Efficient Type-Ahead Search on Relational Data: a TASTIER Approach

Guoliang Li[†] Shengyue Ji[‡] Chen Li[‡] Jianhua Feng[†]
[†]Department of Computer Science and Technology,
Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China
[‡]Department of Computer Science, University of California, Irvine, CA 92697-3435, USA
{liguoliang,fengjh}@tsinghua.edu.cn;{shengyuj,chenli}@ics.uci.edu

ABSTRACT

Existing keyword-search systems in relational databases require users to submit a complete query to compute answers. Often users feel "left in the dark" when they have limited knowledge about the data, and have to use a try-and-see method to modify queries and find answers. In this paper we propose a novel approach to keyword search in the relational world, called TASTIER. A TASTIER system can bring instant gratification to users by supporting type-ahead search, which finds answers "on the fly" as the user types in query keywords. A main challenge is how to achieve a high interactive speed for large amounts of data in multiple tables, so that a query can be answered efficiently within milliseconds. We propose efficient index structures and algorithms for finding relevant answers on-the-fly by joining tuples in the database. We devise a partition-based method to improve query performance by grouping relevant tuples and pruning irrelevant tuples efficiently. We also develop a technique to answer a query efficiently by predicting highly relevant complete queries for the user. We have conducted a thorough experimental evaluation of the proposed techniques on real data sets to demonstrate the efficiency and practicality of this new search paradigm.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; H.2.8 [**Database Applications**]: Miscellaneous

General Terms

Algorithms, Experimentation, Performance

Keywords

Type-Ahead Search, Keyword Search, Query Prediction

1. INTRODUCTION

Keyword search is a widely accepted mechanism for querying document systems and relational databases [1, 6, 15]. One important advantage of keyword search is that it enables users to search information without knowing a complex query language such as SQL, nor having prior knowledge about the structure of the underlying data. In traditional keyword search in databases, a user composes a complete query and submits it to the system to find relevant answers. This search paradigm requires the user to formulate a good, complete query to find the interesting answers. If the user has limited knowledge about the targeted entities, often the user feels "left in the dark" when issuing queries, and has to use a try-and-see method to modify the query and find information. As an example, suppose a user wants to find a movie starring an actor, but remembers the movie's title and the actor's name vaguely. In this case, it is not convenient for the user to find the movie without knowing how to formulate a query properly.

Many systems are introducing various features to solve this problem. One of the commonly used methods is *autocomplete*, which predicts a word or phrase that the user may type based on the partial query the user has entered. Search engines nowadays automatically suggest possible keyword queries as a user types in a partial query.

In this paper, we study how to bring this type of search to the relational world. We propose an approach, called "TASTIER," which stands for type-ahead search techniques in large data sets. A TASTIER system supports type-ahead search on the underlying relational data "on the fly" as the user types in query keywords. It allows users to explore data as they type, thus brings instant gratification to the users in the search process. Figure 1 shows a screenshot of a TASTIER system that supports type-ahead search on a publication data set from DBLP (http://dblp.uni-trier.de). In the figure, a user has typed in a query "`yu keyword search sig`", and the system finds the relevant answers possibly with multiple tuples from different tables joined by foreign keys (e.g., the fourth answer). Notice that in TASTIER, the keywords in a query can appear at different places in the data, while most autocomplete systems treat a query with multiple keywords as a single string, and require the whole string to appear in the answers.

A main requirement of type-ahead search on large amounts of relational data is the need of a high interactive speed. Each keystroke from a user could invoke a query to the system, which needs to compute the answers within milliseconds. This high-speed requirement is technically challenging especially due to the following reasons. First, for a query with multiple keywords, finding tuples matching these keywords, even if the tuples are from a single table, requires

SIGMOD'09, June 29-July 2, 2009, Providence, Rhode Island, USA.

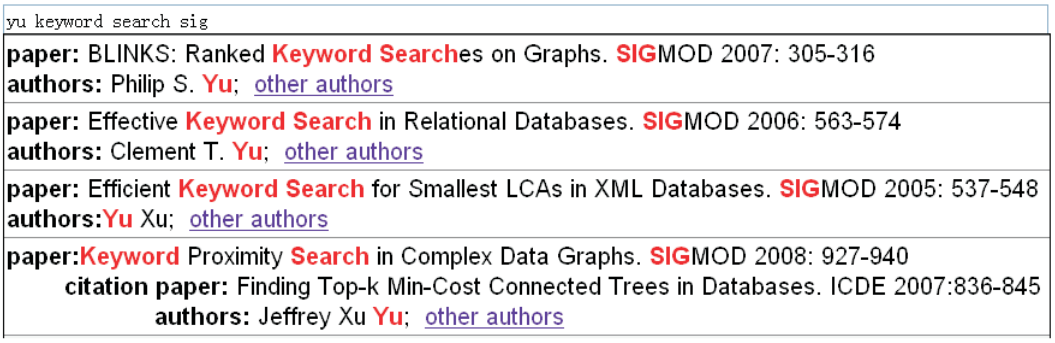


Figure 1: A screenshot of a TASTIER system using a publication database (DBLP).

intersection or join operations, which need to be done efficiently. Second, a keyword in the query can be treated as a prefix, which requires us to consider multiple completions of the prefix. The corresponding union operation increases the time complexity to find answers. Third, the computation can be even more costly if the tuples in the answers come from different tables in a relational database, leading to more join operations.

In this paper, we develop novel techniques to address these challenges, and make the following contributions. We first formulate the problem of type-ahead search in which the relational data is modeled as a database graph (Section 2). This graph presentation has the advantage of finding answers to a query that are from different tables or records but semantically related. For instance, for the keyword query in Figure 1, the system finds related papers even though none of them includes all the keywords. In the fourth answer, two related papers about "`keyword search in databases`" are joined through citation relationships to form an answer. Type-ahead search on the graph representation of a database is technically interesting and challenging. In Section 3 we propose efficient index structures and algorithms for computing relevant answers on-the-fly by joining tuples on the graph. We use a structure called "δ-step forward index" to search efficiently on candidate vertices. We study how to answer queries incrementally by using the cached results of earlier queries.

In Section 4 we study how to improve query performance by graph partitioning. The main idea is to partition the database graph into subgraphs, and build inverted lists on the subgraphs instead of vertices on the entire graph. In this way, a query is answered by first finding candidate subgraphs using inverted lists, and then searching within each candidate subgraph. We study how to partition the graph in order to construct high-quality index structures to support efficient search, and present a quantitative analysis of the effect of the subgraph number on query performance.

The approach presented so far computes answers for all possible queries and suggests possible complete queries as a "by-product." In Section 5 we develop a new approach for type-ahead search, which first predicts most likely queries and then computes answers to those predicted queries. We present index structures and techniques using Bayesian networks to predict complete queries for a query efficiently and accurately. We have conducted a thorough experimental evaluation of the proposed techniques on real data sets (Section 6). The results show that our techniques can support efficient type-ahead search on large amounts of relational data, which prove the practicality of this search paradigm.

1.1 Related Work

Keyword Search in DBMS: There have been many studies on keyword search in relational databases [15, 14, 2, 25, 27, 6, 17, 10, 13, 22, 23]. Most of them employed Steiner-tree-based methods [6, 17, 10, 13, 22]. Other methods [15, 14] generated answers composed of relevant tuples by generating and extending a candidate network following the primary-foreign-key relationship. Many techniques [12, 33, 8, 24, 31, 26, 34] have been proposed to study the problem of keyword search over XML documents.

Prediction and Autocomplete: There have been many studies on predicting queries and user actions [9, 28, 20, 11, 32, 29]. Using these techniques, a system predicts a word or a phrase that the user may type in next based on the input the user has already typed. The differences between these techniques and our work are the following. (1) Most existing techniques predict or complete a query, while TASTIER computes *answers* for the user on the fly. (2) Most existing methods do prediction using sequential data to construct their model, therefore they can only predict a word or phrase that matches sequentially with underlying data (e.g., phrases or sentences). Our techniques treat the query as a set of keywords, and support prediction of any combination of keywords that may not be close to each other in the data.

CompleteSearch: A closely related study is the work by Bast et al. [4, 5, 3]. They proposed techniques to support "CompleteSearch," in which a user types in keywords letter by letter, and the system finds records that include these keywords. CompleteSearch focuses on searching on a collection of documents or a single table. Their work in ESTER [3] combined full-text and ontology search. Our work differs from theirs in the following aspects. (1) We study type-ahead search in relational databases and identify Steiner trees as answers, while ESTER focuses on text and entity search, and finds documents and entities as answers. The data graph used in our approach has more flexibility to represent information and find semantically-related answers for users, which also require more join operations. This graph representation and join operations are not the main focus of ESTER. (2) Our graph-partition-based technique presented in Section 4 is novel. (3) Our query-prediction technique to suggest complete queries presented in Section 5 is also novel.

Table 1: A publication database.

Authors

AID	Name
a1	Vagelis Hristidis
a2	S. Sudarshan
a3	Yannis Papakonstantinou
a4	Andrey Balmin
a5	Shashank Pandit
a6	Jeffrey Xu Yu
a7	Xuemin Lin
a8	Philip S. Yu
a9	Ian Felipe

Citations

PID	CitedPID
p1	p3
p2	p3
p3	p4
p4	p5
p5	p6
p6	p7
p7	p9
p8	p9

Author-Paper

AID	PID
a1	p1
a1	p3
a2	p2
a3	p1
a3	p3
a4	p4
a5	p5
a6	p6
a7	p7
a8	p8
a9	p9

Papers

PID	Title	Conf	Year
p1	DISCOVER: Keyword Search in Relational Databases	VLDB	2002
p2	Keyword Searching and Browsing in Databases using BANKS	ICDE	2002
p3	Efficient IR-Style Keyword Search over Relational Databases	VLDB	2003
p4	ObjectRank: Authority-Based Keyword Search in Databases	VLDB	2004
p5	Bidirectional Expansion For Keyword Search on Graph Databases	VLDB	2005
p6	Finding Top-k Min-Cost Connected Trees in Databases	ICDE	2007
p7	Spark: Top-k Keyword Query in Relational Databases	SIGMOD	2007
p8	BLINKS: Ranked Keyword Searches on Graphs	SIGMOD	2007
p9	Keyword Search on Spatial Databases	ICDE	2008

Notice that the work in [4] predicts complete queries for a prefix using frequent phrases in the data. Our probabilistic prediction approach is different since it treats the query as a set of keywords, and does not require the keywords to form a popular phrase. Thus our approach can predict likely queries even if they do not appear very often in the data as phrases. Other related studies include fuzzy type-ahead search on a set of records/documents [16], type-ahead search in XML data [21], and the work in [7].

2. PRELIMINARIES

Database Graph: A relational database can be modeled as a database graph $G = (V, E)$. Each tuple in the database corresponds to a vertex in G, and the vertex is associated with the keywords in the tuple. A foreign-key relationship from a tuple to another one corresponds to an edge between their vertices. The graph can be modeled as a directed or undirected graph. The graph can have weights, in which the weight of an edge represents the strength of the proximity relationship between the two tuples [6]. For simplicity, in this paper we consider undirected graphs, and all edges have the same weight. Our methods can be extended to directed graphs with other edge-weight functions. For example, consider a publication database with four tables. Table 1 shows part of the database, which includes 9 publications and 9 authors. Figure 2 is the corresponding database graph. For simplicity, we do not include the vertices for the tuples in the `Author-Paper` and `Citations` tables.

Steiner Trees as Answers to Keyword Queries: A query includes a set of keywords and asks for subgraphs with vertices that have these keywords. Consider a set of vertices $V' \subseteq V$ that include the query keywords. We want to find a subtree that contains all the vertices in V' (possibly other vertices in the graph as well). The following concept from [6] defines this intuition formally.

Definition 1. (Steiner Tree) *Given a graph $G = (V, E)$ and a set of vertices $V' \subseteq V$, a subtree T in G is a* Steiner tree of V' *if T covers all the vertices in V'.*

Often we are interested in a "smallest" Steiner tree, assuming we are given a function to compute the size of a tree, such as the summation of the weights of its edges. A Steiner tree with the smallest size is called a *minimum Steiner tree*. For simplicity, in this paper we use the number of edges in a tree as its size. For example, consider the graph in Figure 2. For the vertex set $\{a_1, a_3\}$, its Steiner trees include the trees of vertices $\{a_1, a_3, p_1\}$, $\{a_1, a_3, p_3\}$, $\{a_1, a_3, p_1, p_3\}$, etc. The first two are two minimum Steiner trees.

A query keyword can appear in multiple vertices in the graph, and we are interested in Steiner trees that include a vertex for every keyword as defined below.

Definition 2. (Group Steiner Tree) *Given a database graph $G = (V, E)$ and subsets of vertices $V_1, V_2, \ldots, V_n$ of V, a subtree T of G is a* group Steiner tree *of these subsets if T is a Steiner tree that contains a vertex from each subset V_i.*

Consider a query Q with two keywords {`Yu`, `sigmod`}. The set of vertices that contain the first keyword, denoted by $V_{\texttt{Yu}}$, is $\{a_6, a_8\}$. Similarly, $V_{\texttt{sigmod}} = \{p_7, p_8\}$. The subtree composed of vertices $\{a_8, p_8\}$ and the subtree composed of vertices $\{a_6, p_6, p_7\}$ are two group Steiner trees.

The problem of finding answers to a keyword query on a database can be translated into the problem of finding group Steiner trees in the database graph [6]. Often a user is not interested in a Steiner tree with a large diameter, which is the largest value of the shortest path between any two vertices in the tree. To address this issue, we introduce the concept of *δ-diameter Steiner tree*, which is a Steiner tree whose diameter is at most δ. In this paper, we want to find δ-diameter Steiner trees as answers to keyword queries. For example, consider the graph in Figure 2. For the query Q above, the top-2 minimum group Steiner trees are the subtree of vertices $\{a_8, p_8\}$ and the subtree of vertices $\{a_6, p_6, p_7\}$.

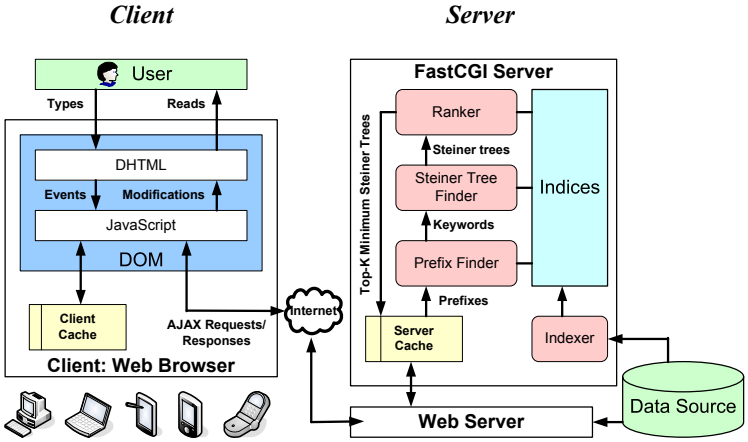


Figure 3: Tastier Architecture.

Type-Ahead Search: A Tastier system supports type-ahead search by computing answers to keyword queries as the user types in keywords. Figure 3 illustrates the architecture of such a system. Each keystroke that the user types could invoke a query, which includes the current string the user has typed in. The browser sends the query to the server, which computes and returns to the user the best answers ranked by their relevancy to the query. We treat

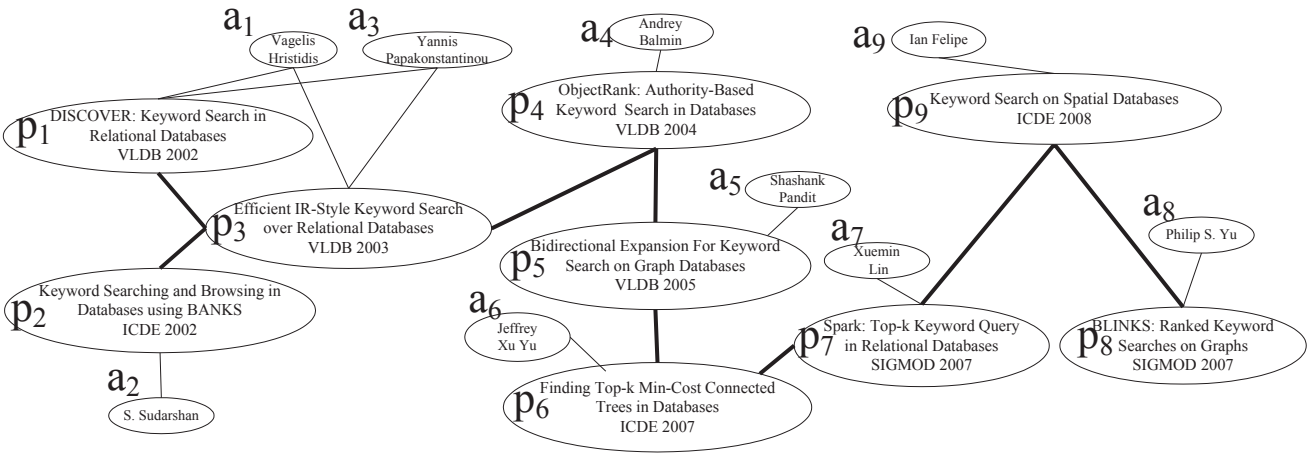


Figure 2: The database graph of the database tuples in Table 1.

every query keyword k_i in a query as a prefix condition, and consider those keywords in the database with k_i as a prefix, called the "complete keywords" of k_i. For instance, for a query keyword `sig`, its complete keywords can be `sigmod`, `sigcomm`, etc. We compute the δ-diameter Steiner trees that contain a complete word for each query keyword.[1] In our running example, suppose a user types in a query "`yu sig`". We identify "`sigmod`" as a complete word of "`sig`", and compute two 2-diameter Steiner trees: the subtree with vertices $\{a_8, p_8\}$ and the subtree with vertices $\{a_6, p_6, p_7\}$.

3. INDEXING AND SEARCH ALGORITHMS

In this section, we propose index structures and incremental algorithms to achieve a high interactive speed for type-ahead search in relational databases.

3.1 Type-Ahead Search on a Single Keyword

We first study how to support type-ahead search on the database graph where a user types in a single keyword. We use a trie to index the tokenized words in the database. Each word w corresponds to a unique path from the root of the trie to a leaf node. Each node on the path has a label of a character in w. For simplicity, we use a trie node and its corresponding string (composed of characters from the root to the node) interchangeably. For each leaf node, we assign a unique ID to the corresponding word in the alphabetical order. Each leaf node has an inverted list of IDs of vertices in the database graph that contain the word. Each trie node has a keyword range of $[L, U]$, where L and U are the minimum and maximal keyword ID in the subtrie of the node, respectively. It also maintains the size of the union of inverted lists of its descendant leaf nodes, which is exactly the number of vertices in the database graph that have the node string as a prefix. For instance, for the database graph in Figure 2, the trie for its tokenized words is shown in Figure 4. The word "`sigmod`", corresponding to trie node 16, has a keyword id 3, represented as "[3,3]" (in order to be consistent with the range representations in other trie nodes). Its inverted list includes graph vertices p_7 and p_8. Node 6 corresponds to a prefix string "`s`". It has a keyword range $[2, 4]$, since all the keywords of its descendants are within this range. Its size is 8, which is the number of graph vertices with the string "`s`" as a prefix.

[1]Our solutions generalize naturally to the case where only the last keyword is treated as a prefix condition, while the other keywords are treated as complete words.

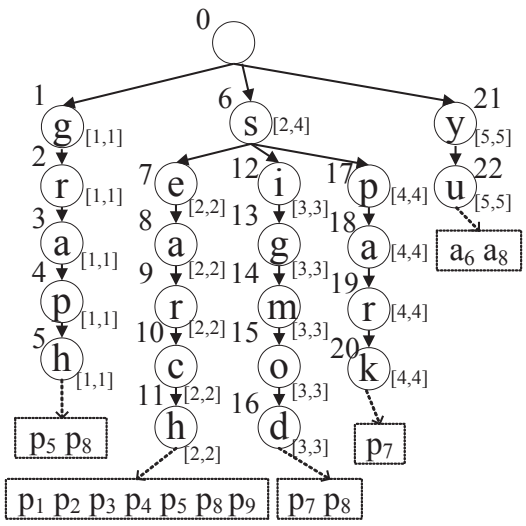


Figure 4: A (partial) trie of the words in Figure 2.

To answer a single-keyword query, we consider the trie node corresponding to the prefix. (If this node does not exist, the answer is empty.) We access its descendant leaf nodes, and retrieve the graph vertices from their inverted lists as the answers to the query. Whenever the user types in more letters, we traverse the trie downwards to compute the answers.

3.2 Multiple Keywords

Basic Algorithm: To support type-ahead search for a query with multiple keywords, we need to efficiently "join" the graph vertices of complete keywords for the query (partial) keywords. To achieve this goal, we propose an index structure called "δ-step forward index." For each graph ver-

tex v, for $0 \leq i \leq \delta$, the index has an "i-step forward list" that includes the keywords in the vertices whose shortest distance to the vertex v is i. The 0-step forward list consists of the keywords in v. Table 2 shows the δ-step forward index for some vertices in our running example. For the vertex a_6, its 0-step forward list contains keyword "`Yu`". Its 1-step forward list contains no keyword. Its 2-step forward list contains keywords "`graph`", "`sigmod`", "`search`", and "`spark`" (in vertices p_5 and p_7). Given a vertex, its δ-step forward index can be constructed on the fly by a breadth-first traversal in the graph from the vertex.

Table 2: δ-step forward index.

Vertex	Keyword ids in neighbors		
	0-step	1-step	2-step
p_5	1 2	–	3 4 5
p_6	–	1 2 3 4 5	–
p_7	3 4	2	1 5
p_8	1 2 3	5	4
a_6	5	–	1 2 3 4
a_8	5	1 2 3	–

Given a query with multiple keywords $Q = \{p_1, p_2, \ldots, p_n\}$, we first locate the trie nodes for the prefixes. We select the node of keyword $p \in Q$ with the minimum size (precomputed) of the union Φ of its leaf-node inverted lists, and compute this union list Φ. For each vertex v on Φ, we construct its δ-step forward index. For each keyword $p_i \neq p$ ($p_i \in Q$), we test whether the δ-step forward index of v includes a complete keyword of prefix p_i. This test can be done efficiently by using binary search based on the fact that the trie node of p_i has a range $[L_i, U_i]$ for the IDs of complete keywords of p_i. Specifically, for each j between 0 and δ, we do a binary search to check if the (sorted) list of keywords of v's j-step neighbors has a keyword within the range. If so, then a complete keyword of p_i appears in a neighbor of vertex v. If all the query keywords appear in neighbors of v within δ steps, then v can be expanded to a Steiner tree as an answer to the query. We construct a δ-diameter Steiner tree by visiting the neighbors of v. This step can be optimized if sufficient information was collected when we construct the δ-step forward index of vertex v. This step is repeated for each vertex on the union list Φ.

For example, assuming the user types in a query "`Yu s`" letter by letter, Figure 5 shows how to compute its answers incrementally. When the user types in a single keyword "`Yu`", we use the trie node of the keyword to find the vertices that include the keyword, i.e., vertices a_6 and a_8 (second column in the figure). When the user types in "`Yu s`", we first locate the trie nodes 22 and 6 for the two keywords (see Figure 4). As node 22 has the minimum union-list size, we compute its union list $\{a_6, a_8\}$. As shown in the third column in the figure, we construct the δ-step forward index as shown in Table 2. The keyword IDs with $p =$ "`s`" as a prefix are within the range [2,4]. For vertex a_6, we test whether its neighbors have a complete keyword in the range [2,4] by binary searching the lower bound 2 in the δ-step forward index. As a_6 contains the prefix, we construct the Steiner tree as follows. We visit a_6's only neighbor, p_6, which does not include a complete word of p, and add it into the tree. Next, we visit p_6's neighbor, p_5, and add it to the tree, since it has a complete keyword of p ("`Search`"). Thus we

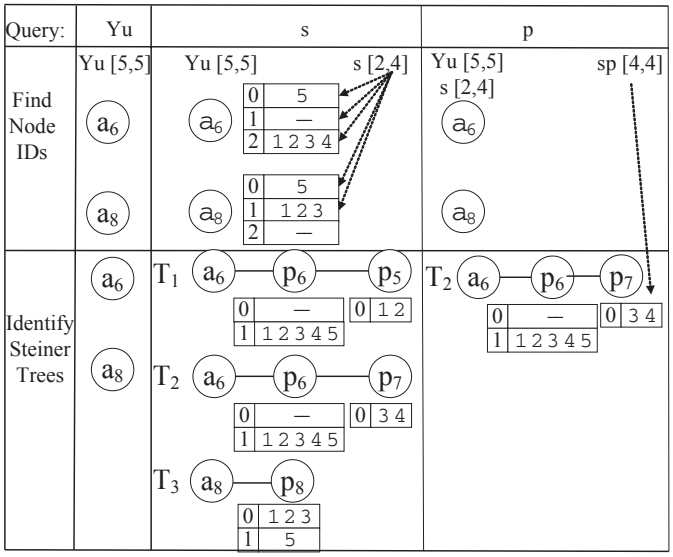


Figure 5: Type-ahead search for query "`Yu sp`".

have constructed a Steiner tree T_1 with vertices a_6, p_6, and p_5 as an answer to the query. Similarly, we can construct another Steiner tree T_2 for a_6, which consists of vertices a_6, p_6, and p_7. For vertex a_8, we construct a Steiner tree with vertices a_8 and p_8.

Incremental Computation: As the user types in more letters in the query, we can compute the answers incrementally by utilizing the results of earlier queries. Suppose a user has typed in a query with keywords p_1, p_2, ..., p_n in the same order. We have computed and cached the vertices whose neighbors contain the keywords (as prefixes) and their corresponding Steiner trees. Now the user modifies the query in one of the following ways. (1) The user types in one more non-space letter after "p_n" and submits a query $\{p_1, p_2, \ldots, p'_n\}$, where p'_n is a keyword obtained by appending a letter to p_n. In this case, for each cached Steiner tree, we check if it contains a complete keyword of the new prefix p'_n, and if so, we add it as an answer to the new query. (2) The user types in one more keyword p_{n+1} and submits the query $\{p_1, p_2, \ldots, p_n, p_{n+1}\}$. In this case, we locate the trie node of p_{n+1}, test whether the cached vertices contain a complete keyword of p_{n+1} using the keyword range of the node and the δ-step forward indices of the cached vertices, and identify the answers to the new query. (3) The user modifies the query arbitrarily. Suppose the user submits a new query $\{p_1, p_2, \ldots, p_i, p'_{i+1}, p'_{i+2}, \ldots\}$, which shares the first i keywords with the original query. In this case, we can use the cached results of the query $\{p_1, p_2, \ldots, p_i\}$ and find those that include a complete keyword for each new keyword p'_j ($j > i$).

In our running example, suppose the user types in another character "`p`" and submits a query "`Yu sp`". As shown in the last column of Figure 5, we test whether the three subtrees computed for the previous query contain a complete keyword for prefix "`sp`". The testing succeeds only for tree T_2 because of the keyword "`spark`" in vertex p_7, and we return T_2 as the answer to the new query.

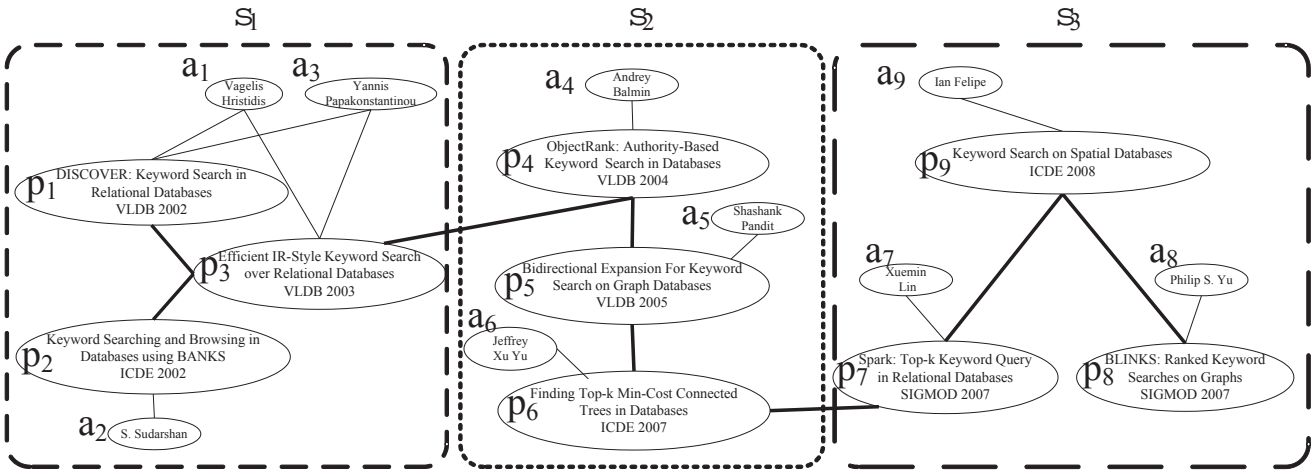


Figure 6: A graph partitioning with three subgraphs.

4. IMPROVING QUERY PERFORMANCE BY GRAPH PARTITIONING

We study how to improve performance of type-ahead search by graph partitioning. We first discuss how to do type-ahead search using a given partition of the database graph in Section 4.1. We then study how to partition the graph in Section 4.2.

4.1 Type-Ahead Search on a Partitioned Graph

Our main idea is to partition the database graph into (overlapping) subgraphs, and assign unique ids to them. On each leaf node w of the trie of the database keywords, its inverted list stores the ids of the subgraphs with vertices that include the keyword w, instead of storing vertex ids. We maintain a forward list for each subgraph, which includes the keywords within the subgraph.

A keyword query is answered in two steps. In step 1, we find ids of subgraphs that contain the query keywords. We identify the keyword with the shortest union list l_s of inverted lists. We use the keyword range of the other keywords to probe on the forward list of each subgraph on l_s (similar to the method described in Section 3.2). Each subgraph that passes all the probes is a candidate subgraph to be processed in the next step. In step 2, we find δ-diameter Steiner trees inside each candidate subgraph. We construct δ-depth forward lists for vertices in each candidate subgraph, using the method described in Section 3. Note that we can also do incremental search as discussed in Section 3.

For instance, consider the database graph in Figure 2. Suppose we partition it into three subgraphs as illustrated in Figure 6. We construct a trie similar to Figure 4, in which each leaf node has an inverted list of subgraphs that include the keyword of the leaf node. Consider a query "`yu sig`". We find answers from subgraphs s_2 and s_3, and identify 2-diameter Steiner trees: the subtree of vertices a_8 and p_8, the subtree of vertices p_6, a_6, and p_5. Incremental computation to support type-ahead search can be done similarly.

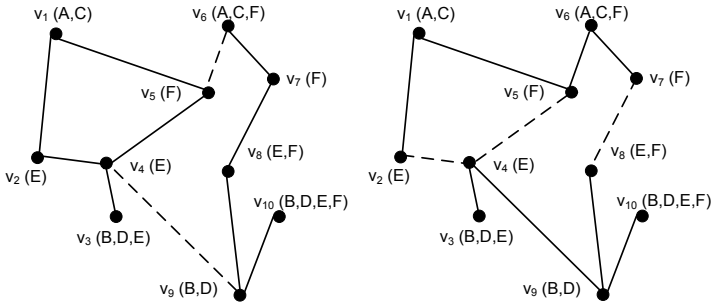
An answer (a δ-diameter Steiner tree) to a query may include edges in different subgraphs. For example, when answering query "`min-cost sigmod`" using the subgraphs in Figure 6, we cannot find any answer, although p_6 and p_7 are connected by an edge. To make sure our approach based on graph partitioning can always find all answers, we expand each subgraph to include, for each vertex in the subgraph, its neighbors within δ-steps, and build index structures on these expanded subgraphs. For example, if $\delta = 1$, we add vertex p_4 into subgraph s_1, add vertices p_3 and p_7 into subgraph s_2, and add p_6 into subgraph s_3. In the rest of the section, we use "subgraphs" to refer to these expanded subgraphs.

4.2 High-Quality Graph Partitioning

A commonly-used method to partition a graph is to have subgraphs with similar sizes and few edges across the subgraphs. Formally, for a graph $G = (V, E)$ and an integer κ, we want to partition V into κ disjoint vertex subsets $V_1, V_2, \ldots, V_\kappa$, such that the number of edges with endpoints in different subsets is minimized. In the general setting where both vertices and edges are weighted, the problem becomes dividing G into κ disjoint subgraphs with similar weights (which is the summation of the weights of vertices), and the cost of the edge cuts is minimized, where the size of a cut is the summation of the weights of the edges in the cut. Graph partitioning is known to be an NP-complete problem [18].

The following example shows that a traditional method with the above optimization goal may not produce a good database-graph partition in our problem setting. Figure 7 shows a graph with 10 vertices v_1, v_2, ..., v_{10}. Each vertex v_i has a list of its keywords in parentheses. For example, the list of keywords for vertex v_3 is B, D, and E. Figure 7(a) shows one graph partition using a traditional method to minimize the number of edges across subgraphs. It has two subgraphs: $\{v_1, v_2, v_3, v_4, v_5\}$ and $\{v_6, v_7, v_8, v_9, v_{10}\}$. (For simplicity, we represent a subgraph using its set of vertices.) After extending each subgraph by assuming $\delta = 1$, the two subgraphs become $S_1 = \{v_1, v_2, v_3, v_4, v_5, v_6, v_9\}$ and $S_2 = \{v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$. Every keyword in $\{A, B, C, D, E, F\}$ has the same inverted list: $\{S_1, S_2\}$. As a consequence, for a query with any of these keywords, we need to search within both subgraphs. Figure 7(b) shows a different graph partition that can produce more efficient

index structures. In this partition there are three edges between the two expanded subgraphs: $S_3 = \{v_1, v_2, v_4, v_5, v_6, v_7, v_8\}$ and $S_4 = \{v_2, v_3, v_4, v_5, v_7, v_8, v_9, v_{10}\}$. The inverted lists for keywords A and C are $\{S_3\}$, those of B and D are $\{S_4\}$, and those for E and F are $\{S_3, S_4\}$. As the inverted lists become shorter, fewer subgraphs need to be accessed in order to answer keyword queries.



(*a*) Partition 1 (*b*) Partition 2

Figure 7: Two graph partitions. Dashed edges are between subgraphs.

We develop a keyword-sensitive graph-partitioning method that produces index structures to support efficient type-ahead search. The algorithm considers the keywords in the vertices. Its goal is to minimize the number of keywords shared by different subgraphs, since shared keywords will cause subgraph ids to appear on multiple inverted lists on the trie. We use a hypergraph to represent the shared keywords by vertices. For a database graph $G = (V, E)$, we construct a hypergraph $G_h = (V_h, E_h)$ as follows. The set of vertices V_h is the same as V. We construct the hyperedges in E_h in two steps. In the first step, for each edge e in E, we add a corresponding hyperedge in E_h that includes the two vertices of e. The weight of this hyperedge is the total number of keywords on the vertices within δ-steps of the two vertices of e. The intuition is that, if this hyperedge is deleted in a partition, the keywords of the neighboring vertices will be shared by the resulting subgraphs. In the second step, for each keyword, we add a hyperedge that includes all the vertices that include this keyword. The weight of this hyperedge is 1, since if this hyperedge is deleted in a partition, the corresponding keyword will be shared by the subgraphs. Figure 8 shows the hypergraph constructed from the graph in Figure 7. Hyperedges from keywords are drawn using rectangles with keyword labels. For example, the hyperedge for keyword E connects vertices v_2, v_3, v_4, and v_8. Hyperedges from the original edges are shown in lines.

We partition the weighted hypergraph using an existing hypergraph-partitioning method (e.g., [19]), with the goal of minimizing the total weight of the deleted hyperedges. For instance, a multilevel approach is to coarsen the graph by merging vertices in pairs step by step, generating small hypergraphs, and partition the smallest hypergraph using heuristic or random methods. The partition is "projected" back to the original hypergraph level by level, with proper partition refinement involved during each uncoarsening step. After computing a partition of the hypergraph G_h, we derive a partition on the original database graph G by deleting the edges whose corresponding hyperedges have been deleted

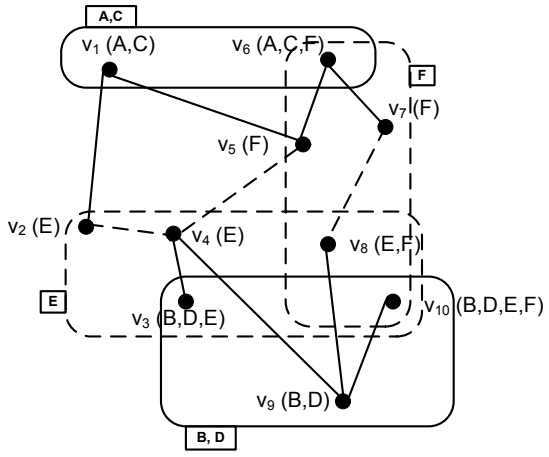


Figure 8: Hypergraph partitioning.

in the hypergraph partition. For instance, Figure 8 also shows a hypergraph partition with two sub-hypergraphs. Dashed lines and dashed rectangles are the deleted hyperedges. This partition minimizes the total weight of deleted hyperedges. This hypergraph partition produces the graph partition shown in Figure 7(b).

Deciding Number of Subgraphs: The number of subgraphs κ in a graph partition can greatly affect the query performance on the generated index structures. As we increase κ, there will be more subgraphs, causing the inverted lists on the trie to become longer. Thus it will take more time to access these lists to find candidate subgraphs. On the other hand, the average size of subgraphs becomes smaller, which can reduce the search time within each candidate subgraph. We have quantitatively analyzed the effect of κ on the overall query performance. We omit the analysis due to limited space.

5. IMPROVING PERFORMANCE BY QUERY PREDICTION

In this section we study how to improve query performance by predicting complete keywords for a partial keyword, and running the most likely predicted complete queries on the database graph to compute answers, before the user finishes typing the complete keyword. For instance, suppose a user types in a query "`similarity se`", where the first keyword has been typed in completely, and "`se`" is a partial keyword. If we can predict several possible queries with high probabilities, such as "`similarity search`", we can suggest these queries to the user, and run them on the database to compute answers. Figure 9 shows a screenshot in which the system predicts top-8 complete queries and displays the top-4 answers to the predicted queries. In this section, we focus on the case where only the last keyword is treated as a prefix. The solution generalizes to the case where each keyword is treated as a prefix, and we predict complete keywords for each of them, assuming the user can modify an arbitrary keyword if he/she does not see the intended complete keyword in the predicted queries.

Notice that the method presented in Section 3 essentially considers all possible complete keywords for a prefix keyword

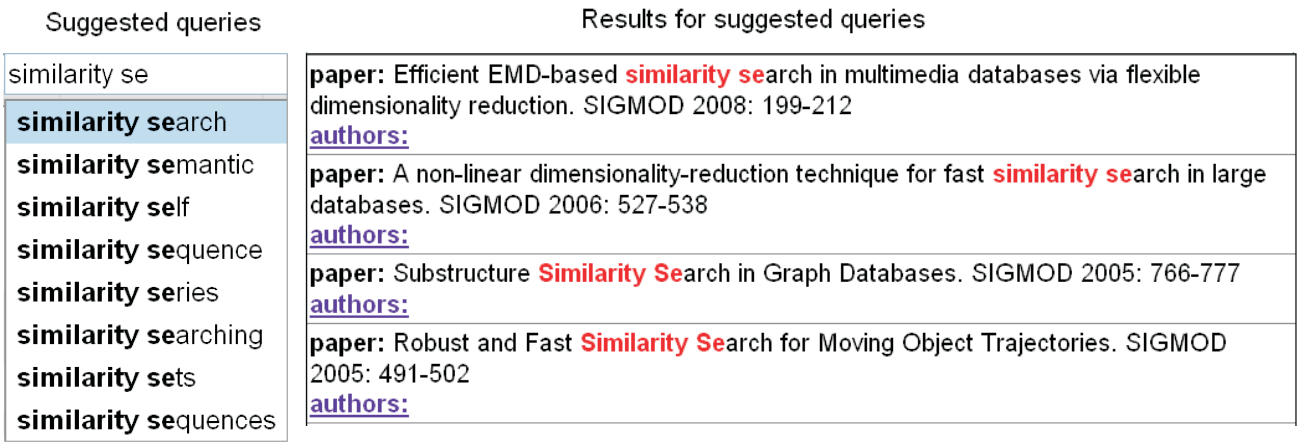


Figure 9: Screenshot of predicated query and answers in TASTIER.

and computes answers to these queries. On the contrary, the approach presented in this section efficiently predicts several possible complete keywords, and runs the corresponding queries. Thus this new approach can compute answers more efficiently. In case the predicted queries are not what the user wanted, then as the user types in more letters, we will dynamically change the predicted queries. After the user finishes typing in a keyword completely, there is no ambiguity about the user's intention, thus we can still compute the answers to the complete query. In this way, our approach can improve query performance while it still guarantees to help users find the correct answers.

5.1 Query-Prediction Model

Given a query with multiple keywords, among the complete query keywords of the last keyword, we want to predict the most likely complete keywords given the other keywords in the query. Notice that this prediction task is different from prediction in traditional "autocomplete" applications, in which a query with multiple keywords is often treated as a single string. These applications can only find records that match this entire string. For instance, consider the search box on Apple.com, which allows autocomplete search on Apple products. Although a keyword query "`itunes`" can find a record "`itunes wi-fi music store`," a query with keywords "`itunes music`" cannot find this record (as of November 2008), because these two keywords appear at different places in the record. Our approach allows the keywords to appear at different places in a tuple or Steiner tree.

Given a single keyword p, among the words with p as a prefix, we predict the top-ℓ words with the highest probabilities, for a constant ℓ. The probability of a word w is computed as follows:

$$Pr(w) = \frac{N_w}{N}, \tag{1}$$

where N is the number of vertices in the database graph and N_w is the number of vertices that contain word w. For example, for the database graph in Figure 2, there are 18 vertices and 3 vertices containing "`relation`". Thus we have $Pr(\texttt{relation}) = \frac{3}{18}$.

Suppose a user has typed in a query with complete keywords $\{k_1, k_2, \ldots, k_n\}$, after which the user types in another prefix keyword p_{n+1}. We use $\{k_1, k_2, \ldots, k_n\}$ to compute the probability of a complete keyword k_{n+1} of prefix p_{n+1}, i.e., $Pr(k_{n+1}|k_1, k_2, \ldots, k_n)$. We use a Bayesian network to compute this probability. In a Bayesian network, if two nodes x and y are "d-separated" by a third node z, then the corresponding variables of x and y are independent given the variable of z [30]. Suppose query keywords form a Bayesian network, and the nodes of k_1, k_2, ..., k_n are d-separated by the node of k_{n+1}. Then we have $Pr(k_i|k_j, k_{n+1}) = Pr(k_i|k_{n+1})$ for $i \neq j$. We can show

$$Pr(k_{n+1}|k_1, k_2, \ldots, k_n) = \frac{Pr(k_1, k_2, \ldots, k_n, k_{n+1})}{Pr(k_1, k_2, \ldots, k_n)}$$
$$= \frac{Pr(k_{n+1}) * Pr(k_1|k_{n+1}) * Pr(k_2|k_{n+1}) \ldots Pr(k_n|k_{n+1})}{Pr(k_1) * Pr(k_2|k_1) * Pr(k_3|k_1, k_2) \ldots Pr(k_n|k_1, \ldots, k_{n-1})}. \tag{2}$$

Notice the probabilities $Pr(k_3|k_1, k_2)$, $Pr(k_4|k_1, k_2, k_3)$, ..., $Pr(k_n|k_1, \ldots, k_{n-1})$ can be computed and cached in earlier predictions as the user types in these keywords. For $Pr(k_1|k_{n+1})$, $Pr(k_2|k_{n+1}), \ldots$, $Pr(k_n|k_{n+1})$, we can compute and store them offline. That is, for every two words w_i and w_j in the databases, we keep their conditional probabilities $Pr(w_i|w_j)$ and $Pr(w_j|w_i)$:

$$Pr(w_i|w_j) = \frac{T_{(w_i,w_j)}}{T_{w_j}}, \tag{3}$$

where T_{w_i} is the number of δ-diameter trees that contain word w_i, and $T_{(w_i,w_j)}$ is the number of δ-diameter trees that contain both w_i and w_j.[2] For example, for the graph in Figure 2, if $\delta = 1$, there are 18 1-diameter trees containing "`keyword`" and 16 1-diameter trees containing both "`keyword`" and "`search`". We have $Pr(\texttt{search|keyword}) = \frac{16}{18}$.

5.2 Two-Tier Trie for Efficient Prediction

To efficiently predict complete queries, we construct a two-tier trie. In the first tier, we construct a trie for the keywords in the database. Each leaf node has a probability of the word in the database. For the internal node, we maintain the top-ℓ

[2]For each vertex, we construct a δ-diameter tree by adding its neighbors within δ steps using a breadth-first traversal from the vertex.

words of its complete keywords (i.e., descendant leaf nodes) with the highest probabilities. In the second tier, for each leaf node w_1 in the trie, we construct a trie for the words in the δ-diameter trees of word w_1, and add the trie to the leaf node w_1. For each node in the second tier, we maintain the number of complete words under the node. A leaf node of keyword w_2 in the second tier under the leaf node w_1 also stores the conditional probability $Pr(w_2|w_1)$. Figure 10 illustrates an example, in which we use radix (compact) tries. Node "`search`" has a probability of $\frac{7}{18}$. We keep a top-1 complete word for node "`s`" (i.e., "`search`"). The node "`r`" under the node "`search`" ("`search+r`" in short) has two words in its descendants. The probability of "`search+relation`", $\big(Pr(\texttt{relation}|\texttt{search})\big)$, is $\frac{9}{16}$.

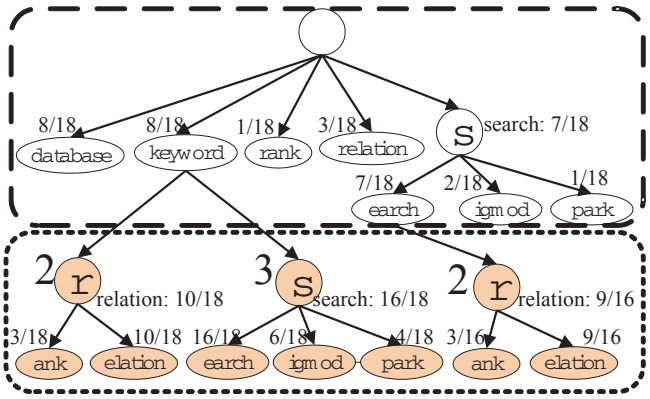


Figure 10: Two-tier trie.

This index structure stores information about the $Pr(w)$ probability in Equation 1 and the $Pr(k_i|k_{n+1})$ probabilities in Equation 2. We use this index structure to predict complete keywords of the last keyword p_{n+1} given other keywords $\{k_1, k_2, \ldots, k_n\}$. A naive way is the following. We use the trie of the first tier to find all the complete keywords of p_{n+1}. For each of them k_{n+1}, we use Equation 2 to compute the conditional probability $Pr(k_{n+1}|k_1, k_2, \ldots, k_n)$ using the conditional probabilities cached earlier and the conditional probabilities stored on the two-tier structure. We select the top-ℓ complete keywords with the highest probabilities.

We want to reduce the number of complete keywords whose conditional probabilities need to be computed. Notice that each complete keyword with a non-zero conditional probability has to appear in the trie of the second tier under the leaf node of each keyword k_i. Based on this observation, for each $1 \leq i \leq n$, we locate the trie node "$k_i + p_{n+1}$", i.e., node p_{n+1} under node k_i. We retrieve the set of complete keywords of this node. We only need to consider the intersection of these sets of complete keywords for the keywords $k_1, \ldots, k_n$.

For instance, suppose a user types in a query "`keyword search r`" letter by letter. We predict queries as follows. (1) When the user finishes typing the query "`key`", we locate the trie node and return the complete query "`keyword`" based on the information on the trie node. (2) When the user finishes typing the query "`keyword s`", we predict the complete query "`keyword search`" based on the information on trie node "`keyword+s`". (3) When the user types in the query "`keyword search r`", we locate the trie nodes "`keyword+r`" and "`search+r`", and select node "`keyword+r`" with the minimum number of words and identify complete words "`rank`" and "`relation`". We intersect their sets of words in their leaf nodes, and find two possible keywords "`rank`" and "`relation`". We compute their conditional probabilities using Equation 2.

In general, among the top-ℓ predicted keyword queries, we choose the predicted queries following their probabilities (highest first), and compute their answers. We terminate the process once we have found enough δ-diameter Steiner trees. If there are not enough answers, we can use previous methods in Section 4 to compute answers.

6. EXPERIMENTAL STUDY

We have conducted an experimental evaluation of the developed techniques on two real data sets: "DBLP"[3] and "IMDB"[4]. DBLP included about one million computer science publication records as described in Table 1. The original data size of DBLP was about 470 MB. IMDB had three tables "`user`", "`movie`", and "`ratings`". It contained approximately one million anonymous ratings of 3900 movies made by 6040 users. We set $\delta = 3$ for all the experiments.

We implemented the backend server using FastCGI in C++, compiled with a GNU compiler. All the experiments were run on a Linux machine with an Intel Core 2 Quad processor X5450 3.00GHz and 4 GB memory.

6.1 Search Efficiency without Partitioning

We evaluated the search efficiency of Tastier without graph partitioning. We constructed the trie structure and built the forward index for each graph vertex and the inverted index for each keyword. Table 3 summarizes the index costs on the two datasets.

Table 3: Data sets and index costs.

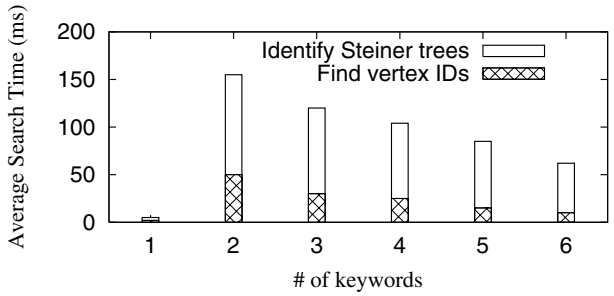
Data set	DBLP	IMDB
Original data size	470 MB	50 MB
# of distinct keywords	392K	14K
Index-construction Time	35 seconds	6 seconds
Trie size	14 MB	2 MB
Inverted-list size	54 MB	5.5 MB
Forward-list size	45 MB	4.8 MB

We generated 100 queries by randomly selecting vertices from the database graph, and choosing keywords from each vertex. We evaluated the average time of each keystroke to answer a query. Figure 11 illustrates the experimental results. We observe that for queries with two keywords, Tastier spent much more time than queries with a single keyword. This is because we need to find the vertex ids and identify the Steiner trees on the fly. When queries had more than two keywords, Tastier achieved a high efficiency as we can use the cached information to do incremental search.

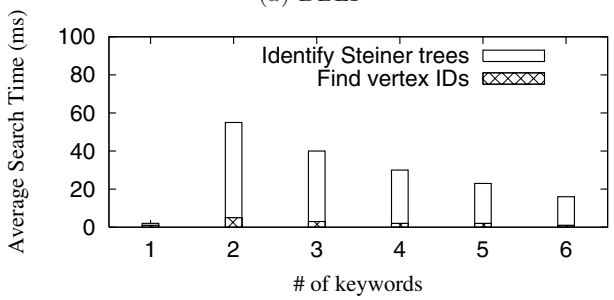
6.2 Effect of Graph Partitioning

We evaluated the effect of graph partitioning. We used a tool called hMETIS[5] for graph partitioning. We constructed the trie structure, and built the forward index for each vertex and the inverted index for each keyword on top of subgraphs and vertexes. Figure 12 illustrates the index cost for

[3]http://dblp.uni-trier.de/xml/
[4]http://www.imdb.com/interfaces
[5]http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview



(a) DBLP



(b) IMDB

Figure 11: Search efficiency without partitioning.

graph partitioning. We observe that the index size is a bit larger than that without partitioning. When we increased the number of subgraphs, the index size increased slightly.

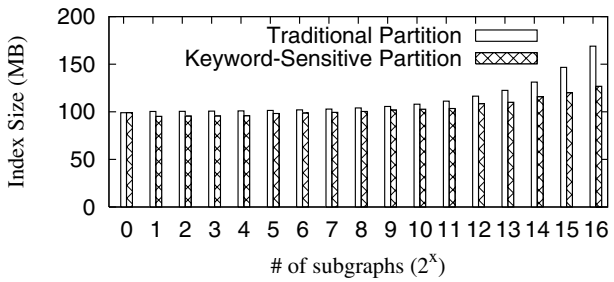


Figure 12: Index size vs # of subgraphs on DBLP.

We evaluated search efficiency. We randomly generated 100 queries and evaluated the average time of each keystroke to answer a query. Figure 13 gives the experimental results. We observe that, for a very small number of subgraphs, graph partitioning cannot improve search efficiency as it is hard to prune large subgraphs. For a very large number of subgraphs, the inverted lists on top of subgraphs became larger, and thus they could not improve search efficiency. We find that when the number of subgraphs was about 10K, the graph-partition-based methods achieved the best performance. Moreover, our keyword-sensitive-based method outperformed the traditional one, as it reduced the inverted-list size and could prune more irrelevant subgraphs.

6.3 Effect of Query Prediction

We evaluated the effect of query prediction. Besides the forward index and inverted index, we also constructed a two-tier trie structure which was about 200 MB. Note that the two-tier trie size depends on the number of keywords in the database, but not the number of tuples.

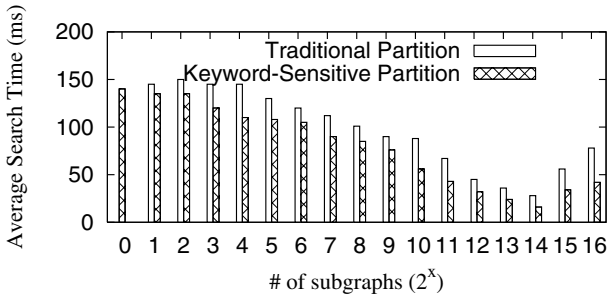


Figure 13: Efficiency vs # of subgraphs on DBLP.

Quality of Query Prediction: We evaluated the result quality of query prediction. We first used the reciprocal rank to evaluate the result quality of prediction. For a query, the reciprocal rank is the ratio between 1 and the rank at which the first correct answer is returned; or 0 if no correct answer is returned. For example, if the first relevant query is ranked at 2, then the reciprocal rank is $\frac{1}{2}$. We randomly selected 100 queries with two to then keywords on each dataset, and used them as the complete correct queries. We then composed prefix queries of the first three letters of each keyword in the complete queries. We evaluated the average reciprocal rank. The average reciprocal rank of the 100 queries on DBLP was 0.91, and that of IMDB was 0.95. Table 4 gives several sample queries.

Table 4: Reciprocal rank.

(a) DBLP

Complete Query	Partial Query	reciprocal rank
keyword search database	key sea dat	1
approximate search sigmod	app sea sig	$\frac{1}{2}$
similarity search icde	sim sea icd	1
christos faloutsos mining	chr fal min	1

(b) IMDB

Complete Query	Partial Query	reciprocal rank
children corn	chi cor	1
star wars	sta war	1
slumber party massacre	slu par mas	1
toxic venger comedy	tox ven com	1

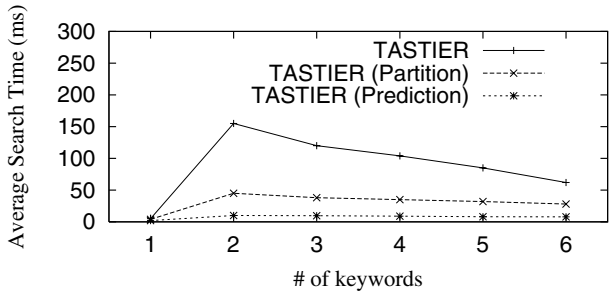
We then used top-k precision to evaluate the result quality. We used the graph-partition-based method and the query-prediction-based method to respectively identify the top-k results. For the latter method, we predicted top-10 complete queries for each partial query and identified answers on top of the 10 complete queries. Given a query, the top-k precision of the prediction-based method is the ratio between the number of same answers from the two methods and k. Table 5 gives the experimental results on the two datasets. We observe that the prediction-based method achieved good quality as we can accurately predict the most relevant complete queries. When we increased k, the top-k precision decreased as there may not be enough top-k answers for the predicted queries.

Efficiency of Prediction: We randomly generated 100 queries on the two datasets. We evaluated the average time of each keystroke to answer a query. We partitioned DBLP into 10,000 subgraphs and IMDB into 1,000 subgraphs. Fig-

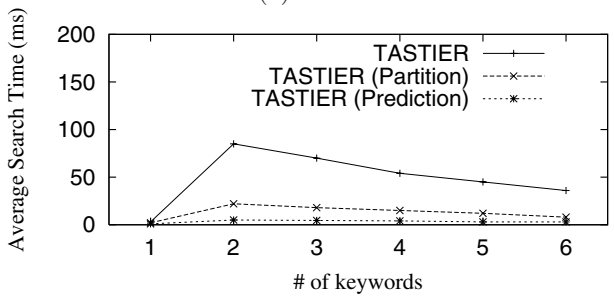
Table 5: Top-k precision.

k	1	2	5	10	20	50	100
DBLP	96%	97%	95%	96%	94%	92%	90%
IMDB	99%	98%	99%	97%	95%	93%	92%

ure 14 shows the results. The prediction-based method achieved the best performance, as it predicted the most relevant complete queries and used them to identify the answers. The other two methods needed to consider all possible complete queries. Moreover, the graph-partition-based method also improved the efficiency as it shortened the inverted lists on subgraphs and pruned many irrelevant subgraphs.



(a) DBLP



(b) IMDB

Figure 14: Search efficiency for different approaches.

6.4 Scalability

We evaluated the scalability of our algorithms on the DBLP dataset. We first evaluated the index size. Figure 15 shows how the index size increased as the number of database tuples increased. It shows that the sizes of trie, inverted index, forward index, and partition-based inverted index increased linearly or sublinearly. Note that the inverted index on subgraphs was smaller than that of the inverted index on vertices.

We measured the query performance as the data size increased. We used 100 queries with multiple keywords, each of which asked for 10 best answers. We generated queries with two to then keywords, and measured the average time of each keystroke. Figure 16 shows that our algorithms can answer such queries very efficiently. We obverse that TASTIER (Prediction) outperformed TASTIER (Partition), which was better than TASTIER. This is because TASTIER (Partition) reduced the inverted-index size and can prune many subgraphs. TASTIER (Prediction) predicted the highly relevant queries and did not need to identify the Steiner trees for all complete queries. For instance, when the data set had one million vertices, a query was answered within 20 ms for TASTIER (Prediction), within 50 ms for TASTIER (Partition), and within 140 ms for TASTIER.

6.5 Round-Trip Time

The round-trip time of the type-ahead search consists of three components: server processing time, network time, and JavaScript running time. Different locations in the world could have different network delays to our server. We measured the time cost for a sample query "`2008 keyword search sig`" on the DBLP dataset from four locations around the world: Location A (in the same country with the server), Location B (in different countries and the same continent with the server), and Location C (in different countries with the server). We used the prediction-based method. Figure 17 shows the results. We can see that the server running time was less than 1/4 of the total round-trip time. JavaScript took around 50 ms to execute. The relative low speed at some locations was mainly due to the network delay. For all the users from different locations, the total round-trip time for a query was always below 220 ms, and all of them enjoyed an interactive interface.

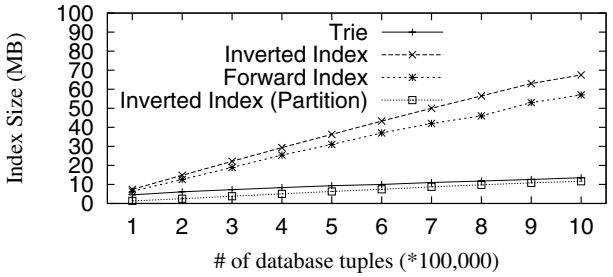


Figure 15: Scalability: index size.

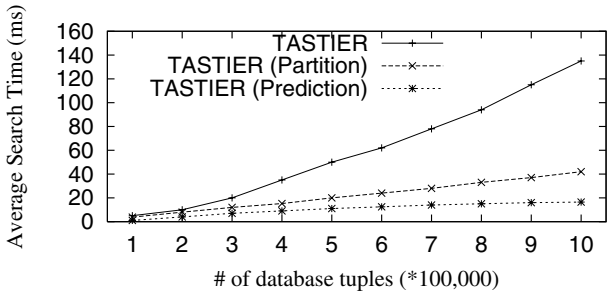


Figure 16: Scalability: average search time.

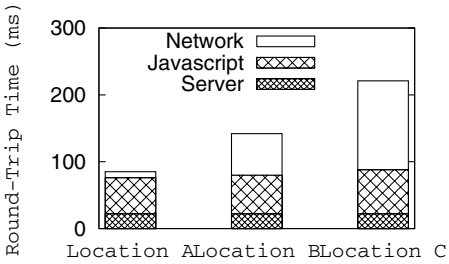


Figure 17: Round-trip time.

7. CONCLUSION

In this paper, we studied a new information-access paradigm that supports type-ahead search in relational databases with

multiple tables. We proposed efficient index structures and algorithms for incrementally computing answers to queries in order to achieve an interactive speed on large data sets. We developed a graph-partition-based method and a query-prediction technique to improve search efficiency. We have conducted a thorough experimental study of the algorithms. The results proved the efficiency and practicality of this new computing paradigm.

Acknowledgements

Special thanks go to Jiannan Wang who helped with query prediction and the experiments. The work was partially supported by the National Science Foundation of the US under the award No. IIS-0742960, the National Science Foundation of China under the grant No. 60828004, the National Natural Science Foundation of China under Grant No. 60873065, and the National High Technology Development 863 Program of China under Grant No. 2007AA01Z152, the National Grand Fundamental Research 973 Program of China under Grant No. 2006CB303103, and 2008 HP Labs Innovation Research Program.

8. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.

[2] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *VLDB*, pages 564–575, 2004.

[3] H. Bast, A. Chitea, F. M. Suchanek, and I. Weber. Ester: efficient search on text, entities, and relations. In *SIGIR*, pages 671–678, 2007.

[4] H. Bast and I. Weber. Type less, find more: fast autocompletion search with a succinct index. In *SIGIR*, pages 364–371, 2006.

[5] H. Bast and I. Weber. The completesearch engine: Interactive, efficient, and towards ir& db integration. In *CIDR*, pages 88–95, 2007.

[6] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.

[7] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *SIGMOD*, 2009.

[8] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. Xsearch: A semantic search engine for xml. In *VLDB*, pages 45–56, 2003.

[9] B. D. Davison and H. Hirsh. Predicting sequences of user actions. In *AAAI/ICML Workshop on Predicting the Future*, 1998.

[10] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, pages 836–845, 2007.

[11] K. Grabski and T. Scheffer. Sentence completion. In *SIGIR*, pages 433–439, 2004.

[12] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *SIGMOD Conference*, pages 16–27, 2003.

[13] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD Conference*, pages 305–316, 2007.

[14] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

[15] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[16] S. Ji, G. Li, C. Li, and J. Feng. Interative fuzzy keyword search. In *WWW 2009*, 2009.

[17] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[18] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[19] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. In *DAC*, pages 343–348, 1999.

[20] K. Kukich. Techniques for automatically correcting words in text. *ACM Comput. Surv.*, 24(4):377–439, 1992.

[21] G. Li, J. Feng, and L. Zhou. Interactive search in xml data. In *WWW*, 2009.

[22] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD Conference*, pages 903–914, 2008.

[23] G. Li, X. Zhou, J. Feng, and J. Wang. Progressive keyword search in relational databases. In *ICDE*, 2009.

[24] Y. Li, C. Yu, and H. V. Jagadish. Schema-free xquery. In *VLDB*, pages 72–83, 2004.

[25] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD Conference*, pages 563–574, 2006.

[26] Z. Liu and Y. Chen. Identifying meaningful return information for xml keyword search. In *SIGMOD Conference*, pages 329–340, 2007.

[27] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD Conference*, pages 115–126, 2007.

[28] H. Motoda and K. Yoshida. Machine learning techniques to make computers easier to use. *Artif. Intell.*, 103(1-2):295–321, 1998.

[29] A. Nandi and H. V. Jagadish. Effective phrase prediction. In *VLDB*, pages 219–230, 2007.

[30] J. Pearl. Evidential reasoning using stochastic simulation of causal models. *Artif. Intell.*, 32(2):245–257, 1987.

[31] C. Sun, C. Y. Chan, and A. K. Goenka. Multiway slca-based keyword search in xml data. In *WWW*, pages 1043–1052, 2007.

[32] H. E. Williams, J. Zobel, and D. Bahle. Fast phrase querying with combined indexes. *ACM Trans. Inf. Syst.*, 22(4):573–594, 2004.

[33] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD Conference*, pages 537–538, 2005.

[34] Y. Xu and Y. Papakonstantinou. Efficient LCA based keyword search in XML data. In *EDBT*, pages 535–546, 2008.