query being issued at a given time, connection subgraphs involving concepts and relationships with lower costs (higher weights) are ranked higher in the top-K list (decay model). A key challenge addressed in previous work is the indexing and management of the dynamic weight graph model and efficient algorithm for computing the Steiner Trees.

Motivation. Like the non-personalized techniques for query interpretation, our earlier proposed context-aware interpretation technique is heavy-weight because it is based on graph exploration on memory resident summary graphs. For big data, the number of hits per keywords and the degree of connectivity between nodes rises, increasing the exploration search space for connection subgraphs and thus memory requirements. For example, the keyword “conference” has 19 occurrences in DBPedia 3.6\(^1\), which includes 272 concept nodes and around 3000 properties (edges). Further, it has 1785 hits in the Billion Triple Challenge datasets (BTC 2009)\(^2\), which contains 150232 class nodes and around 1.3 million edges (BTC is a superset of DBpedia). In personalized query interpretation, each user has a different dynamic weighted graph because of differences in their query contexts. This could imply a separate graph exploration process for each user. For example, the memory requirements for interpreting a single query on the BTC dataset is around 500MB, which leads that a server with 8GB RAM can handle only 17 user queries concurrently. Consequently, we are faced with the limitation of only being able to meet the goal of personalized and contextual search under low concurrency/throughput, which is impractical for real multiserver search environments.

Contributions and Scope. In this paper, we focus on increasing the practicability of personalized semantic search in big data contexts by introducing lightweight interpretation techniques. Specifically, we propose an approach called “SKI that uses a dual indexing scheme to maximize the concurrency and throughput of the query interpretation process while minimizing the latency.SKI indexes user-specific query context information separately from the more general user-independent information about concepts and their relationships. The former is captured in an index – personalized query context map (PCM) and the latter in dense path index (DPI) – an index of subgraph structures. The data-specific indexes inform a graph exploration-free interpretation algorithm’s (GeneFree) decisions about which substructures to prune and how to assemble substructures into complete interpretations. GeneFree avoids the need for graph exploration and is fast and memory-efficient reducing both latency and memory requirements of query interpretation.

Impact. We posit that integrating our proposed scalable query interpretation approach with many ongoing-efforts on scalable query-answering techniques, will result in scalable Semantic Web search architectures.

The rest of the paper is organized as follows: Section II includes the preliminaries of the keyword query interpretations and the problem definition. Section III describes the overview of the SKI architecture and the details of the index structures and algorithms. Section IV and V present performance evaluation results and related work.

II. Problem Definition

Let \(\Sigma\) be a universe of words. An RDF schema graph is a labeled graph \(G_S = (V_S, E_S, \lambda_S)\) where the nodes represents classes and edges represent properties. \(\lambda_S\) is a labeling function that maps a graph element (node or edge) to \(l \subseteq \Sigma\) that contains words that make up the label of the class or property. An RDF data graph is also a labeled graph \(G_D = (V_D, E_D, \lambda_D, \pi_D)\) that has \(V_D, E_D\) and \(\lambda_D\) are similarly defined with \(\lambda_D\) mapping a graph element to set of words contained in the string literal associated with that element. \(\pi_D\) maps each data graph element (node or edge) \(g_D\) to a set of schema graph elements: those to which \(g_D\) is connected by a “\(rdf:\text{type}\)” relation.

Given a schema graph \(G_S\) and data graph \(G_D\), an annotated schema graph \(G_A\) is a tuple \((G_S, \lambda_A)\) where \(\lambda_A\) is a labeling function that maps each graph element \(g_S\) such that \(\lambda_A(g_S) = \{ \lambda_S(g_S) \cup \lambda_D(g_D) \mid g_S \in \pi_D(g_D) \}\). In other words, the function \(\lambda_A\) lifts labels on elements of data graph to the schema elements that they are instances of, thereby annotating the schema graph with keywords from the data graph.

A keyword query \(Q\) is a set of words \(\{k_1, k_2, \ldots, k_{|Q|}\}\) from the universe \(\Sigma\). An element \(g_t\) of an annotated schema graph \(G_A\) is called a hit for keyword \(k\) if \(k \in \lambda_A(g_t)\). The set of all hits for a keyword \(k\) is denoted as \(hit(k)\). An interpretation \(|Q|\) of a keyword query \(Q\) is a subtree of \(G_A\), that contains one hit for each keyword in \(Q\). Since there are potentially multiple interpretations of a keyword query (due to different possible combinations of hits for all keywords), we focus on a specific (top) interpretation for \(Q\) which we denote \(|Q|*\), but ignore for now, how to choose \(|Q|*\) from the set of alternatives for \(Q\).

**DEFINITION 2.1:** A context-aware annotated graph or CAG for short, associates weights with an annotated schema graph based on the sequence of queries that have been posed on that graph. A sequence of queries \(Q_1, \ldots, Q_m\) produces a sequence of weighted graphs \(G_0 \rightarrow G_1 \ldots \rightarrow G_m\) where \(G_t (0 \leq t \leq m)\) is the weighted annotated schema graph produced after \(Q_t\). \(G_t\) is defined as \((G_A, \omega_t)\) where \(\omega_t \in \mathbb{F}\) is a family of weighting functions s.t:

- \(\omega_0 \in \mathbb{F}\) is an init. function that assigns weights to all elements of the annotated schema graph prior to \(Q_1\);
- \(\omega_t \in \mathbb{F}\) is defined s.f. for \(g \in |Q|*\), i.e. in the top interpretation for \(Q_t\), \(\omega_t(g) = \theta(\omega_{t-1}(g))\).

In other words, after \(Q_t\), the weights of a graph elements in the top interpretation of \(Q_t\), is a function of their weights after

\(^1\)http://blog.dbpedia.org/2011/01/17/dbpedia-36-released/
\(^2\)http://km.aifb.kit.edu/projects/btc-2009/
$Q_{i-1}$). This leads to the definition of an annotated schema graph as a dynamically weighted graph whose elements’ weights change with queries.

As mentioned earlier, the details of how to select $\|Q\|_*$ – the top interpretation for a keyword query $Q$, are omitted. However, this can be thought of as the lowest weighted interpretation when the weights of all graph elements in that interpretation are combined. The combination cost function and $\theta$ are discussed in [6], but we illustrate the concept of a CAG with an example.

**Context-Aware Interpretation Example** Fig. 1(a) shows the example CAG with some classes and literal nodes which are annotated with literals from the data graph. Given a keyword “Mississippi River Bank”, we find matches in this graph and select the “highest” weighted connecting subtree as the “top” interpretation. In Fig. 1(a), the graph has been initialized with some weights, e.g., the class nodes “Mississippi River Bank” and “Bank” are initialized with the weight 1.0 and 2.0 respectively. For example, the green subtree involving the class nodes Mississippi River Bank and Bank represents the interpretation of the query as the bank of the Mississippi River while the purple subtree involving nodes Mississippi River and River represent the interpretation the Mississippi River. However, based on the weight assignment, it is difficult to determine which interpretation should be selected as a highest weighted one since the total weight of subgraphs are the same, i.e. 3.0 and 3.0 for both subtrees. Note that the method for computing the initial weights can vary and is not our primary concern here. We calculate the total weights as the sum of the weights of nodes and edges, not the sum of the weights of all path represented in the subgraph.

Fig. 1(b-d) show how context-aware interpretations are processed with two paths originating from the initial graph. Each path represents a sequence of queries performed by a user: the top path ($Q_1$ and $Q_2$) by the user $X$ and the bottom path ($Q_3$ and $Q_4$) by the user $Y$. In the top path, Fig. 1(b) shows that the keyword query $Q_1$ “Mortgage Rate” results in increasing the weight of the nodes Mortgage Loan and Rate from 1.0 to 2.0, and the weight of its neighboring node Bank from 2.0 to 2.5. These changes make the top interpretation of the keyword query $Q_2$ becomes the subtree consisting of the nodes Mississippi River Bank and Bank in Fig. 1(c) because the sum of the weights of those nodes are the highest one, i.e. 2.5 + 1.0 = 3.5. Similarly, in the bottom path, the weight of the node River is increased from 1.0 to 1.5 because of the keyword query $Q_3$ “Fishing Activity” in Fig. 1(d). This increment leads to the subtree with the nodes Mississippi River, River, and River Bank as the top interpretation of the subsequent query $Q_4$ because its weight now becomes the highest one, i.e., 1.0 + 1.5 + 1.0 = 3.5. Note that $Q_2$ from the user $X$ and $Q_4$ from the user $Y$ are the same queries but produce different top interpretations.

Although our example illustrates the changes in weights as increasing weights, the real situation is the inverse (higher weight leads to lower cost). Since we are solving this as a translation of the Steiner Tree problem which is a minimization problem, the “top” interpretations should actually be the smallest-weighted ones. Consequently, our concept of increasing weights in example to reflect importance, is actually done as decreasing weights.

The problem we address in this paper is supporting personalized CAGs, i.e. for two users $X$ and $Y$ with query sequences $Q_SX$ and $Q_SY$ resp., $CAG_X$ and $CAG_Y$ are two (possibly different) CAGs associated with $X$ and $Y$ resp. The conceptual model of CAGs extends very naturally to multiple users. However, the key challenge is how to scale up the number of concurrent users, since a different CAG has to be managed for each user? Particularly, in the presence of big annotated schema graphs. The following section discusses our approach.

### III. THE SKI APPROACH

#### A. Architecture Overview

In this section, we present the SKI strategy for scalable personalized query interpretation that consists of three main components:
1) a distributed master-slave architecture for scalability
2) an index for user query-context information called a personalized query context map (PCM), which is separate from a more heavy-weight index of user-independent information about graph structures, i.e., concepts and relationships in the schema graph 3 captured in a couple of other indexes. After each query has been interpreted, PCM is updated to reflect this new information about query context i.e. \( \omega_i^{U_1} \rightarrow \omega_i^{U_{i+1}} \) for user \( U_i \). To avoid the overhead of maintaining all dynamic weighted graphs for supporting multiple concurrent interpretations, PCM maintains only node-weight maps for each user.
3) an algorithm GeFree for computing top-k connection subgraphs for each user query in which user-independent indexes (i.e., dense path index, DPI) are shared across interpretation instances for each user and are then combined with their user-specific context information in the PCM. GeFree is graph exploration free making it more efficient and allowing higher throughput and the sharing of user-independent indexes minimizes amount of memory required for interpretation to allow increased concurrency.
4) form the query interpretation layer which is interfaced with the query answering layer that uses standard RDF query processing engines such as SESAME.

In this paper, we focus on the interpretation layer. The slave nodes are grouped into clusters, or interpretation clusters for and the master node is responsible for scheduling user query requests to interpretation clusters and exchanging information such as resulting interpretations and query answers between users and interpretation clusters. Each interpretation cluster is responsible for processing a specific group of users’ query requests. Indexes are replicated across all slave nodes with one exception where, a partitioning of PCM in an interpretation cluster contains only the context map for the group of users maintained by this cluster, it is replicated across slave nodes within this cluster. Each slave node in an interpretation cluster runs a separate thread for each interpretation instance. Each interpretation instance has 3 key steps depicted in Fig. 2:

1) Step 1.1 Initialize Keywords Hits: by looking up the KS-Map (inverted index) that maps keywords to schema graph elements;
2) Step 1.2 Identify Candidate Roots: identifies all possible roots of candidate connecting trees linking at least one keyword hit for all keywords in a query. This process is supported by Rabbit Index: a group reachability index for rapid identification of root nodes of connecting trees;
3) Step 1.3 Top-K Graph Patterns Generation: uses the GeFree algorithm to assemble and generate top-K graph patterns representing top-K interpretations based on candidate roots computed in 1.2. GeFree is a “graph exploration free” algorithm that uses DPI for fast construction of interpretations. It uses materialized path information stored in DPI and Rabbit to enable rapid assembly of the top-K minimum connecting trees representing the top-k interpretations. To personalize the interpretation process, it uses the user’s information in the PCM partition assigned to that slave node.

Fig. 2. Architecture and workflow for a single slave node

After the personalized top-K candidate graph patterns are generated, Steps 2 to 4 take care of sending them to the user layer and sending the highest ranked queries to the query answering layer for processing. Since the ranking is based on heuristics and may not be perfect, sending the top-K list of graph patterns allows a user to select a different top-1 pattern if the highest ranked pattern is not their intended interpretation. When this happens, the processing of the originally dispatched top-1 query is halted and query processing for the newly selected graph pattern begins.

Elaborating on the algorithms requires the introduction of some concepts: \( \Delta \)-path covers and \( \Delta \)-graph exploration graphs which define structural information maintained by DPI and GeFree. Candidate root is a key concepts for Rabbit index and GeFree.

B. Keyword Query Interpretation in SKI

1) Foundations:

**DEFINITION 3.1:** A \( \Delta \)-path cover of a node \( r \) in graph \( G \) is denoted by \( \pi(\Delta, r, G) = \{p_i\} \), where \( p_i \) is a path such that \( \forall p_i, p_i \in \pi(\Delta, r, G) \) if and only if \( r \) in \( G \) is the source node of \( p_i \) and \( |p_i| \leq \Delta \), i.e. the path cover \( \pi(\Delta, r, G) \) contains all reachable paths of length less than or equal to \( \Delta \) from \( r \). For example, a 2-path cover of node \( D \) in the graph shown in Fig. 3 is \( \{D, D \rightarrow A, D \rightarrow B, D \rightarrow E, D \rightarrow X, D \rightarrow X \rightarrow N, D \rightarrow X \rightarrow I, D \rightarrow H \rightarrow M, D \rightarrow H \rightarrow I, D \rightarrow H \rightarrow E, D \rightarrow E \rightarrow H, D \rightarrow E \rightarrow I, D \rightarrow E \rightarrow F\} \).

**DEFINITION 3.2:** A \( \Delta \)-Exploration Graph of a node \( r \) in graph \( G \) (denoted by \( g_2^\Delta \)) is a rooted subgraph of \( G \), which contains all edges in \( \pi(\Delta, r, G) \). For example in Fig. 3, let \( G_S \) be the whole graph, the subgraph in the box is a 2-exploration graph rooted at node \( Z \) : \( g_2^\Delta \), which contains all edges in \( \pi(2, Z, G_S) = \{Z \rightarrow Q, Z \rightarrow Q \rightarrow M, Z \rightarrow Q \rightarrow S, Z \rightarrow Q \rightarrow R\} \).

**DEFINITION 3.3:** Given a keyword query \( Q \) and a schema graph \( G_S \), a candidate root of \( Q \) is the root node \( r \) of a \( \Delta \)-exploration graph \( g_2^\Delta \), which contains at least one hit of each
keyword in $Q$. For example, in Fig. 3, $I$ is a candidate root for keyword query $k1, k2, k3$, because the 2-exploration graph $g^2_i$ contains a hit $P$ (matches $k1$), $K$ (matches $k2$) and $Y$ (matches $k3$). Those hits (such as $P, K, Y$ in $g^2_i$) that are covered by the $\Delta$-exploration graph $g^\Delta_i$ rooted at a candidate root $r$ are called productive hits. Productive hits can always reach a candidate root following a path of length less than or equal to $\Delta$.

2) Identifying Candidate Roots: Different from the graph exploration based approaches, where roots are identified by exploring paths originated from each hits, in this section, we introduce the idea of fast candidate roots and productive hits identification.

Group Reachability Bitmap Index. Given a keyword query $Q = \{k_i\}$, in order to identify a group of candidate roots, a “group reachability” question that needs to be answered is: Given a set of groups of nodes, whether every node in a group of nodes $CR$ can reach at least one node in ALL the other groups within $\Delta$ hops. We propose a group reachability bitmap index (Rabit) to identify candidate roots and productive hits more efficiently. For each node $u_i$ in the graph, we propose to use a bit vector denoted by $\beta_i = b_1 b_2 \ldots b_{|V(G)|}$ to represent the nodes that can reach $u_i$ within $\Delta$ hops. Notice that, in Fig. 3, each bit vector for each node is 26 bits long because there are 26 nodes in that graph, which can then be represented using a 32-bit integer, such that we can use one integer to represent the $\Delta$ exploration graph for each node. Using the Rabbit index, we can identify for each group of keyword hits, a set of neighboring nodes that can reach at least one hit in that group by using bitwise OR operation. Then, we can identify a bit vector that represents the common nodes in all groups of neighboring nodes. This can be achieved by bitwise AND operation, and those nodes are candidate roots. Those hits that can NOT reach any candidate roots are unproductive. Those unproductive hits can be identified using bitwise AND operation.

3) GeFree: A Graph Exploration Free Approach: After identifying candidate roots, we present another important index called dense path index (DPI). Having unproductive hits pruned, given a set of candidate roots $CR = \{r_i\}$, for keyword query $Q$, and each root $r_i$, there are many possible interpretations rooted at $r_i$. However, only knowing the roots is not enough to identify the top-K interpretations among all possible interpretations that rooted at each root in $CR$. The paths from a root node to all hits need to be computed and assembled to construct an interpretation.

Dense Path Index. The dense path index is a two-level hierarchical index. The first layer is called node-exploration graph map, or $NE$ map, which is organized as a hash table that maps any node $r$ in $G_S$ to a $\Delta$-exploration graph $g^\Delta_r$. $NE(r) = g^\Delta_r$ denotes such mapping. The second layer of the dense path index is called destination-path map, or $DP$ map that maps a destination node to a set of paths: For each $\Delta$-exploration graph, the $\Delta$-path cover of $r$, i.e., $\pi(\Delta, r, g^\Delta_r)$ is pre-computed and those paths in $\pi(\Delta, r, g^\Delta_r)$ are grouped by destination nodes $\pi(\Delta, r, g^\Delta_r) = \{P_{v_0} \cup P_{v_1} \cup \ldots \cup P_{v_m}\}$, where $P_v$ is called a path group of $v$, where $v \in V(g^\Delta_r)$ is a destination node, and $\forall i, j, P_{v_i} \cap P_{v_j} = \phi$. $DP(v, g^\Delta_r) = P_v$ denotes the mapping for $g^\Delta_r$ from a destination node $v$ to the path group $P_v$ of $v$. The $\Delta$-path cover can be organized as a hash table that maps any destination node $v_j \in V(g^\Delta_r)$ to the path group of $v_j$, i.e., $P_{v_j}$.

For each root-keyword pair, they are updated and stored in PCM after each query generation and retrieved from PCM during query time at most the top-K minimum weighted paths are necessary for computing top-K interpretations. For example, given a keyword query $Q = < w_1, w_2 >$, considering a candidate root $r$, assuming that we are only interested in top-2 interpretations, let $P_1 = < p_{11}, p_{12}, p_{13} >$ be the top-3 paths from $r$ to any hits of $w_1$; $P_2 = < p_{21}, p_{22} >$ be the top-2 paths from $r$ to any hits of $w_2$, any combinations of paths including $p_{11}$ will not form a top-2 interpretation and should not be investigated during the processing of generating top-K interpretations. The GeFree algorithm is illustrated in Algorithm 1.

Algorithm 1 GeFree

1: Input: $Q = \{w_i\}, CR = \{r_i\}, PKE = \{u_i\}.$
2: $TOPK = \phi$ is a priority queue for maintaining top-K interpretations.
3: for all $w_i \in Q$ do
4: $H[w_i] = hit(w_i) \cap PKE;$
5: end for
6: for all $r_i \in CR$ do
7: $g^\Delta_r = NE(r_i);$ 
8: for all $w_i \in Q$ do
9: $PQ_j; // Initialize a priority queue for maintaining top-K paths$
10: for all Hit $v_{jk} \in H[w_i]$ do
11: $PG = DP(v_{jk}, g^\Delta_r);$ 
12: Calculate $P_{k} = TopK paths in PG as a heap;$
13: $P_k \rightarrow PQ_j;$
14: Sorted Path Group $S_j = PQ_j, HeapSort();$
15: end for
16: end for
17: $TOPK = TopKCombinations(\{S_j\});$
18: end for
19: return $TOPK$

GeFree Algorithm. For each keyword $w_i$, only productive hits are left and stored in $H[w_i]$ for $w_i$ (line 4). For each candidate root $r_i$, the corresponding $\Delta$-exploration graph is returned using $NE$ map (line 7). Then, for each keyword $w_i$, $H[w_i]$ contains a set of productive hits for $w_i$; for each productive hit $v_{jk}$, $DP$ map returns a path group $P_k$ (line 11). For a candidate root $r_i$ and a keyword $w_j$, Top-K root-keyword paths are computed by merging top-K-heaps from each path groups of all productive hits of $w_j$ into a priority queue $PQ_j$ of size $K$ (line 12 and line 13). In line 14, for the specific root $r_i$, a sorted list $S_j$ containing $K$ paths is computed for each keyword. $\{Q\}$ lists are computed in total. $TopKCombinations(\{S_j\})$ is an algorithm to quickly generate top-K combinations of paths from $|Q|$ sorted path
lists. The algorithm is based on the TopKCombinations algorithm proposed in [6]. This algorithm suggests an early termination strategy such that top-K combinations will be generated without enumerating all combinations of paths and has guaranteed \( O(k^n) \) time complexity. The time complexity of GeFree is \( O(CR((K^n + \sum_{i=1}^{m} PKE) \lg(K))) \), where \( CR \) is the set of candidate roots, and \( PKE \) is the set of all productive hits, \( K \) is the number of top-K interpretations generated. Because in the worst case, \( CR \) may contain all nodes in the graph and \( PKE \) is equal to a set of all possible hits, the worst case time complexity is \( O(n(k^n + \sum_{i=1}^{m} \frac{PKE}{\lg(K)})) \), where \( n = |V(G_s)| \) and \( m = \sum_{i=1}^{m} |hit(u_i)| \) is the total number of hits. In comparison, the time complexity for graph exploration based algorithm as reported in CoSi [6] is \( O(n d^2 K^n) \), where \( d \) is the degree of the graph.

### IV. Evaluation

We evaluated the scalability of the proposed multi-tenant query interpretation system (SKI) by comparing its performance with other systems (CoSi [6] and TKQ2S [8]). In this evaluation, we mainly focused on measuring the efficiency improvement and scalability in terms of concurrency because the effectiveness of the personalized techniques proposed in CoSi has been studied in our prior work [6].

**Datasets and Testbed:** Two real-life datasets were employed: (a) DBPedia 3.6 (approx. 300 classes, 3K property types, 3.5 million entities – 678 million triples) and (b) the “Billion Triple Challenge 2009” or BTC (1.19 billion triples). For the BTC, we created the schema graph from its dataset, which includes 150232 classes (nodes) and around 1.3M properties (edges). For each experiment, we first initialized random weights and used the same set of initial weights for all personalized algorithms to represent the bootstrapping context of user query history. The evaluation was conducted on a cluster of up to 30 slave nodes, each of which is equipped with Intel dual core 2.0 GHz CPU and 8GB RAM. All algorithms were implemented with C#, and all results were averaged over at least 5 trials.

**Tasks and Metrics:** We designed two tasks to evaluate the performance of each approach. The first task measured the performance metrics such as execution time and DOTA for each query and dataset. We introduced DOTA or Degree Of Term Ambiguity in our previous work [6] that characterizes the ambiguity of a query as a reflection of how much work (i.e., exploration search space) is needed for its interpretation. DOTA is defined as a function of the number of combinations of all keyword hits. The second task measured the latency of query responses while varying the MPL (Multi-Programming Level), i.e., the number of concurrent running programs.

**Queries:** We employed 8 real life queries from Semantic Search Challenge 2011$^4$. The additional details can be found in the project page$^5$.

#### A. Task 1. Measuring Execution Time and DOTA

We recorded the execution time and DOTA of the 8 queries (Q1-Q8) in the query set. Fig. 4(a) shows that SKI generally outperformed CoSi for both datasets. Fig. 4(b) shows that the queries tend to have much higher DOTA in the BTC dataset than in DBPedia because all keywords appeared in both datasets but with varying frequencies because BTC was a superset of the DBPedia dataset. (Note that the queries are ordered by the DOTAs of queries over DBPedia, not BTC.) The performances of both CoSi and SKI dropped when interpreting the same queries on a dataset with larger scale since the degree of ambiguity increases for the same keyword query. For queries Q6 to Q8, the DOTA increased significantly; SKI performed much stable on both BTC and DBPedia while the performance of CoSi dropped considerably because the search space increased significantly with the growth of DOTA. In particular, the DOTA of Q2 over BTC was larger than the DOTAs of Q1 and Q3; therefore, the time spent by CoSi for Q2 on BTC was higher than that for Q1 and Q3.


In this task, we increased the MPL and measured the corresponding response latency of concurrently running queries to compare the degree of concurrency. Due to the resource contention between multiple programs, the performance and memory consumption of the personalized interpretation algorithm impact the response latency of concurrently running queries for different users. Fig. 4(c) shows the impact of MPL on the latency (in milliseconds) for interpretation algorithms. Although all approaches experience increased latency as MPL increases, the rate of increase is higher with the exploration-based approaches than the index approach of SKI. This suggests that the concurrency overhead for CoSi and TKQ2S is higher than SKI. Note that SKI supports much higher multi-programming level while keeping low latency. CoSi or TKQ2S were not able to maintain reasonable latency for MPL greater than 5 due to high memory consumption and CPU overhead. As an explanation for this behavior, we report the amount of memory consumption of SKI, CoSi and TKQ2S in Table I for each user. Note that the reported memory usage of graph for graph exploration-based algorithms includes the overhead of maintaining intermediate graph exploration states, e.g., paths explored and cursors generated. The column EstMaxU shows the estimation of the maximum number of user queries that can be running concurrently based on the memory consumption of each approach. In case of BTC dataset, each user in CoSi/TKQ2S requires around 445MB for maintaining graph exploration states associated with the summarized schema graph while the SKI consumes only 2.8MB for PCM and other indexes for each user. Thus, CoSi/TKQ2S can only support up to 17 users using 8GB memory, but SKI can handle up to 2857 users concurrently.

Additionally, we conducted experiments on the interpretation slave clusters (SC) with SKI deployed on them and report the throughput in Table II. Six groups of experiments were conducted with varying sizes of an interpretation cluster (up to 17 nodes). The results show that the performance improvement is more significant for smaller clusters, but the scalability is acceptable even for large clusters.

<table>
<thead>
<tr>
<th>Data/Appr.</th>
<th>Graph</th>
<th>PCM</th>
<th>KS-MAP</th>
<th>DPT</th>
<th>RABIT</th>
<th>EstMaxU</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTC-C/T</td>
<td>337</td>
<td>2.8</td>
<td>221.94</td>
<td>155.66</td>
<td>25.6</td>
<td>2857</td>
</tr>
<tr>
<td>BTC-C/20</td>
<td>445</td>
<td>3</td>
<td>38.83</td>
<td>6.75</td>
<td>1.33</td>
<td>114256</td>
</tr>
<tr>
<td>DBPedia-S</td>
<td>1.1</td>
<td>0.07</td>
<td>107.57</td>
<td>445</td>
<td>0.07</td>
<td>178.43</td>
</tr>
<tr>
<td>DBPedia-C/T</td>
<td>1.4</td>
<td>2.4</td>
<td>54.67</td>
<td>2.8</td>
<td>1.4</td>
<td>5714</td>
</tr>
</tbody>
</table>

$^4$https://km.aifb.kit.edu/ws/semsearch11/
$^5$http://research.csc.ncsu.edu/coul/SKI/experiments.xlsx

---

**TABLE I. MEMORY CONSUMPTION AND CONCURRENCY (MB) – SKI(S), CoSi/TKQ2S(C/T)**

<table>
<thead>
<tr>
<th>Data/Appr.</th>
<th>SC5</th>
<th>SC10</th>
<th>SC15</th>
<th>SC20</th>
<th>SC25</th>
<th>SC30</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTC-C/T</td>
<td>1000</td>
<td>54.47</td>
<td>78.88</td>
<td>89.24</td>
<td>107.57</td>
<td>150.27</td>
</tr>
<tr>
<td>BTC-C/20</td>
<td>5000</td>
<td>81.92</td>
<td>129.09</td>
<td>178.43</td>
<td>229.45</td>
<td>258.85</td>
</tr>
<tr>
<td>DBPedia-S</td>
<td>5000</td>
<td>80.45</td>
<td>152.59</td>
<td>215.47</td>
<td>268.22</td>
<td>307.46</td>
</tr>
</tbody>
</table>

**TABLE II. THROUGHPUT (QUERIES PER SECOND)**

<table>
<thead>
<tr>
<th>Data/Appr.</th>
<th>SC5</th>
<th>SC10</th>
<th>SC15</th>
<th>SC20</th>
<th>SC25</th>
<th>SC30</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTC-C/T</td>
<td>1000</td>
<td>54.47</td>
<td>78.88</td>
<td>89.24</td>
<td>107.57</td>
<td>150.27</td>
</tr>
<tr>
<td>BTC-C/20</td>
<td>5000</td>
<td>81.92</td>
<td>129.09</td>
<td>178.43</td>
<td>229.45</td>
<td>258.85</td>
</tr>
<tr>
<td>DBPedia-S</td>
<td>5000</td>
<td>80.45</td>
<td>152.59</td>
<td>215.47</td>
<td>268.22</td>
<td>307.46</td>
</tr>
</tbody>
</table>
Fig. 4. (a) The execution time of the three approaches (SKI, CoSi, and TKQ2S) and (b) The DOTAs for the query set QS, (c) The latency of the three approaches while varying the MPL.

to 30 slave nodes) and different sizes of the query mixes (1k to 10k queries). The throughput (queries per second) of the larger cluster was higher.

V. RELATED WORK

Personalized search has been extensively studied for IR styled search paradigm [9][10][11]. For semantic search, in some previous work COSI [6][7], we proposed techniques for modeling ambient search context and generating context-aware query interpretations. Yi et al., [12] proposed a “refill” based view selection algorithm for efficiently maintaining dynamic top-K views. In our case, to efficiently maintain dynamic top-K paths, we introduced indexing scheme in SKI to overcome the overhead of expensive graph exploration.

Graph exploration is a key component in most keyword search techniques. They can be categorized by i) Distinct Root Semantics (DRS), where every answer tree has a root node and the root-hit path in the answer tree is the shortest. BLINKS [2] and DPBF [13] are typical DRS based approaches. ii). Steiner Tree Semantics (STS). Compared with DRS, under STS, an answer to a keyword query is also a rooted tree but the root-hit path is not necessary to be the shortest. STAR [14] presents an efficient algorithm to identify top-k minimum Steiner trees. However, they assume that each keyword matches a single node in the graph. For Group Steiner Tree problem, where each keyword has multiple matches, different types of graph traversal algorithms are proposed such as BANKS [1] and BANKS-II [15]. iii). Sub-Graph Semantics (SGS). Instead of tree representation of answers, under SGS, some approaches propose to return sub-graphs as answers. EASE [16] defines an r-radius Steiner graph. Ladwig et al., [17], argues that assuming a center of an answer graph restricts the search space. They introduce a d-length Steiner graph semantics.

As to approaches that do not require graph exploration, BLINKS [2] proposes a similar idea to pre-compute shortest search paths. However, we propose to save and manage richer path information (not only shortest paths) and generate answers with Steiner tree semantics (larger search space). In addition, BLINKS maintains mappings from every keyword to a ranked list of nodes ordered by weights; EASE [16] proposes Ef-index, where ordered weighted r-radius graphs for pair-wise tokens are pre-computed. However, to achieve personalized search, after every query, weights will need to be changed to reflect the evolving context, the ordering will also need to be updated. For existing techniques this may require a complete rebuild after each query, which is impractical.

VI. CONCLUSION

This paper presents an approach for “interpretive” search that scales well under high concurrent workloads. The approach achieves this by relying on a suitable set of indexes rather than expensive graph exploration based algorithms. Comprehensive evaluation over real world datasets showed very promising results.

VII. ACKNOWLEDGMENT

This work was partially funded by NSF grant IIS-0915865.

REFERENCES