Index Structures and Top-k Join Algorithms for Native Keyword Search Databases

Günter Ladwig  
Institute AIFB, Karlsruhe Institute of Technology  
D-76128 Karlsruhe, Germany  
guenter.ladwig@kit.edu

Thanh Tran  
Institute AIFB, Karlsruhe Institute of Technology  
D-76128 Karlsruhe, Germany  
duchthanh.tran@kit.edu

ABSTRACT

For supporting keyword search on structured data, current solutions require large indexes to be built that redundantly store subgraphs called neighborhoods. Further, for exploring keyword search results, large graphs have to be loaded into memory. We propose a solution, which employs much more compact index structures for neighborhood lookups. Using these indexes, we reduce keyword search result exploration to the traditional database problem of top-k join processing, enabling results to be computed efficiently. In particular, this computation can be performed on data streams successively loaded from disk (i.e., does not require the entire input to be loaded at once into memory). For supporting this, we propose a top-k procedure based on the rank join operator, which not only computes the k-best results, but also selects query plans in a top-k fashion during the process. In experiments using large real-world datasets, our solution reduced storage requirements and also outperformed the state-of-the-art in terms of performance and scalability.

1. INTRODUCTION

Today, we find on the Web not only textual documents but also a large amount of structured data. This includes metadata associated with Web pages, semi-structured data, and highly-structured information made publicly available such as Linked Data\(^1\).

Especially for lay users, keyword search has been regarded as an effective mechanism because it helps to circumvent the complexity of structured query languages, and hide the underlying data representation. Without knowledge of the underlying database engine can be fully utilized. On the other hand, there are schema-agnostic approaches ([10, 5, 12]), which operate directly on the data. Since they do not rely on a schema, the applicability of these approaches is not limited to structured data. For instance, semi-structured RDF data [18] as well as the combination of structured, semi-structured and unstructured data [12] have been supported.

In this paper, we focus on the latter type, also called native approaches. Fig. 1 shows a data example, describing people \((p1 – p4)\) and their relationships. For computing results of a query such as \(Q_1 = \{miller, corp\}\), a native approach firstly finds matching elements in the data graph, which are \(miller = \{p1, p4\}\) and \(corp = \{o1, o2\}\). Next, structures connecting these so-called keyword elements are explored. These structures constitute the query results, which are called Steiner trees [3], or Steiner graphs ([18, 12]).

Since the search space may be large, two main directions have been investigated for optimization. One is based on materializing paths [5]. The fastest solution proposed recently, called EASE, is based on materializing even more complex structures in the form of graphs, which capture entire neighborhoods of data elements [12]. The other direction is to improve the efficiency of the exploration algorithm ([10, 3]). The main drawbacks of current state-of-the-art is that materialized indexes [12], which redundantly store neighborhood information may become too large. Further, existing search strategies for efficient result exploration assume the entire data to be available in memory, a requirement that is too limited when the amount of data is large.

The index structures and algorithms proposed so far are very specific to the keyword search problem. In this work, we aim to reduce this problem to more standard database ones to leverage the vast body of existing work. In particular, we propose the concept of native keyword search databases, which solves the keyword search problem using two standard database operations, data access and join. Instead of loading the data entirely at once and exploring results in memory, keyword search is supported through a series of fine granular

\(\text{http://linkeddata.org/}\)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

\(\text{CIKM'11, October 24–28, 2011, Glasgow, Scotland, UK.}\)
\(\text{Copyright 2011 ACM 978-1-4503-0717-8/11/10 ...$10.00.}\)
We propose a new processing strategy for dealing with the keyword search problem that is based on the standard database operations data access and join.

- For efficient data access, we extend the 2-hop cover concept ([17, 1]) to pre-compute and materialize the neighborhoods of data elements. Existing work indexes every maximal neighborhood, i.e., when it is not completely covered by one another [12]. In this work, we determine coverage at the level of paths, instead of graphs. Indexing only maximal paths reduces the space requirements.

- For efficient and scalable Steiner graph search, we propose an extension of the hash rank join [8] called push-based top-k rank join to terminate early after the top-k results have been found. Instead of reading the entire graph into memory, this procedure operates on input streams.

- The main difference between join processing in keyword search and a standard database is that different query plans may lead to different, but also valid results. Instead of focusing on one single optimal plan, a large number of query plans have to be considered to generate all top-k answers. The push-based top-k rank join procedure proposed in this paper not only computes the best results according to one plan, but also selects plans in a top-k fashion during the process.

We evaluated our approach by comparing it to state-of-the-art native keyword search approaches ([12, 10]), using several large real-world datasets. Our solution reduced storage requirement (up to 80%) and improved scalability (by several factors) and performance (over 50% on average).

**Outline.** In Section 2 we present our data model and a formal definition of keyword search. Section 3 describes our extension of the 2-hop cover concept. In Section 4 we show how keyword queries can be answered using top-k join processing on the proposed data structures. Related work is presented in Section 5. In Section 6 we present the result of our evaluation. We conclude in Section 7.

2. OVERVIEW

This work deals with general graph structured data: data access (path lookups) and join operations (path and graph joins). For supporting these operations, we build upon existing work on index structures and top-k join processing.

The main contributions can be summarized as follows:

- For efficient data access, we extend the 2-hop cover concept ([17, 1]) to pre-compute and materialize the neighborhoods of data elements. Existing work indexes every maximal neighborhood, i.e., when it is not completely covered by one another [12]. In this work, we determine coverage at the level of paths, instead of graphs. Indexing only maximal paths reduces the space requirements.

- For efficient and scalable Steiner graph search, we propose an extension of the hash rank join [8] called push-based top-k rank join to terminate early after the top-k results have been found. Instead of reading the entire graph into memory, this procedure operates on input streams.

- The main difference between join processing in keyword search and a standard database is that different query plans may lead to different, but also valid results. Instead of focusing on one single optimal plan, a large number of query plans have to be considered to generate all top-k answers. The push-based top-k rank join procedure proposed in this paper not only computes the best results according to one plan, but also selects plans in a top-k fashion during the process.

We evaluated our approach by comparing it to state-of-the-art native keyword search approaches ([12, 10]), using several large real-world datasets. Our solution reduced storage requirement (up to 80%) and improved scalability (by several factors) and performance (over 50% on average).

**Definition 1 (Graph Structured Data).** Data is modeled as a directed, labeled graph $G(N, E)$ called data graph, where $N = N_F \cup N_A$ is a set of nodes, conceived as the disjoint union of entities $N_F$ and attribute values $N_A$, and $E = E_R \cup E_A$ is a set of directed edges, where $E_R$ are edges between entities called relations, and $E_A$ are edges between an entity and an attribute value called attributes. The length of a path denotes the number of edges in the path. We use distance between $u, w$ to denote the length of the shortest path between $u$ and $w$. Two graph elements (nodes or edges) are connected if they appear together in a path (no matter the directions of edges in that path).

In addition to graph-structured data such as RDF, this model can also be used to capture relational databases: a tuple captures an entity and its attribute values; and tuples with a foreign key relationship are connected via a relation edge.

Query keywords match entities’ attribute value nodes, or labels of attribute or relation edges. Thus, keyword matches can always be associated with some entities. We call these matching entities keyword elements:

**Definition 2 (Keyword Element).** Given a keyword query $K$ consisting of keywords $k_1, \ldots, k_n$, a node $n_k \in N_E$ is a keyword element for $k \in K$ if there is $e(n_k, a) \in E$ and $k$ matches any of the elements $e(n_k, a)$ or $a$ (i.e., $k$ matches edges that contain $n_k$ in the “subject” position). The set of all keyword elements for $k$ is $N_k$.

For processing a keyword query, we firstly find keyword elements in the data graph. Then, we search for substructures, which connect the keyword elements. The most commonly used substructures are Steiner trees [10], i.e., minimal rooted trees in the data graph, which contains at least one keyword element for every keyword in the query. Instead of rooted trees, general graphs have also been used [18, 12]. This notion of Steiner graph is also employed in this paper:

**Definition 3 (Keyword Query Answer).** An answer to a keyword query $K = \{k_1, \ldots, k_n\}$ also called Steiner graph is a subgraph of $G$ denoted as $G_S = (N_S, E_S)$, which satisfies the following conditions:

- For every $k \in K$ there is at least one keyword element $n_k \in N_S$. The set of keyword elements containing one for every $k \in K$ is $N_K \subseteq N_S$.

- For every possible pair $n_i, n_j \in N_K$ and $n_i \neq n_j$, there is a path $n_i \ldots n_j$ such that every $n_i \in N_K$ is connected to every other $n_j \in N_K$. 

---

**Figure 1:** Example data graph, queries and keyword matching elements.
We call such a graph a d-length Steiner graph when direct paths, i.e., paths that connect exactly two keyword elements \( n_i, n_j \) s.t. there is no other keyword element \( n_k \) in the path between \( n_i \) and \( n_j \), have length \( d \) or less (we use \( n_i \ldots n_j \) to denote these paths).

This notion captures the standard semantics of keyword query answers [10, 18]. Limited by its internal index structure, EASE [12] departs from this and employs an alternative semantics: while it also searches for Steiner graphs, it implicitly assumes one “center keyword element” that is connected to all other keyword elements over a maximum distance \( d \). While this leads to a more restricted search space, results corresponding to the standard semantics may be missed. In this work, we support the standard semantics.

The d-length restriction only implies that for every keyword element \( n_i \) in the Steiner graph, there is at least one other keyword element connected to it via a path of length \( d \) or less (i.e., the directly connected one), while all other keyword elements may be (indirectly) connected to it via longer paths.

The computation of top-k Steiner graphs typically requires a monotonic function for ranking. Widely used in keyword search is the score of a node calculated using a probabilistic IR model. Scores which measure nodes’ prestige have also been incorporated – PageRank for instance [7]. Besides, keyword search solutions generally rest on the assumption that more compact Steiner graphs more likely match the user needs, i.e., the length of paths connecting two keyword elements has a negative effect on the rank. The solution proposed here is orthogonal to the ranking function being used. For ease of exposition, we use path length as the only metric:

**Definition 4.** Let \( G_S \) be a Steiner graph and \( P \) be the set of direct paths that connect its keyword elements, the rank of \( G_S \) is determined by Score \( (G_S) = \sum_{p \in P} \text{Score}(p) \), where Score \( (p) = \frac{1}{\text{length}(p)} \).

The solution we propose is based on two main operations. For data access, a specialized index is employed to retrieve connecting paths for every pair of keyword elements. In a top-k fashion, these paths are then joined to construct the best ranked Steiner graphs.

### 3. D-LENGTH 2-HOP COVER

A 2-hop cover is a compact representation of connections in a graph that has been previously used to answer reachability and distance queries [17, 1]. In this work, we propose to leverage this concept for retrieving paths between two keyword elements. Our data graph is directed. However, while it is essential to find paths during the computation of Steiner graphs, the direction of the edges that establish these paths are irrelevant (have to be preserved only for outputting the actual results). For the efficient retrieval of paths, we propose the d-length 2-hop cover, which in contrast to the 2-hop cover employed in existing work, only considers paths up to length \( d \). First, we define the neighborhood of a node \( u \) that captures paths from \( u \) to nodes in its vicinity:

**Definition 5 (Neighborhood).** A neighborhood label (or short: neighborhood) \( NB_u \subseteq G \) of a node \( u \in N \) (also called the center node of \( NB_u \)) is the union set of nodes and edges that are connected to \( u \) via some paths. The set of paths between \( u \) and a node \( w \in NB_u \) is denoted as \( P(u, w) \), and its Score \( (P(u, w)) \) is defined as \( \max \{ \text{Score}(p) | p \in P(u, w) \} \).

The combined information is represented as a path entry \((u, s, w) \) (or short: \(u, w\)), where \( u \) is the center node of \( NB_u \), \( w \in NB_u \), and \( s = \text{Score}(P(u, w)) \). A d-neighborhood of node \( u \) is the set of nodes and edges connected to \( u \) via paths of length \( d \) or less.

**Definition 6 (d-length 2-hop Cover).** A labeling of a graph \( G = (N, E) \), which consists of a set of neighborhood labels is a d-length 2-hop cover, if the following two conditions hold for any two nodes \( u, v \in N_G \): 1) if there is path of length \( d \) or less between \( u \) and \( v \) then \( NB_u \cap NB_v \neq \emptyset \); 2) all paths of length \( d \) or less between the center nodes \( u \) and \( v \) are of the form \((u, s_1, \ldots, v)\), where \( w \in NB_u \cap NB_v \) (\( w \) is called a hop node).

This cover is used to find all paths of length \( d \) or less between two nodes by forming the intersection of their neighborhoods, i.e., the set of paths between \( u \) and \( v \) is \( P(u, v) = \{ p_{uv} \mid p_{uv} \in P(u, w), p_{uv} \in P(w, v) \} \) for all \( u, v \in N \).

### 3.1 Construction

The problem of constructing a 2-hop cover of minimal size has been reduced to the minimum set cover problem that is NP-hard [2]. Consequently, approximative algorithms are necessary for dealing with large graphs [17, 1]. The basic idea is as follows: in a greedy manner, one neighborhood is selected at every iteration to prune redundant paths in other neighborhoods. The goal is to reduce redundancy as much as possible, while preserving the 2-hop cover property.

We adopt existing algorithms in two directions. (1) First, we define a trivial d-length 2-hop cover for the keyword search setting, which is used as a basis for later pruning. (2) Then, the d-length restriction is taken into account during the pruning of paths. The detailed algorithm, complexity and proofs are presented in our report [11].

Basically, we observe that a trivial d-length 2-hop cover is simply the set of d-neighborhoods:

**Theorem 1.** The set of d-neighborhoods constructed for every node \( u \in N \) in \( G(N, E) \) is a valid d-length 2-hop cover.

**Proof Sketch:** By definition, a d-neighborhood of a node \( u \) contains all nodes in \( G(N, E) \) reachable from \( u \) via a path of length \( d \) or less. Thus, for all \( u, v \in N \), \( u, v \in NB_u \), and \( v \in NB_u \), if \( u \ldots v \) then \( u \in NB_u \) and \( v \in NB_u \). Hence, \( NB_u \cap NB_v \) must be non-empty (this ensures condition 1). Then, the d-length restriction is taken into account during the pruning of paths.

Further, the set \( NB_u \cap NB_v \) contains hop nodes for all paths of length \( 2 \times d \) for less, which include all paths of length \( d \) or less (this ensures condition 2).

Then, the following intuition is employed for pruning redundant paths captured by this trivial d-length 2-hop cover: Given two nodes \( u, v \in N \) that are connected via the set of paths \( P_{uv} = \{ u, \ldots, v \} \) of length \( d \) or less, \( NB_u \) contains some “partial paths” \( P_{uv} = \{ \ldots, v \} \) that are also in \( NB_v \), and vice versa \( NB_v \) contains the parts \( P_{uv} = \{ u, \ldots, w \} \) that are also in \( NB_u \). Then, these paths are redundant in the sense that after removing \( P_{uv} \) from \( NB_u \) and \( P_{uv} \) from \( NB_v \), all the paths in \( P_{uv} \) are still preserved and can be computed via hop nodes in \( NB_u \cap NB_v \).

The approximate algorithm for this consists of \( |N| - 1 \) iterations. At each step, a neighborhood \( NB \) is selected.
(based on “pruning power” simply measured in terms of node counts), marked as complete, and used for pruning other neighborhoods not completed yet. Neighborhood pruning is discussed in the example below.

**Example 1.** Fig. 2 shows the two not yet pruned neighborhoods of \( p_2 \) and \( p_3 \). Here, we prune \( NB_{p_3} \) using \( NB_{p_2} \) by starting a simultaneous breadth-first search towards the leaf nodes from \( p_2 \). All edges in \( NB_{p_3} \) that also occur in \( NB_{p_2} \) are marked and later removed from \( NB_{p_3} \). The part pruned this way is highlighted in Fig. 2.

Note that the procedure here operates at the level of paths while existing work on keyword search applies pruning at the level of neighborhoods, i.e., a neighborhood is discarded only when it is completely covered by one another [12].

### 3.2 Storage

We store neighborhoods as well as the actual paths. We define the path and path entry indexes, which are used during the computation of Steiner graphs.

**Definition 7 (Path Entry Index).** The path entry index \( I_{PE} \) maps nodes \( u \in N \) to a list of path entries \( (u, s, w) \), sorted by \( s = \text{Score}(P(u, w)) \), where \( w \in NB_u \).

**Definition 8 (Path Index).** The path index \( I_P \) maps a path entry \( (u, w) \) to a list of paths \( P(u, w) \), sorted by \( \text{Score}(p) \).

**Table 1: Example path entry index.**

<table>
<thead>
<tr>
<th>Node</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_2 )</td>
<td>( (p_2, 2.0, p_2) )</td>
</tr>
<tr>
<td></td>
<td>( (p_2, 1.0, p_1) ) ( (p_2, 1.0, p_3) ) ( (p_2, 1.0, o_2) )</td>
</tr>
<tr>
<td></td>
<td>( (p_2, 0.5, l_2) ) ( (p_2, 0.5, p_4) ) ( (p_2, 0.5, o_1) )</td>
</tr>
</tbody>
</table>

**Table 2: Example path index.**

<table>
<thead>
<tr>
<th>Center.Hop</th>
<th>Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_2, p_1 )</td>
<td>( 1.0 ) ( (p_1, \text{knows}, p_2) )</td>
</tr>
<tr>
<td></td>
<td>( 1.0 ) ( (p_2, \text{knows}, p_1) )</td>
</tr>
<tr>
<td>( p_2, l_1 )</td>
<td>( 0.5 ) ( (p_2, \text{worksAt}, o_2) ) ( (o_2, \text{locatedIn}, l_1) )</td>
</tr>
<tr>
<td>( p_2, o_1 )</td>
<td>( 0.5 ) ( (p_2, \text{knows}, p_3) ) ( (p_3, \text{worksAt}, o_1) )</td>
</tr>
<tr>
<td></td>
<td>( 0.5 ) ( (p_2, \text{knows}, p_2) ) ( (p_3, \text{worksAt}, o_1) )</td>
</tr>
</tbody>
</table>

**Example 2.** Tab. 1 shows an index entry for node \( p_2 \) from the running example. It contains path entries from \( p_2 \) to all hop nodes in the neighborhood with their associated scores. Recall that this is simply the maximal score of paths from \( p_2 \) to a hop node. Tab. 2 shows an excerpt of the path index for nodes in the neighborhood of \( p_2 \). For example, the key \( (p_2, o_1) \) can be used to retrieve all paths (and their scores) from \( p_2 \) to \( o_1 \). In this case there are two such paths.

### 4. KEYWORD QUERY PROCESSING

In this section we present the process of answering keyword queries using a \( d \)-length 2-hop cover.

#### 4.1 Basic Join Operations

Given a keyword query \( K \) and its keyword elements \( N_K \), the goal is to find Steiner graphs. The basic idea is to use the pruned neighborhoods of the \( d \)-length 2-hop cover to find paths between every pair of keyword elements and iteratively join them until they all are connected. For this, we (1) firstly perform data access operations to retrieve the neighborhoods for every keyword, called keyword neighborhoods, (2) then perform neighborhood join to merge two keyword neighborhoods to obtain a keyword graph and then successively (3) apply graph joins to combine a keyword neighborhood with a keyword graph.

**Definition 9 (Keyword Neighborhood).** Given a keyword \( k \) and its keyword elements \( N_k \), the keyword neighborhood \( NB_k \) of \( k \) is the union of path entries retrieved from the index \( I_{PE_k} \). It captures the neighborhoods of all keyword elements in \( N_k \), i.e., \( NB_k = \bigcup_{n \in N_k} I_{PE_k}(n) \).

**Definition 10 (Neighborhood Join).** Given two keyword neighborhoods \( NB_k, NB_{k'} \), the neighborhood join \( \bowtie_{NB_k} \) combines two path entries \( (n_{k_1}, w) \) in \( NB_k \) and \( (n_{k_2}, w) \) in \( NB_{k'} \) that match on \( w \), i.e., \( NB_k \bowtie_{NB_k} NB_{k'} = \{ (n_{k_1}, w) + + (n_{k_2}, w), w \in NB_{k_1}, n_{k_2}, w \in NB_{k_2}, \text{distance}(n_{k_1}, w) + + (n_{k_2}, w) \leq d \} \). The result \( (n_{k_1}, w) + (n_{k_2}, w) \) establishes a path between \( n_{k_1} \) and \( n_{k_2} \) with \( w \) being the hop node. A join of 2 or more path entries such as \( (n_{k_1}, w) + (n_{k_2}, w) \) form a keyword graph \( G_K = (N_K, PE_K) \) with \( N_K \) subdivided into the two disjoint sets of center nodes \( N_C \) and hop nodes \( N_H \). For every path entry \( (n, w) \in PE_K \), we have \( n \in N_C \) and \( w \in N_H \).

Based on this \( \bowtie_{NB} \) operation, we provide Theorem 2 for computing results of 2-keyword queries.

**Theorem 2.** Given a 2-keyword query \( K = \{k_1, k_2\} \) and the retrieved keyword neighborhoods \( NB_{k_1}, NB_{k_2} \), the keyword graphs resulting from \( NB_{k_1} \bowtie_{NB} NB_{k_2} \) are \( d \)-length Steiner graphs for \( K \).

**Proof Sketch:** In this 2-keywords setting, the set of all \( d \)-length Steiner graphs for \( K = \{k_1, k_2\} \) is exactly the set of all paths of length \( d \) or less between the pairs of nodes \( n_1 \in N_{k_1}, n_2 \in N_{k_2} \). By definition of the \( d \)-length 2-hop cover, the intersection \( NB_{k_1} \cap NB_{k_2} \) contains the hop nodes for all path of length \( d \) or less between \( n_1 \) and \( n_2 \). The neighborhood join \( NB_{k_1} \bowtie_{NB} NB_{k_2} \) leverages this, joining all paths in \( NB_{k_1} \) with paths in \( NB_{k_2} \) on the hop node \( w \) in \( NB_{k_1} \cap NB_{k_2} \). The result comprises all paths connecting the pairs of nodes \( n_1 \in N_{k_1}, n_2 \in N_{k_2} \) that are of length \( d \) or less, i.e., all \( d \)-length Steiner graphs for \( K \).

A keyword graph resulting from the neighborhood join contains only path entries in \( NB_{k_1} \) and \( NB_{k_2} \), which connect the two keyword elements \( n_{k_1} \) and \( n_{k_2} \). Thus, only hop nodes in these paths remain, while other nodes in the neighborhood of \( n_{k_1} \) and \( n_{k_2} \) have been eliminated during the process. For the subsequent graph join \( \bowtie_{G} \), which takes a keyword graph \( G_K \) and a keyword neighborhood \( NB_k \) as inputs, these hop nodes still have to be considered because they might help in connecting \( G_K \) with \( NB_k \). In order to
Given a keyword graph \( G \), the keyword graph neighborhood \( NB \) is a set of graph entries constructed from \( G \) and a path entry \( (n, w) \), i.e., \( NB = \{ G + (n, w) | (n, w) \in P_E(n) \} \), where every \( G + (n, w) \) forms a graph entry denoted as \( G_N \).

As a result of this expansion, each graph entry \( G_N \) contains a “free” path entry \( (n, w) \), whose hop node \( w \) is yet not connected. Similar to \( NB \), this hop node is used as the join attribute of the \( NB \) operation.

Definition 12 (Graph Join). Given a keyword neighborhood \( NB \) and a keyword graph neighborhood \( NBG \), the result of a graph join \( NBG \) is the combination of path entries \( (n, w) \in NB \) and graph entries \( G_N \) defined as \( NBG = \{ G_N + (n, w) | (n, w) \in P_E(n) \} \).

The result of this operation is also a keyword graph, which however, connects more than two keyword elements via a number of path entries. For each entry, we use the path index to retrieve all paths between its center and hop nodes and join them along the edges of the keyword graph to construct the final Steiner graphs.

Example 3. Fig. 3 shows an example for processing the 2-neighborhoods of nodes \( p_2, p_4 \). The keyword neighborhoods have three hop nodes in common: \( p_3, p_4, p_5, o_n \). The figure shows the keyword graph resulting from a join combination of path entries \( (p, p_3), (p_4, p_3), (p_4, p_5) \) and two entries of the resulting keyword graph neighborhood. Two Steiner graphs are generated from this keyword graph.

4.2 Integrated Query Plan

A query \( Q \) can be processed according to a query plan. It is a sequence of join operators \( (NB \times NB, NB \times G, \ldots, G \times G) \), starting with the operator \( NB \times NB \) joining two keyword neighborhoods (called base inputs), followed by \( NB \times G \) operators that combine a base input with a graph. Since the base inputs represent elements retrieved for the keywords, a plan also represents a particular order of query keywords.

The principle difference to standard database join processing is that in keyword search, connections between two keywords are not known in advance. This is why typically, connections have to be explored at runtime by traversing edges of the data graphs that are loaded into memory. For this, efficient techniques for searching results in the data graph [10, 3], or pruning joined neighborhoods [12] have been proposed. For “searching” keyword search results via join processing, we observe there is no single optimal plan but different plans might produce different results, or even no results at all, as illustrated by Example 4.

Example 4. Consider query \( Q_3 \) from Fig. 1. For \( d = 2 \), keywords steve and malta are connected and mary is connected to steve, but not to malta. Fig. 4 shows two query plans for query \( Q_3 \). The first join on the keyword neighborhoods of steve and malta produces empty results as there are no paths of length \( d \) or less. However, results can be produced when the keyword neighborhoods of steve and malta or steve and mary are joined first.

Now, we can use Theorem 3 to compute keyword search results for queries with more than 2 keywords, using an integrated query plan:

Theorem 3. Let \( G \) be the data graph, \( K \) be an \( n \)-keyword query, and \( P(K) \) represents all permutations of keywords in the set \( K \) where each \( p \in P(K) \) stands for a query plan, keyword graphs resulting from executing all \( p \in P(K) \) capture all Steiner graphs for \( K \) that can be obtained from \( G \).

Proof Sketch: Let \( K = \{ N_1, \ldots, N_k \} \) be the set of keyword elements obtained for \( K \). We must generate (1) all graphs from \( G \), which (2) contain an element \( N_k \) for every \( k \in K \), (3) all these keyword elements are pairwise connected, and (4) direct paths between two keyword elements are of length \( d \) or less. The integrated plan contains different orders of joins. Every join is either \( NB \times NB \) or \( NB \times G \). As both operations rest on the same concept of \( d \)-length 2-hop cover (i.e., join two path entries on a hop node), the resulting paths are of length \( d \) or less (4). Every plan joins every
The number of join operators, which always combine a graph with a base input such that there is a correspondence between level and the number of inputs, i.e., every join operator at level $l$ consumes exactly $l+1$ base inputs. That is, operators at level $l$ take all the base inputs from previous join operators and base inputs not processed yet. Thus, the total number of join operators is $N_{K} = N(1) + \sum_{l=2}^{|K|-1}(|K| - l)N(l-1)$.

**Complexity**

Given the number of join operators, the upper bound on complexity of keyword query processing can be established as follows. In worst case, the result of a join is the cross product of its inputs. Here, input size is given by the number of path entries in each keyword neighborhood, which can be calculated as $|N| \cdot b^d$ for each $k \in K$, where $b$ is the average branching factor of the pruned d-neighborhood. $b^d$ gives the number of paths in the neighborhood while $|N_k|$ denotes the total number of neighborhoods that can be obtained for a keyword $k$. The upper bound can then be calculated as $N(K) \cdot \max\{|N_k| \cdot b^d | k \in K\}^2$. Note that due to pruning at the level of paths, the branching factor of the neighborhoods is smaller than the one of the original graph.

As illustrated in Example 5, the integrated query plan consists of operators at levels $l = 0, \ldots, |K|$. Level 0 represents data access operators, employing $|K|$ base inputs. Level 1 to $|K| - 1$ contain join operators. Only $\forall_{N_B}$ operators are needed at level 1. Subsequent levels consist exclusively of $\forall_{G}$ operators, which always combine a graph with a base input such that there is a correspondence between level and the number of inputs, i.e., every join operator at level $l$ consumes exactly $l+1$ base inputs. That is, operators at level $l$ take all the base inputs from previous join operators and base inputs not processed yet. Thus, the total number of join operators is $N_{K} = N(1) + \sum_{l=2}^{|K|-1}(|K| - l)N(l-1)$.

**Complexity**

Given the number of join operators, the upper bound on complexity of keyword query processing can be established as follows. In worst case, the result of a join is the cross product of its inputs. Here, input size is given by the number of path entries in each keyword neighborhood, which can be calculated as $|N| \cdot b^d$ for each $k \in K$, where $b$ is the average branching factor of the pruned d-neighborhood. $b^d$ gives the number of paths in the neighborhood while $|N_k|$ denotes the total number of neighborhoods that can be obtained for a keyword $k$. The upper bound can then be calculated as $N(K) \cdot \max\{|N_k| \cdot b^d | k \in K\}^2$. Note that due to pruning at the level of paths, the branching factor of the neighborhoods is smaller than the one of the original graph.

**Example 5.** Fig. 5 shows an excerpt of the integrated query plan for a query of size 4. One subplan for the join order (steve $\forall_{N_B}$ mary $\forall_G$ corp $\forall_G$ malta) is emphasized. The figure also shows that intermediate results such as the one produced by steve $\forall_{N_B}$ mary is used for this as well as for one another plan.

### 4.3 Top-k Keyword-Join Processing

Different top-k query processing techniques have been proposed to reach early termination after obtaining the top-k results [9]. Of particular interest in this context is the top-k join, which produces ranked join results without consuming all inputs [8].

A top-k join operator, also called rank join, takes two inputs that are sorted by the scores of their elements and iteratively processes them as data streams (i.e., no random access) to produce $k$ top results. Crucial in top-k processing is the threshold $T$, an upper bound maintained for every join operator to capture the maximum score that can be achieved using as yet unprocessed input tuples. A join result can be reported if its score is higher than $T$. For a join between the inputs 1 and 2, $T = \max(\text{Agg}(s_i^{max}), \pi_2, \text{Agg}(\pi_1, s_2^{max}))$, where $\text{Agg}(\cdot)$ returns the score for the joined result, $s_i^{max}$ denotes the best score of all tuples of input $i$, and $\pi_1$ denotes the score of the tuple lastly seen from $i$. The last seen score and threshold decrease as new tuples are processed (because they arrive at a decreasing order of scores).

This concept of rank join is utilized for top-k query processing in a pull-based architecture: each operator in the query plan has a next method that is called to produce the next result (i.e., a reportable result whose score exceeds $T$); a tree-shaped query plan is employed where the root operator calls the next method of lower level operators, which in turn call next on the base inputs. Processing inputs in this way is guaranteed to preserve the top-k property, i.e., the resulting $k$ results have best scores. This follows directly from the notion of threshold, as results reported by the next methods have scores greater than $T$, and thus are guaranteed to have scores greater than all other result candidates that can be produced using remaining tuples.

**Top-k Processing**

We adopt this top-k query processing to our setting, as early termination can avoid the processing of a possibly large amount of join operators in the integrated query plan. Further, implementing $\forall_G$ and $\forall_{N_B}$ as rank join naturally enables us to treat data as input streams. Instead of loading large graphs into memory [12] at once, we use rank join to incrementally access and process streams of path entries. We note that the only difference to the stan-
dard query plan is that in the integrated query plan, the root is a union operator. We apply the notion of threshold to obtain a rank union operator in order to preserve the top-
k property. The threshold of a union operator \( \cup \) is defined as \( T(\cup) = \max\{T_{\text{op}} \mid \text{op} \in \cup \} \), where \( T_{\text{op}} \) is the threshold of a rank join operator \( \text{prj} \), and \( \cup \) is the set of all rank join operators that feed into \( \cup \). A result for \( \cup \) can be reported when its score is higher or equal to \( T(\cup) \). Processing the integrated query plan with the rank version of our operators (i.e., operators with thresholds for result output) using the pull-based procedure discussed above [9], yields top-
k results. We provide the following Theorem to capture this:

**Theorem 4.** Given a query \( K \), the data \( G \) and a predefined parameter \( k \), an integrated query plan consisting of rank operators that feed into \( \cup \) produces a sorted list of \( k \) Steiner graphs \( \{G_{s_1}, \ldots, G_{s_k}\} \) for \( K \) s.t. there exist no other Steiner graph \( G_{s_i} \) in \( G \) with \( \text{Score}(G_{s_i}) > \text{Score}(G_{s_k}) \).

The proof for this essentially exploits the top-
k property of the underlying rank operators. The intuition is this: the integrated query plan can be decomposed into a number of subplans, each represents a different join order for the keywords in \( K \). Because every such subplan corresponds exactly to the notion of query plan used previously [9], the established top-
k property holds in this case, providing the guarantee that every subplan yields the \( k \) best Steiner graphs (of one particular structure reflected by the join order). The rank union operator reports a top-
k result only when its score equals or is higher than the maximum of the thresholds of the underlying rank join operators (subplans). This maximum threshold can be seen as a global threshold applied to results of all subplans. The top-
k property is preserved because by definition, no subplan can produce further results with scores exceeding this global threshold.

**Push-based Architecture** The pull-based architecture [9] is “driven” by the final results. When called by the next method, each operator consumes its inputs until it is able to produce one result. Such an architecture is problematic in keyword query processing, as some operators of the integrated query plan may produce no results, which can only be determined after completely reading at least two base inputs. Detecting and avoiding these “broken operators” (and the subplans resulting from them) can improve efficiency. To this end, we propose a push-based architecture, which is “driven” by the data access operators instead of the results. These operators push their outputs into the join operators they are connected to, from where keyword graphs propagate upwards through the query plans. In contrast to pull-based architectures, control and data flow are in the same rather than opposite direction.

**Operator Ranking** Using such a push-based architecture, we can apply operator ranking to accommodate knowledge related to which operators are broken, which ones produce more results, or which ones produce results earlier than others. The ranking of operators determines its order during execution. That is, we propose not only to rank inputs and (partial) results, but also the join operators themselves. We associate rank join operators with a global score, which is estimated based on current results (\( R \)), and upper bound estimates for subsequent join operations to be performed on the remaining base inputs (\( N_{B_k} \)).

**Definition 13 (Operator Rank).** Given the results \( R \) of a \( \text{prj}_{\text{Rank}} \) operator and the remaining set \( N_{B_k} \) of keyword neighborhoods not included in the inputs to \( \text{prj}_{\text{Rank}} \), the global score of \( \text{prj}_{\text{Rank}} \) is defined as \( S = \max\{\text{Score}(G_K) \mid G_K \in R\} + \sum_{N_{B_k} \in N_{B_k}} \max\{\text{Score}(P) \mid P \in N_{B_k}\} \).

**Query Execution** The global score is used to guide query execution. When a join operator has results that can be pushed to the next operator, it is associated with an (updated) global score and becomes active. If there are no active join operators, the lower level data access operators are activated in a round-robin fashion, each pushing a path entry into all rank \( \text{prj}_{\text{NB}} \) operators they are connected to. If there are active operators, the operator with the highest global score pushes its results to subsequent operators until its result stack is empty, or another join operator has a better global score.

**Detailed Algorithms** Alg. 1 shows the query execution algorithm. First, the keyword elements \( N_k \) for all \( k \) in \( K \) are retrieved and their keyword neighborhoods are created. Priority queue \( J \) keeps track of all currently active join operators, sorted by their global scores. While the number of results is lower than \( k \), the algorithm gets the best result (via the \( \text{topR} \) method) from the topmost push rank join (PRJ) operator \( \text{op} \) in the queue \( J \). By calling the push method of every connected operators (retrieved via \( \text{nextLevelOps} \)), this result is propagated to the next level. Now we check if the second best result (the one not been pushed yet) passes the threshold to determine if \( \text{op} \) is still active, and add it back to the queue if this is the case. If no active operators are available, the base inputs are activated until at least one operator becomes active or all inputs are exhausted.

**Algorithm 1: EvaluateQuery(\( K, I_{PE}, k \))**

<table>
<thead>
<tr>
<th>Input</th>
<th>Keyword query ( K ), ( k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Queue ( J ) of active join operators, the integrated query plan ( IQP ).</td>
</tr>
<tr>
<td>( 1 ) Retrieve ( N_k ) for all ( k ) in ( K )</td>
<td></td>
</tr>
<tr>
<td>( 2 ) Retrieve ( N_{B_k} \in N_{B_k} ) for each ( N_k )</td>
<td></td>
</tr>
<tr>
<td>( 3 ) \text{while} \ number of final results ( &lt; k ) \text{do}</td>
<td></td>
</tr>
<tr>
<td>( 4 ) \text{if} \ #J &gt; 0 \text{then}</td>
<td></td>
</tr>
<tr>
<td>( 5 ) \text{op} = \text{J.pop()}</td>
<td></td>
</tr>
<tr>
<td>( 6 ) \text{NOps} = \text{IQP.nxtLevelOps(op)}</td>
<td></td>
</tr>
<tr>
<td>( 7 ) \text{foreach} \ \text{nop} \in \text{NOps} \text{ do} \text{nop.push(op.topR)}</td>
<td></td>
</tr>
<tr>
<td>( 8 ) \text{op.active} = \exists \text{op.secR}(\text{Score(op.secR)} &gt; \text{op.T})</td>
<td></td>
</tr>
<tr>
<td>( 9 ) \text{if} \ \text{op.active} \text{ then} \text{Add op to} J</td>
<td></td>
</tr>
<tr>
<td>( 10 ) \text{else}</td>
<td></td>
</tr>
<tr>
<td>( 11 ) \text{Select next input} \text{N}_{B_k}</td>
<td></td>
</tr>
<tr>
<td>( 12 ) \text{NOps} = \text{IQP.nxtLevelOps(} \text{N}_{B_k} \text{)}</td>
<td></td>
</tr>
<tr>
<td>( 13 ) \text{foreach} \ \text{nop} \in \text{NOps} \text{ do} \text{nop.push(nB_k.topR)}</td>
<td></td>
</tr>
<tr>
<td>( 14 ) Add all active operators to ( J )</td>
<td></td>
</tr>
</tbody>
</table>

The employed PRJ shown in Alg. 2 is based on the hash rank join [8]: it also maintains a hash table for each input \( (H_1, H_2) \) for efficient lookup and computation of the join results \( J \), and a threshold \( T \) (updated via the \( \text{threshold} \) method). Additionally, the PRJ operator also maintains the global score \( S \). As previously defined, \( S \) is computed from the score of the currently best result and the maximum score of the combination of remaining base inputs that have not been joined during the process, i.e., are not part of the current results.
Results pushed to operators at level 1

<table>
<thead>
<tr>
<th>Key</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>{p2, 5.0, p4}</td>
</tr>
<tr>
<td>p3</td>
<td>{p2, 5.0, p4}</td>
</tr>
<tr>
<td>p4</td>
<td>{p2, 0.5, p6}</td>
</tr>
</tbody>
</table>

Input from operators at level l-1

<table>
<thead>
<tr>
<th>Key</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>{p2, 5.0, l}</td>
</tr>
<tr>
<td>p2</td>
<td>{p1, 0.5, p3}</td>
</tr>
<tr>
<td>p3</td>
<td>{p2, 2.0, p4}</td>
</tr>
<tr>
<td>p4</td>
<td>{p2, 0.5, p6}</td>
</tr>
</tbody>
</table>

Figure 6: Keyword processing based on PRJ.

Algorithm 2: PRJ: Push(r)

**Input:** Pushed input element r  
**Data:** Queue R of sorted results, hash tables H1, H2 for inputs, global score S

1. // Use m, n to identify hash tables and inputs  
2. if r is left input then m = 1, n = 2  
3. else m = 2, n=1  
4. Insert r into hash table H_m  
5. T = threshold()  
6. J = probe H_n for valid joined results with keys of r  
7. if |R| > 0 then  
8. if Score(this.topRes) > T then  
9. this.active = true  
10. Update S

**Example 6.** Fig. 6 shows a PRJ operator at level l, joining neighborhoods of p2 and p4. The inputs, i.e., path entries (p2, 0.5, l2), (p2, 0.5, o1) from one operator and inputs (...) from other operators, are pushed from operators at level l - 1. The two hash tables contain 4 x 2 elements already processed. One output with score 2.5 has been pushed to the next level. A candidate result with score 2.0 is in the queue. Because the last seen score is 0.5, T = 2.0 + 0.5 = 2.5. As the result has a score less than threshold T, it cannot be pushed to subsequent operators yet (is thus inactive). This may change subsequently because T may decrease when lower input scores are processed. The query execution controls the push operation: an operator is only activated to push its results if its global score S is the highest among all active operators.

Note that the proposed modifications to the basic strategy, i.e., push-based execution in combination with operator-ranking, only aims at improving the efficiency. It has no effect on the top-k property. Essentially, because the same threshold checks are carried out, Theorem 4 must also hold in this case.

5. RELATED WORK

We discussed research work on neighborhood indexing [17, 1] and join processing [9, 8] that underlies our approach. We also provided an overview of existing keyword search solutions, which can be categorized into the categories of (1) schema-based ([6, 13, 14, 16, 15]) and (2) schema-agnostic ([10, 5, 12]) approaches. A similar distinction was made in a recent survey of keyword search in relational databases [19]. The capabilities of the underlying database engine can be fully utilized by the former approaches, while the applicability of the latter approaches is not limited to structured data that have well defined schemas. Our work follows the line of schema-agnostic approaches, aiming at providing keyword search support for different types of data, from structured to semi-structured up to unstructured data [12]. In particular, we compute Steiner graphs by operating directly on the data graph [12, 10, 18]. We make two kinds of contributions that are substantially different from previous work along this line: (1) previous strategies for searching Steiner graphs load the data graph into memory, and then find Steiner graphs by traversing graph edges [10], or pruning them [12]. Instead, we propose the use of join operations for this. (2) Further, for efficient data access, we propose an index that stores materialized paths in the neighborhoods of graph elements. While the concept of neighborhood is conceptually similar to the notion used by EASE [12], the data structure to implement it is more efficient because the employed strategies for neighborhood pruning is based on the more fine-grained notion of paths, instead of graphs. Further, this implementation enables us to conceive a neighborhood as a set of paths, enabling the search for Steiner graphs to be conducted via path-based join operations.

6. EVALUATION

**Systems** We compare our approach with an implementation of EASE [12], a state-of-the-art native keyword search solution. We also implemented the bidirectional search algorithm (BDS) [10]. We used two variants of our approach: the first features operator ranking (KJ) and the second one does not (KJU).

For all approaches, a Lucene keyword index is used to retrieve the top 300 matching keyword elements for each keyword of the query. The time needed for this is not counted and is the same for all systems. For BDS, the exploration is performed directly on the data graph to obtain top-k Steiner graphs. For EASE, first all maximal 2-radius graphs, which contain at least one keyword element for every keyword, are identified. The union of these graphs computed for all query keywords is then loaded into memory, and pruned successively to obtain Steiner graphs.

All systems were implemented in Java 1.6 on top of Oracle Berkeley DB (Java Edition). Experiments were performed on a Linux system with two Intel Xeon 2.80GHz Dual-Core processors, a Samsung HE322HJ SATA 320GB disk and 8GB of main memory, 4GB of which were assigned to the Java VM. Operating system caches were cleared after each query run.

**Data and Queries** We used a crawl of the Billion Triple Challenge 2009 \(^2\) (BTC) dataset, and 3 DBLP datasets from the SP2Bench benchmark \(^3\) (DBLP) that vary in size, which we employ for the scalability experiment. Tab. 3 shows detailed statistics about the datasets and the size of the input.

\(^2\)http://vmlion25.deri.ie/  
\(^3\)http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B
indexes for KJ (path entry index $I_{PE}$ and path index $I_P$) and EASE. For each dataset, we created nine keyword queries of length 2 ($Q_1 - Q_3$), 3 ($Q_4 - Q_6$) and 4 ($Q_7 - Q_9$). Table 4 shows example queries for both datasets.

| Dataset | $|K|$ | $|K| - 1| * d$ |
|---------|-----|----------------|
| BTC     | 1M  | 237            |
| DBLP1   | 1M  | 42             |
| DBLP5   | 5M  | 250            |
| DBLP10  | 10M | 559            |

Table 3: Dataset and index statistics (index sizes in MB).

**Index Statistics** For each dataset we created the path entry and path indexes with $d = 2$ for our approach and an index containing maximal 2-radius graphs for EASE. As discussed, the scale-free nature of Web data graphs makes values $d > 2$ impractical because neighborhoods become as large as the entire data graph. Note that unlike EASE, this is not such a strong limitation because paths between 2 keyword elements can still have lengths up to $|K - 1| * d$ (instead of $2 * d$).

Due to more fine-grained pruning, the KJ indexes were much smaller than the EASE indexes: for the DBLP datasets, the size of the KJ indexes was between 50% (DBLP1) and 21% (DBLP10) the size of the corresponding EASE indexes. For the BTC dataset, the difference was even larger: the KJ index was worse by a factor of 1.66 on DBLP10.

**Operator Ranking** We compare KJ and KJU to examine the effect of operator ranking. Fig 7(c) shows query performance for different values of $k$ for the BTC dataset. Clearly, query times for all systems increase with higher values of $k$.

**Top-k Processing** Fig 7(c) shows query performance for different keyword query lengths.

**Scalability** Fig 7(d) shows overall query times for the three different DBLP datasets. The query time is split into the time for data access (load) and the time for join processing (process). We can see that data access makes up the main share of total time. For all datasets, KJ outperforms EASE. With respect to scalability, we found that the difference in performance between EASE and our system increases with larger datasets. On DBLP1, the smallest dataset, EASE was worse by a factor of 1.44. This increases with larger versions of the DBLP dataset, e.g. EASE was worse by a factor of 1.66 on DBLP10.

For an examination of the separate impact of data access
(load) and data processing (process). Figure 8 illustrates a decomposition of query time.

**Data Access** Fig. 8 shows access times at different query lengths $|K|$ for DBLP with $k = 10$. For all systems, access times increase with longer queries. Again, we can see the positive effect of operator ranking, as access times for KJ do not increase as sharply as for KJU. For $|K| = 2$, both KJ and KJU exhibit similar access times, whereas for $|K| = 4$ KJU is 1.6 times worse than KJ. Access times for EASE are worse than KJ for all query lengths. This confirms that the more fine-grained path-level pruning implemented by KJ helps to focus on smaller neighborhoods. Further, this efficiency gain can also be attributed to the fact that instead of loading all entire neighborhoods (i.e., all matching 2-radius graphs), KJ only loads path entries necessary to compute top-k results.

**Data Processing** Fig. 8(b) shows processing times at different values for $|K|$. Clearly, processing times increase for larger values of $|K|$. However, this effect is more pronounced for KJ and KJU than for EASE. From $|K| = 2$ to $|K| = 4$, KJ’s processing times increase by a factor 12.2 where EASE times increase only by 1.6. This result is not surprising, given our approach supports the standard semantics whereas EASE assumes a center node. With more complex queries, the search space for KJ becomes larger. This is reflected in a higher number of query plans and join operators that have to be processed. Regardless of query length, EASE always operates on the union of the 2-radius graphs, and obtains results using the same pruning procedure.

Here, we can more clearly see the benefit of operator ranking: KJU’s processing times are higher than KJ’s times; this difference increases with $|K|$, indicating that operator ranking is beneficial when the queries are more complex.

7. CONCLUSION

We proposed a native keyword search database solution based on the basic operations data access and join. We introduced the $d$-length 2-hop index and a top-k procedure for implementing this database style processing. In experiments using large scale datasets, we showed that our solution consistently outperformed state-of-the-art solutions, i.e., it reduced storage requirement up to 86%, improved scalability by several factors, and improved performance by more than 50% on average. Further, our approach exhibits two main qualitative advantages. (1) While the existing solution requires large graphs to be loaded entirely into memory at once, the database style processing we propose can operate on data streams that are loaded successively. (2) Also, the semantics of the results supported is more general, leading to more results formed by longer paths. Whereas this work focuses on the efficiency of top-k keyword search processing, the question of how differences in semantics and scoring functions affect the quality of results is considered as main future work.

Acknowledgements. The authors would like to thank Jörg Diederich for providing the DBLP query logs. Research reported in this paper was supported by the German Federal Ministry of Education and Research (BMBF) in the igreen project (grant 01IA08005K).

8. REFERENCES


