Bulletin of the Technical Committee on

Data Engineering

March 2010 Vol. 33 No. 1 IEEE Computer Society

Letters

Special Issue on Keyword Search

Conference and Journal Notices

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TC on Data Engineering web page is
http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at
http://tab.computer.org/tcde/bull_about.html.

Distribution
IEEE Computer Society
1730 Massachusetts Avenue
Washington, D.C. 20036-1992
(202) 371-1013
jw.daniel@computer.org

Letter from the Editor-in-Chief

About the Bulletin

Bulletin Editors

Bulletin editors have two year appointments in which they take responsibility for two issues, one per year. The result is that at two year intervals, prior editors "retire" and new editors are appointed. This transition takes place in March of even numbered years. So now is the time to thank the editors whose terms are ending with the publication of the current issue. These editors (still listed on the inside front cover) are Sihem Amer-Yahia, Beng Chin Ooi, Jianwen Su, and Vassilis Tsotras. Each has produced two issues on topics of current interest to the database community. It is important for me to emphasize that the topics are chosen by these editors. I play, at most, a modest role in avoiding obvious overlap and trying to provide broad coverage. The success of the Bulletin depends in an essential way, on the efforts of the editors. So I want to thank them for having provided our readers with two years of special topic issues containing high quality papers focused on technology that is actively being investigated.

Bulletin Publication

I have dropped all the "draft" postscript versions from the Bulletin web site. Essentially everyone has web access that is sufficiently reliable and capable that these issues can now be dropped. I continue to weigh the dropping of postscript versions in an effort to further simplify the web site. Please provide input to this process, especially if you would like to continue accessing postscript versions.

ICDE: International Conference on Data Engineering

In 2011, ICDE will be in Hanover, Germany. ICDE is the signature conference of the IEEE Technical Committee on Data Engineering. This issue contains a "Call for Papers" for ICDE'11 on the inside back cover. Take note of the deadline for paper submission (July 16, 2010). I would encourage you all to submit papers to this high quality conference and hope to see you in Hanover in the spring of 2011.

The Current Issue

Keyword search has been and continues as a topic of great research interest and clear commercial importance. There are a large number of technologies being pursued, from trying to make search produce more relevant information, to including structured data, to making search more immediately responsive, and the list goes on. Indeed, no single issue can capture everything. The current issue includes survey articles as well as ongoing research articles and provides excellent coverage for this field. I want to thank Beng Chin Ooi for a fine job of assembling the issue and, as always is needed, handling the difficulties involved in publication of the issue. Readers will be well rewarded with this issue and its fine overview of keyword search.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editor

The simple keyword-based query interface has to a great extent contributed to the wide acceptance of the Internet and its proliferation of user-contributed contents. The interface allows users to query vast collections of information freely, and hence improves the usability of the technology. Over the past two decades, database systems have made great advances in terms of performance, scalability and fault tolerance. They can now process a huge number of concurrent complex queries efficiently over enormous and diverse data sets. Naturally, the next challenge is to improve the usability of database systems in terms of providing search and query interfaces more than structured query languages can, as well as query by examples and query by forms that prevail.

Coupled with the increasing volume of text-based data, the keyword-based query mechanism becomes a natural and effective means for users to interact with databases. However, outstanding issues remain to be addressed before we can have a paradigm shift that allows users to query a database meaningfully without having any knowledge about the underlying data repository. In this issue, we have eight papers that provide different aspects and insights of keyword-based search and retrieval methods. I thank the authors for their contributions.

One of the earlier approaches to keyword-based retrieval from structured data is to identify the connection of data items containing keywords. Recent work has attempted to go beyond such approaches, and Chakrabarti, Sarawagi and Sudarshan provide a survey of recent work on adding structure to keyword search.

The large collections of user-generated content in community systems enable construction of large knowledge bases, and these are typically represented in the RDF model. Elbassuoni et al. give an overview of recent and ongoing work on ranked retrieval of RDF data with keyword-augmented structured queries.

Strings are a common data type in many applications, ranging from relational databases, semi-structured and unstructured databases to scientific databases such as genome databases. Hadjieleftheriou and Srivastava discuss the issues of indexing techniques and algorithms based on inverted indexes and a variety of weighted set-based string similarity functions.

With the recent trend in generalizing prefix-based auto-completion, Li and Li provide an overview of the information-access mechanism, where the system attempts to find answers to queries as users type in their keywords. They discuss various technical challenges in terms of interactive search speed, and open problems that remain.

The keywords entered by users may imply different information needs of different users, and this causes ambiguity during query processing. Liu and Chen discuss several post-processing methods for keyword-based retrieval on structured data with the objective of making the results more meaningful to users.

Keyword-based retrieval is well studied in the context of information retrieval. Webber examines the evolving practices and resources for effectiveness evaluation of keyword search over relational databases, compares them with longer-standing full-text evaluation methodologies in information retrieval, and offers some suggestions for future development.

Duplicates are common in databases due to abbreviation and typographical errors, and these create the nearly duplicate records problem. Yang et al. describes a system called RSEARCH for identifying nearly duplicate records so that meaningful results may be generated efficiently.

Yu, Qin and Chang present a survey on recent developments in keyword-based search over relational databases that focus on identifying primitive structures as answers and finding top-k answers efficiently. The focus is on proposals that support keyword search over RDBMS using SQL, and those viewing a relational database as a directed graph.

Beng Chin Ooi
National University of Singapore

Enhancing Search with Structure

Soumen Chakrabarti   Sunita Sarawagi   S. Sudarshan

Computer Science and Engineering Department
Indian Institute Of Technology, Bombay, India
{soumen,sunita,sudarsha}@cse.iitb.ac.in

Abstract

Keyword search has traditionally focussed on retrieving documents in ranked order, given simple keyword queries. Similarly, work on keyword queries on structured data has focussed on retrieving closely connected pieces of data that together contain given query keywords. In recent years, there has been a good deal of work that attempts to go beyond the above paradigms, to improve search experience on unstructured textual data as well as on structured or semi-structured data. In this paper, we survey recent work on adding structure to keyword search, which can be categorized on three axes: (a) adding structure to unstructured data, (b) adding structure to answers, and (c) adding structure to queries allowing more power than simple keyword queries, but while avoiding the complexity of elaborate query languages that demand extensive schema knowledge.

1 Introduction

Web search and information retrieval (IR) have traditionally focused on retrieving documents in ranked order, given simple keyword queries. Meanwhile, work on keyword queries on structured data has focused on retrieving closely connected pieces of data that together contain given query keywords. Recent years have witnessed many attempts to go beyond the above paradigms, to improve search experience on data sources (see Section 2) ranging from relational data with textual fields to semi-structured graphs with text associated with nodes and edges to completely unstructured text. In this paper, we survey recent efforts toward adding structure to keyword search.

In Section 3 we review a few data models somewhere in the gap between relational tables from the database community and the sequence-of-tokens or vector-space models from the IR community. We are specifically interested in lowest common denominator representations of ER graphs, text segments connected to type hierarchies with attached entity nodes, RDF, and tables represented using HTML markup without formal schema.

Then, in Section 4.1, we discuss various possible definitions of the *unit of answer*, which is a tuple in relational queries and a document in IR. In the ER graph model, an answer may be a single node, or a small tree or subgraph. The answer may be an entity node in a type hierarchy. The answer may be a table, even if the source is unstructured text. The (initial) answer may be a form that the user has to fill to issue a more precise second query. In Section 4.2, we move on to the design of structure-enabling features in queries, while avoiding the complexity of full query languages.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

In Section 5 we outline algorithmic and systems issues in efficiently evaluating some forms of structured queries, as well as queries that provide some forms of structured answers. We first summarize how to annotate unstructured text with links to entities in a graph-structured entity and type catalog. Then we discuss indexing and query processing for entity search in annotated text. Next we describe how to respond to queries with subgraphs, forms, and tables.

2 Structure in data sources

Databases and text search systems evolved very differently for several decades, but new applications are pushing them toward common ground [15]. In this section, we describe three types of data sources useful in new applications somewhere between strictly structured and completely unstructured data management systems.

We begin in Section 2.1 with the most explicit form of structure, where the job is to index and search entity-relation (ER) tables or graphs, except that there may be free-form text associated with table cells or nodes and edges in the graph. Standard relational or XML search systems do not provide sufficiently flexible support for combining text and structure search in such settings.

We continue in Section 2.2 with a less friendly situation where record-like regular structure has to be automatically extracted from HTML source. Here HTML markup tags and their regular patterns provide clues to extracting structure.

The most challenging scenario is where there is no explicit structure in the source text, but the structure is created by linking text tokens to entities in a structured database, which might be tabular or graphical. This is discussed in Section 2.3.

2.1 Structured data with textual content

Relational databases are widely used to store and generate textual content for e-commerce and social media sites. Familiar examples in academia include DBLP and CiteSeer. Many table columns in such applications are free-form or somewhat stylized text. The dominant query paradigm combines small forms for structured (e.g., numeric) fields and text boxes for free-format text input. Unlike with SQL queries where any row ordering must be deliberate and explicit, users expect tuples to be implicitly sorted by a relatively vague notion of relevance, as in IR. Finally, data normalization and the use of primary and foreign keys across multiple tables means that different words in a query may match cells in different tables, connected via joined attributes. We can think of every row in every table as a node in a graph, with edges representing primary and foreign key matches.

This "lowest common denominator" graph representation is quite generic, and can also be used to represent XML documents with text "blobs" associated with leaf nodes, and nodes being labeled with some element type. This kind of labeled entity-relation graph can also be used to represent RDF data. A specific and interesting example for us is DBPedia [1], which is an RDF repository built by extracting triples from Wikipedia.

2.2 Records and tables embedded in HTML

Web pages often contain embedded tables and lists that can be interpreted as sets of multi-attribute records. Such sources can serve as a compact and high quality source of structured information, that has been ignored until recently [9, 11, 21, 23]. For example, researchers at Google report the extraction of 154 million information bearing tables from their search engine crawl [11]. In addition to providing partially structured sets of records, such artifacts also provide useful meta-data in the form of headers and titles. With proper tools, it is possible to collect several useful kinds of information such as instances of type-entity pairs from a table's header and its entity columns, and multi-way relationships from rows of a table. Specifically under the assumption that the first

[1] `http://dbpedia.org/About`

non-numeric unique column is an entity, tables can be used to extract (entity, attribute name, value) triples [10]. Lists on the web can be converted to yield useful multi-way relationships as discussed in [23]. The headers of tables provide an implicit grouping of attribute names that can be exploited in useful ways. For example, by exploiting the co-occurrence pattern of a column header with headers of other tables, one can group attribute names that are synonyms [11].

2.3 Unstructured text mentioning entities from catalog

The most "organic" sources of structure are in unstructured text on Web pages, and these are the most difficult to exploit reliably in search. In general, an *annotation* associates a short token segment with either a *type* of entities, or a specific entity. Named entity taggers can associate a coarse-grained type like *person* with the token sequence *Albert Einstein* with high accuracy. More fine-grained annotations may assert that tokens *Albert Einstein*, in a given context, refers to a *physicist* rather than a more generic type *person*. The most unambiguous level of information is that these tokens refer to a known entity, possibly expressed by a standard URN in an entity *catalog* such as Wikipedia.

Unstructured text in Wikipedia articles already have references to standard entity URNs within Wikipedia, and there are also many links from other pages that refer to a specific entity, to the corresponding URN in Wikipedia. Such annotated text leads to a new document representation (Section 3.2) that can be used for entity search. Authoring entity-annotated text is labor-intensive. Much of the Web is not annotated thus. In Section 5.1.2 we will summarize recent algorithms to automatically annotate free-form text with likely mentions of entities from a given catalog. These automatic annotation algorithms are not perfect, but the hope is that they have sufficient accuracy to assist semi-structured search.

3 Data representation

In Section 2 we discussed various forms in which structure may be explicit in, or be mined out of a data source. In this section we describe the corresponding abstract representations used in building storage and query systems. These representations bring together tabular relational data, document representation in traditional IR, and graph-structured data.

3.1 Entity and type catalog

A catalog provides a directed acyclic graph where nodes are *types* and *entities*. A directed edge labeled "$\subseteq$" from type T_1 to T_2 means that T_2 is a subtype or subclass of T_1. E.g., *Physicist* is a subtype of *Scientist*. Entities e are also represented as nodes, connected to types T they belong to, using edges labeled "$\in$". The catalog may be provided by WordNet [34], Wikipedia, or YAGO [43].

Each entity (or type) in the catalog is known by a unique ID. It is also known by one or more *lemmas*: short token sequences describing the entity (or type). E.g., the city of New York is known by lemmas "New York City" and "Big Apple". Over and above lemmas, each entity also has an associated *description* that uniquely defines it. In case of WordNet, the description is a dictionary definition. In case of Wikipedia, the description is a whole article page. More generally, we may find hyperlinks anywhere on the Web (including Wikipedia itself) to an entity e, and consider the anchor and surrounding text as another form of "description" of e.

3.2 Text linked to catalog

The vector space model [44] has served its purpose well in traditional IR, but more elaborate models are needed if we are to augment the query system with more structure. One way to do so is to exploit connections between the text and a catalog of types and entities. An *annotation* is a record with fields: document ID, range of token offsets, entity (or type) ID from the catalog, and an optional score of confidence

We might represent the corpus in graph format as a set of chains, one for each document. Each node in a chain represents a token in the corresponding document. In case a token is part of a suspected *mention* of entity e, the node corresponding to the token is attached by an edge to the node corresponding to e. The weight of this edge may be designed to reflect the (un)certainty of the mention. In case of multi-token mentions, depending on the application, one may add edges to each token or to the first, center or last token, with weights adjusted suitably.

In some cases, as in explicit hyperlinks from Web or Wikipedia documents to e, there is no uncertainty. But a majority of potential mentions are uncertain. E.g., the token segment "New York" may refer to either the city or the state, and "Einstein" may refer to the famous physicist or the well-known bagel makers. Many pages in Wikipedia are *disambiguation* pages that list alternative entity references for a given lemma.

3.3 General graphs with textual properties

In the above model of text linked to a catalog, the graph is very specific (catalog graph connected to document token chains) and there are only a handful of relationships: entity is instance of type, type is subtype of type, token/s mention entity, and tokens are adjacent/nearby. In a more general entity-relation (ER) graph model, nodes are arbitrary entities of given types. E.g., in representing DBLP, a node may represent a paper or author. Entity nodes may be explicitly connected by edges to a type "meta data" node, or the type may be interpreted as a discrete attribute of the entity node. In addition, nodes may have other named attributes; these could be numeric (e.g. date of publication) or string-valued. The string attributes may be short (such as author address) or long (an abstract).

Edges represent arbitrary binary relations between the nodes they connect. Like nodes, edges have types. E.g., a node representing a person may be connected to a node representing a paper, using an "is author of" edge. Generally, most binary relations have transposed relations, and we can model this as a reversed parallel edge, say, "is authored by" in the above example.

We can tokenize (long) text fields and represent each token as a node in the same ER graph. Each token node would then be connected to the entity nodes where they appear in attribute fields. We might define a "token appears in node" edge type to represent these relations.

3.4 RDF

The Resource Description Framework (RDF) is a framework originally designed for representing information on the Web. It is based on a set of (subject, predicate, object) triples, where subject and object are nodes of a graph, and the predicate (or property) is the label of an edge from subject to object. Each node may represent an entity, in which case it has a unique identifier, or it may represent a literal such as a string or number. Attribute values relationships are represented using edges. For example, we may have a graph with entity nodes "Jim Gray" and "Transaction Processing Concepts", the book written by Jim Gray. The Jim Gray entity may have an edge HasName linking it to a node representing the string "Jim Gray". and similarly the book may have an edge HasTitle linking it to a node representing the string "Transaction Processing Concepts". Although other graph models have been proposed for semistructured data, such as Lore [**?**], RDF is today the most widely used semistructured data model.

An RDF graph is conventionally represented as a set of triples, where each triple asserts a fact; an RDF graph asserts the conjunction of all triples in the graph. Unlike full featured knowledge representation languages, RDF does not allow specification of disjunction, negation, or quantification. In this aspect RDF is like the relational model, but unlike the relational model RDF does not require a strict schema, and it is restricted to the triple representation. Although RDF was originally proposed as a way for Web sites to provide semantics for their content, it has not been widely used in that context, but is a convenient unified representation for knowledge extracted from a variety of sources. For example, the YAGO project [43] uses RDF to represent knowledge extracted from Wikipedia and WordNet, among other sources.

3.5 Tabular data

At the top-level, tabular data extracted from the Web represents sets of records where the record boundaries are indicated by rows for tables and items for lists. Records comprise of a set of attributes. For lists, the attribute structure is latent and expensive extraction methods are required to reveal it (Section 5.4). For tables, the columns often correspond to attributes. Further, these can be partitioned relatively easily into those that represent entities, and those that are properties or attributes of the entities. Tables can optionally contain one or more header rows, and titles/context tokens describing the content.

Normally, the cells and headers of a table are not tied to any standardized schema. Greater leverage is possible by inter-linking the information across the millions of tables on the Web. One way to acheive this is to infer join links among table pairs — for example, Cafarella et al [11] propose to add join links between a column C of table T to a column C' of another table T' when the headers of C and C' are synonyms and co-occur with similar columns. A second approach is to annotate table columns and entities to a standardized catalog. In Section 3.2 we discussed the annotated text model where text snippets are annotated with entity nodes of a type hierarchy. For tables, in addition to entity annotation on the cells, type annotations can be attached to columns and relationship annotations among groups of columns. For example, based on its contents, a table's columns can be assigned type annotation such as "Western Movies", "Currency", "American director" and associated with relationship tags such as "box office total" and "directed by".

4 Models for queries and answers

To take advantage of explicit structure in the data, or latent structure in a corpus, we need one or both of the following features in the system:

- The unit of an *answer* has to be defined carefully. Unlike in Web search or IR, the answer will usually not be a document or page. As we shall see in Section 4.1, an answer to a keyword query may be a node in a graph, or a small subgraph, a table with textual and other fields, or even a form to fill for a second round of more structured query.
- There must be constructs in the query language to express more structure than in traditional keyword search systems. However, in doing so, the query language should not demand complete schema knowledge from users and border on to SQL or XQuery. Striking a balance in this matter is an important design issue. We will visit some promising query dialects in Section 4.2.

4.1 Answer structure for keyword queries

The unit of an answer to a keyword query on structured data can be defined in several ways, including nodes, trees, and graphs. Alternatively a keyword query can retrieve not raw data, but forms, or form results. We outline these options in this section.

4.1.1 Single node in a graph as answer

A common search task in the ER graph model is to rank single nodes in response to keyword queries. In standard IR, the score of a document in response to a query would be a function of the overlap of query words with text in the document. In the graph ER model, however, we would be interested in taking advantage of the graph structure for better ranking of response nodes.

As a specific motivating example, consider the DBLP or CiteSeer graphs. The paper about the Object Exchange Model (OEM) [38] is widely cited by papers about XML search, but it was written before XML became a mainstream research area in databases. Given that the OEM paper is directly related to XML technology and highly cited, one may argue that it should be ranked highly in response to a query containing 'XML'. As a more generic motivation, an edge between nodes u, v is evidence of an association, which might lead us to postulate that the score of of node u in response to a query should not be too different from the score of node v.

Proximity search approaches, such as [22], the near query mechanism in BANKS [1], and OBJECTRANK [7], return an entity node as an answer to a keyword query. For example, in the BANKS system, a query "author(near recovery)" would return authors that are closely related to the keyword recovery. The key issue is how to rank nodes in the input graph, in response to the given keyword query. Goldman et al. [22] find shortest paths from each potential answer node to each node containing the given keywords, and aggregate the shortest path lengths to get a node score. In constrast, BANKS [1] is based on a spreading activation model.

4.1.2 Trees and graphs as answers

A majority of the work on keyword search on structured data is based on finding a subtree connecting the keywords specified in the query. Early examples of this approach include BANKS [8], DBXplorer [4] and Discover [24].

While some early work ranked trees based on the number of edges, BANKS [8] proposed a more sophisticated ranking model based on edge directionality and edge weights; this model was later refined based on traversal probabilities in [26]. The primary motivation for the more sophisticated ranking approach was that unrelated nodes often have very short parts connecting them through "hub" nodes; for example if a paper node links to the conference in which it appears, every two papers in that conference have a path of length 2 connecting them, even if they are otherwise unrelated. The directed edge weight model penalizes such shortcut paths that connect a large number of nodes.

When nodes can contain significant amounts of text, IR ranking techniques are important for computing answer tree scores. Fang et al. [32] describe how to apply IR ranking techniques such as TF-IDF, document size normalization and phrase-based ranking to rank answer trees.

Given a keyword query, Précis [28] first finds nodes containing the keywords; but instead of just returning these nodes, for each such answer node Précis returns a subgraph of related nodes. The subgraph of related nodes is generated based on foreign key joins with other closely related relations, which can be automatically chosen based on some heuristics.

The Précis approach starts with a node or a path/tree and grows a graph around it. An alternative approach is based on finding small subgraphs that contain the given query keywords. For example, the EASE system [31] finds r-radius Steiner subgraphs that contain the given query keywords, ranked based on their compactness and length-normalized TF-IDF scores.

4.1.3 Forms and Queries as Answers

Presenting nodes, trees or graphs as answers is often not satisfactory for end users who are not concerned with the schema. For example, users of enterprise information systems are used to getting richly structured answers in response to filling in forms. A different approach is required to generate such structured answers; in particular, answers formats need to be predecided in some way, and a keyword query retrieves one of the predecided answer formats.

One approach, proposed by [17], precomputes a set of forms, with associated SQL queries, from the given database schema. A keyword search then returns a set of forms that are relevant to the query, instead of returning actual data; a form is considered relevant to a query if the keywords occur in relations used in the form query. The user can then provide values for form parameters to get at required information.

A second approach, proposed by [36], is to specify units of information called qunits, each defined using a parametrized query, defined in a language such as SQL/XML. Executing each such qunit template with different instantiations of the parameters leads to a collection of instantiated qunits. Keyword queries are evaluated against the set of instantiated qunits, and relevant qunits are returned as (ranked) answers. A major focus of [36] is on how to automate the generation of qunits, and a user study shows that automated generation can perform reasonably close to human generated qunits.

A third approach is applicable to a situation where an enterprise information system has already defined a large number of forms; here, the goal of a keyword query is to find a form, along with an instantiation of form parameters, such that the result of the instantiated form contains the specified keywords. In the approach proposed by [20], forms results are precomputed for all possible parameter values, and indexed offline. In enterprise information systems, this approach is complicated by the fact that different users may see different results on submitting the same form. In Section 5.3 we outline how Duda et al. [20] handle this problem. The problem of deep Web search, e.g. [40], is closely related. Work in this area has generally focussed on generating values to be filled into Web forms to surface data hidden behind the forms; pages thus surfaced are then indexed just like regular Web pages.

4.1.4 Tables as Answers

For many user information needs, the answer is best represented as a single table rather than as a collection of documents. For example, in Google Squared [2] a search for entity types such as “fighter jets” outputs a table with rows representing instances of fighter jets and columns denoting attributes like dimensions and crew size. The column of the table are either specified by the user or inferred by referring to an offline extracted knowledge base of attributes and classes [39]. Typically, such tables are constructed from noisy consolidation of information from multiple sources. So, the cells of the table contain multiple ranked plausible answers.

4.2 Query structure

Several approaches have been proposed for adding some degree of structure to keyword queries. We outline a few of these in this section.

4.2.1 Type and proximity queries

A broad class of *entity search* queries ask about an unknown entity belonging to a known type that satisfy some properties. These properties are loosely expressed in terms of the entity being mentioned in close juxtaposition with certain keywords. E.g., using the category *Physicist* in Wikipedia, we might look for physicists who play(ed) the violin by searching for short token sequences that contain both a descendant of type *Physicist* and the literal word *violin*. In Section 5.1.3 we present various issues of answering such queries. Searching for entities is now a standard track in the TREC and INEX competitions.

4.2.2 EntityRank

The next step is to ask for not one entity of a given type but a few entities with given types, related to each other in specified ways [16]. (The relation is again favored in a soft manner through lexical proximity within a short token sequence.) E.g., if we had surface recognizers (Section 2.3) for emails and phone numbers, and we managed to reliably annotate entities that are instances of *database researcher*, we could ask for a table with three columns: the mention of a database researcher (“David Lomet”), an email (`lomet@xyz.com`) and a phone number (“800-555-1212”). Note that the use of schema in the query “language” thus far is not very

[2] `http://www.google.com/squared`

overbearing. The only new query construct is a *type name*. Of course, the user has to know the type name from a standard catalog, but autocompletion and query suggestions can help [42].

4.2.3 Adding variables, contexts, and joins

Pushing further on the paradigm of structured queries on unstructured data, the logical next stop involves *variables* and *joins*. Suppose we can name a variable `?m` as a placeholder for a *movie*, variable `?o` as a number, and variable `?b` as a money amount, expressed as these subqueries:

- `?m` $\in^+$ *Type:Movie*
- `?o` $\in$ *Quantity:Number*
- `?b` $\in^+$ *Quantity:MoneyAmount*

(Here "$\in^+$" denotes transitive membership via intermediate subtypes.) Then we can ask for token segments or *contexts* `?c1` and `?c2` such that

- $\textsc{InContext}(\texttt{?c1}; \texttt{?m}, \texttt{?o}; \textit{won}, \textit{Oscar})$
- $\textsc{InContext}(\texttt{?c2}; \texttt{?m}, \texttt{?b}; \textit{production}, \textit{cost}, \textit{budget})$

and finally aggregate over contexts `?c1` and `?c2` to get a table with three columns: a movie name mention, the number of Oscars it won, and its production budget. Although such a language may not be used by end-users, it can be used as a "decision support" tool.

4.2.4 Specifying graph structures

Keyword queries can specify partial structures on graphs. For example, XPath expressions, or approximate match XPath expressions can be used to restrict the scope of a keyword query on XML data; see for example, [6], which explores flexible path specifications in the context of XML data. The idea of flexible path specifications can be easily applied to graph data. In the context of RDF graphs, one could use SPARQL, a structured, SQL-like query language for querying RDF.

The NAGA search engine [27] proposes a query language that allows specification of some kinds of graph constraints such as path and edge constraints. Unlike SPARQL, the graph-based NAGA query language takes into account uncertainty of underlying facts, and aggregates support for a fact from multiple sources. A comparison of result quality using NAGA with results of Google search, and results of keyword queries using the BANKS scoring model is presented in [27].

4.2.5 Specifying table structure

In Section 4.1.4 we discussed the Google Squared method of returning table answers to keyword queries. These provide limited control to the user on how to describe the answer table. We propose two alternative modes of querying for tables. The first kind of queries are called column description queries where the user fills in multiple keyword boxes where each text box describes a column of the answer table. For example to retrieve a list of scientists and their inventions, he fills two text boxes: one containing words like "inventor, scientist" and the second containing words like "major invention". The second kind are content queries where a user submits a few examples of structured rows of the desired table and expects to get more instances of such rows. These examples can prove invaluable for teaching an extractor how to convert unstructured list sources to structured tables in the answer [23].

5 Algorithms and systems

In this section we review some algorithmic and system issues that come up when one implements the annotator, index, and search capabilities introduced before.

5.1 Entity search in annotated text

5.1.1 Broad type annotation

Machine learning techniques have come a long way toward annotating text with coarse-grained types. Let a text token sequence be called $x = (x_1, x_2, \ldots, x_T)$, and designate (unknown) labels $y = (y_1, y_2, \ldots, y_T)$ to these token positions. Each y_t can take on type values like *Person*, *Street name*, *Paper title*, *Phone number*, and so on. Usually there is also a "none of the above" label for tokens not recognized to belong to any of the specified groups. Then the job is to find the best labeling y. Note that, if each token position can have one of M labels, the number of possible labelings is astronomical, M^T. This explosion can be avoided using the Viterbi algorithm used in hidden Markov models. Recent techniques have improved upon HMMs by defining general families of *feature functions* $\phi(x, y) \in \mathbb{R}^d$ and then learning a *model* $w \in \mathbb{R}^d$ such that the best labeling is given by $\arg\max_y w^\top \phi(x, y)$ [41].

5.1.2 Entity disambiguation

Beyond broad types, we want to annotate tokens with disambiguated entities. Suppose s is a spot (token segment) in a document that is suspected to mention some entity from a set Γ_s. It is possible that s does not mention *any* entity in Γ_s, and the "null" label is denoted $N.A.$. Let S_0 be the set of all spots in a document. Then the job is to pick, for each s, an entity label $\gamma \in \Gamma_s \cup \{N.A.\}$. Recall that γ comes from an entity catalog. Specifically, if WordNet is the catalog, each concept has a dictionary-like definition with an example "gloss" showing typical usage. If the catalog is Wikipedia, there is a whole article describing the entity at length. On the other side, the potential mention s has a textual context. We would generally expect some affinity between the context of s and the definition of γ. This can be expressed in a manner neutral to the specific identity of s or γ, as a feature vector $f_s(\gamma)$ in some space. Again, we can learn a model w in the same space such that the predicted entity label is $\arg\max_{\gamma \in \Gamma_s \cup N.A.} w^\top f_s(\gamma)$ [29, 33].

However, one may do better by exploiting the fact that entities mentioned in a single discourse tend to be semantically related. E.g., Wikipedia lists several Michael Jordans of who only one is a Computer Science professor, and likewise with Stuart Russell. However, a single Web page listing both names almost certainly refer to the two computer scientists. A collective optimization over all spot labels [19,35] can improve disambiguation accuracy noticeably.

5.1.3 Indexing and query processing

To process INCONTEXT queries as presented in Section 4.2.3, we need two basic devices:

- Find connections between types in the query to entities mentioned in token segments.
- Score and rank these token segments using textual proximity as an important feature.

Reachability testing between the query type and a candidate entity answer can be done in at least two ways:

- At query time, expand the query type into each candidate in turn, and score the candidates. In other words, if the query is looking for a *scientist* who studied whales, effectively ask if e and 'whale' appear in a context, where $e \in^+$ *scientist*. This slows down queries unacceptably: Even WordNet knows of 650 scientists and 860 cities, not to mention Wikipedia or YAGO.

- At indexing time, suppose a token is a mention of entity e, and let $\mathcal{T}_e$ be all types to which e belongs directly or transitively. Then we insert pseudotokens encoding all types $T \in \mathcal{T}_e$ at the same token position and index these. This works well in applications with a small type system (5–10) [16]. In contrast, Wikipedia and YAGO have over 250,000 types and over 2.2 million entities, and the average size of $\mathcal{T}_e$ may be 10–30. This leads to unacceptable index size blowup.

One approach [13] is to compromise on both ends: index a carefully chosen subset of types, then do more work at query time. I.e., follow a "pre-generalize/post-filter" paradigm. Given a query type a, if a has not been indexed, generalize to some type $g \supseteq^+ a$, and launch an IR-style query. Afterward, check if the instance of g in each candidate context is an also instances of a; if not, discard. Experiments with TREC and WordNet suggest that index size can be kept comparable to a standard text index while slowing down queries by less than a factor of 2.

5.2 Ranking entity nodes in ER graphs

Proximity queries in ER graphs seek entity nodes that are "near" nodes representing query words, or entity nodes that match query words. This paradigm is easiest to express in terms of personalized PageRank [25, 37]. Each directed edge (u, v) of the data graph has a *conductance* $C(v, u) \in (0, 1)$, which is the conditional probability $\Pr(v|u)$ that a "random surfer", in PageRank parlance, will walk from node u in the previous step to node v in the current step, should he decide to walk, which happens with probability $\alpha \in (0, 1)$. By design, $\sum_v C(v, u) = 1$. With probability $1-\alpha$, he *teleports* to a node w in the graph with probability $r(w)$, with $\sum_u r(u) = 1$. It is well-known that the PageRank vector p_r corresponding to teleport r satisfies the recurrence $p_r = \alpha C p_r + (1 - \alpha)r$, which solves to $p_r = (1 - \alpha)(\mathbb{I} - \alpha C)^{-1} r$. Summarizing, query processing consists of

- Designing the data graph, C and α ahead of time
- Using the query to define r
- Computing PageRank vector p_r
- Reporting the top K nodes u with the largest $p_r(u)$ scores.

A variant of the above paradigm (that allowed taking much of the computation offline) was proposed as OBJECTRANK [7]. The matrix C can also be learned automatically from training data [2, 3].

Traditional offline computation of PageRank is done by power iteration: Initialize p_r to some random nonzero vector, then iterate $p_r \leftarrow \alpha C p_r + (1 - \alpha)r$. For large data graphs (beyond tens of millions of nodes), it is not practical to compute p_r at query time using power iterations. The special structure of personalized PageRank can be exploited [12] to achieve an almost *constant* query time independent of graph size, provided special indices are precomputed whose size scales as a modest fraction of graph size.

5.3 Executing tree and form queries

In this section, we address the issue of answering tree and form queries, focusing on preprocessing/indexing, and query execution.

Work in the area of efficient computation of connection trees or graphs as answers to keyword queries can be divided into two approaches:

1. approaches based on generating and executing SQL queries, and
2. approaches based on graph traversal.

In a companion article in this issue, Yu et al. [45] provide an extensive survey of techniques for executing keyword queries on relational and graph databases.

Work on indexing deep Web content, such as by Raghavan and Garcia-Molina [40], is primarily based on surfacing the content by filling in form values, so that the content can be indexed subsequently. This approach is implemented in popular search engines today.

As an alternative, if the domain of an input to a form is known, e.g. a set of flight numbers, the set of values associated with the domain can be added to the document representing the form; a query containing these keywords could then retrieve the form.

In contrast to the Web scenario, in an enterprise setting the underlying data is available, and can be used to retrieve forms without surfacing all results of the form; however access restrictions must be taken into account. For example, in the approach of Duda et al. [20], each form is specified using a representation language which allows the system to understand what results are generated for what form parameters, with the identity of the user also treated as a parameter. Then, keywords from parts of the form that are dependent on a parameter value (for example, the name of a person where employee identifier is a parameter) are stored in a keyword index with an associated predicate.

For example, consider a form that outputs an employee name, given the employee identifier. If employee ID 2345 had the name John, the keyword index entry for John would include the above form, with an associated predicate "empID=2345". A query "John" would retrieve an index entry with the above predicate, allowing the system to infer that the form, with parameter empId=2345 is an answer to the query. If the query contains multiple keywords, the inverted lists are merged, taking the predicates into account. Parts of the form which are common across all parameter values would not have any associated predicate, minimizing the storage and query cost.

5.4 Processing table queries

In this section we discuss how to process table search queries of the kind described in 4.2.5 by tapping tables and lists on the Web. We present a brief summary; more details can be found elsewhere [23][3].

5.4.1 Pre-processing and annotation

Recognizing useful tables and lists First, we need to identify parts of HTML pages that contain useful record sets. Depending only on HTML tags such as ⟨table⟩ and ⟨li⟩ is not sufficient as these are often used as layout tools or navigational aids for non-record data. For instance, as observed in [9] only 10% of HTML table tags are used to represent structured content. Similarly, only a small fraction of lists on HTML pages actually represent record sets.

Catalog annotations on web tables A catalog is used to annotate table cells with entity nodes, table columns with type nodes and column pairs with relationship identifiers. The lemma attached with these nodes are used to associate additional keywords with table cells and columns.

Indexing A text index is constructed on the extracted record sets where each set is associated with three kinds of fields: a bag of word representation of all the contents in the table, the catalog annotations, and the union of the title and descriptive context of the table.

5.4.2 Processing at query time

During query processing the index is first probed using the query words to find a list of candidate tables and lists. These are then subjected to the following steps:

[3] `http://www.cse.iitb.ac.in/~sunita/wwt`

Table extraction from HTML record sets There is a long history of extracting structure from record data on the Web [5, 14, 18, 21, 30, 46]. However, most of this work has been restricted to specific verticals like shopping, advertisements, and publications. Recent work [23] seeks robust extractors that do not depend on prior schema knowledge. At query time they have only (1) the few example rows provided by the user, and (2) other candidate record sets with overlapping content. The trick is to first use the query rows to create a partially labeled dataset, and then fit a model that simultaneously maximizes the probability of the labeled data and the probability that overlapping content across different lists get the same label.

Consolidation and ranking The extractor outputs a set of tables along with row and cell confidence scores indicating the probability of correctness of the extraction. The consolidator merges these tables into a single result table by resolving the set of rows that are duplicates. A cell in the consolidated table contains a set of cell values that are aliases of each other. Intuitively, a high scoring consolidated row is one which repeats in sources which overlap greatly with query words, has a high confidence score of extraction, and contains most of the useful columns.

6 Outlook

A great deal of recent work aims to improve the quality of results of keyword queries by adding structure to data, answers, and to queries themselves. We believe that the results of this research will have a significant impact on bridging the gap between structured and unstructured search. This article gives the reader a feel for the fast-growing area, but we cannot claim to have covered it exhaustively.

Acknowledgements: This work was supported in part by research grants from Microsoft and Yahoo.

References

[1] B. Aditya, S. Chakrabarti, R. Desai, A. Hulgeri, H. Karambelkar, R. Nasre, Parag, and S. Sudarshan. User interaction in the BANKS system (demo paper). In *ICDE*, pages 786–788, 2003.

[2] A. Agarwal and S. Chakrabarti. Learning random walks to rank nodes in graphs. In *ICML*, 2007.

[3] A. Agarwal, S. Chakrabarti, and S. Aggarwal. Learning to rank networked entities. In *SIGKDD Conference*, pages 14–23, 2006.

[4] S. Agrawal, S. Chaudhari, and G. Das. DBXplorer: A system for keyword-based search search over relational databases. *ICDE*, 2002.

[5] M. Álvarez, A. Pan, J. Raposo, F. Bellas, and F. Cacheda. Extracting lists of data records from semi-structured web pages. *Data Knowl. Eng.*, 64(2):491–509, 2008.

[6] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. Flexpath: Flexible structure and full-text querying for xml. In *SIGMOD*, pages 83–94, 2004.

[7] A. Balmin, V. Hristidis, and Y. Papakonstantinou. ObjectRank: authority-based keyword search in databases. In *VLDB Conference*, 2004.

[8] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.

[9] M. Cafarella, N. Khoussainova, D. Wang, E. Wu, Y. Zhang, and A. Halevy. Uncovering the relational web. In *WebDB*, 2008.

[10] M. J. Cafarella. Extracting and querying a comprehensive web database. In *CIDR*, 2009.

[11] M. J. Cafarella, A. Halevy, Y. Zhang, D. Z. Wang, and E. Wu. Webtables: Exploring the power of tables on the web. In *VLDB*, 2008.

[12] S. Chakrabarti. Dynamic personalized PageRank in entity-relation graphs. In *WWW Conference*, Banff, May 2007.

[13] S. Chakrabarti, K. Puniyani, and S. Das. Optimizing scoring functions and indexes for proximity search in type-annotated corpora. In *WWW Conference*, Edinburgh, May 2006.

[14] C. Chang. and S. Lui. Iepad: Information extraction based on pattern discovery. In *WWW*, pages 681–688, 2001.

[15] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR technologies: What is the sound of one hand clapping? In *CIDR*, 2005.
[16] T. Cheng, X. Yan, and K. C. Chang. EntityRank: Searching entities directly and holistically. In *VLDB Conference*, pages 387–398, Sept. 2007.
[17] E. Chu, A. Baid, X. Chai, A. Doan, and J. Naughton. Combining keyword search and forms for ad hoc querying of databases. In *SIGMOD*, 2009.
[18] W. W. Cohen, M. Hurst, and L. S. Jensen. A flexible learning system for wrapping tables and lists in html documents. In *WWW*, 2002.
[19] S. Cucerzan. Large-scale named entity disambiguation based on Wikipedia data. In *EMNLP Conference*, pages 708–716, 2007.
[20] C. Duda, D. A. Graf, and D. Kossmann. Predicate-based indexing of enterprise web applications. In *CIDR*, 2007.
[21] H. Elmeleegy, J. Madhavan, and A. Halevy. Harvesting relational tables from lists on the web. In *VLDB*, 2009.
[22] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *VLDB*, pages 26–37, 1998.
[23] R. Gupta and S. Sarawagi. Answering table augmentation queries from unstructured lists on the web. In *Proc. of the 35th Int'l Conference on Very Large Databases (VLDB)*, 2009.
[24] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-Style keyword search in relational databases. *VLDB*, 2002.
[25] G. Jeh and J. Widom. Scaling personalized web search. In *WWW Conference*, pages 271–279, 2003.
[26] 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.
[27] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. NAGA: Searching and ranking knowledge. In *ICDE*, pages 953–962, 2008.
[28] G. Koutrika, A. Simitsis, and Y. E. Ioannidis. Précis: The essence of a query answer. In *ICDE*, pages 69–78, 2006.
[29] S. Kulkarni, A. Singh, G. Ramakrishnan, and S. Chakrabarti. Collective annotation of Wikipedia entities in Web text. In *SIGKDD Conference*, 2009.
[30] K. Lerman, C. Knoblock, and S. Minton. Automatic data extraction from lists and tables in web sources. In *Workshop on Advances in Text Extraction and Mining (ATEM)*, 2001.
[31] 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*, 2008.
[32] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD*, pages 563–574, 2006.
[33] R. Mihalcea and A. Csomai. Wikify!: linking documents to encyclopedic knowledge. In *CIKM*, pages 233–242, 2007.
[34] G. Miller, R. Beckwith, C. FellBaum, D. Gross, K. Miller, and R. Tengi. Five papers on WordNet. Princeton University, Aug. 1993.
[35] D. Milne and I. H. Witten. Learning to link with Wikipedia. In *CIKM*, 2008.
[36] A. Nandi and H. V. Jagadish. Qunits: queried units in database search. In *CIDR*, 2009.
[37] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the Web. Manuscript, Stanford University, 1998.
[38] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. of ICDE*, 1995.
[39] M. Pasca, B. V. Durme, and N. Garera. The role of documents vs. queries in extracting class attributes from text. In *CIKM*, pages 485–494, 2007.
[40] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *VLDB*, pages 129–138, 2001.
[41] S. Sarawagi. Information extraction. *FnT Databases*, 1(3), 2008.
[42] A. Singh, S. Kulkarni, S. Banerjee, G. Ramakrishnan, and S. Chakrabarti. Curating and searching the annotated web. In *SIGKDD Conference*, 2009. System demonstration.
[43] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A core of semantic knowledge unifying WordNet and Wikipedia. In *WWW Conference*. ACM Press, 2007.
[44] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan-Kaufmann, May 1999.
[45] J. X. Yu, L. Qin, and L. Chang. Keyword search in relational databases: A survey. *IEEE Data Eng. Bull.*, 33(1), Mar. 2010.
[46] Y. Zhai and B. Liu. Web data extraction based on partial tree alignment. In *WWW*, pages 76–85, 2005.

Searching RDF Graphs with SPARQL and Keywords

Shady Elbassuoni, Maya Ramanath, Ralf Schenkel, Gerhard Weikum

Max-Planck Institute for Informatics
Saarbruecken, Germany
E-mail: {elbass,ramanath,schenkel,weikum}@mpi-inf.mpg.de

Abstract

The proliferation of knowledge-sharing communities like Wikipedia and the advances in automated information extraction from Web pages enable the construction of large knowledge bases with facts about entities and their relationships. The facts can be represented in the RDF data model, as so-called subject-property-object triples, and can thus be queried by structured query languages like SPARQL. In principle, this allows precise querying in the database spirit. However, RDF data may be highly diverse and queries may return way too many results, so that ranking by informativeness measures is crucial to avoid overwhelming users. Moreover, as facts are extracted from textual contexts or have community-provided annotations, it can be beneficial to consider also keywords for formulating search requests. This paper gives an overview of recent and ongoing work on ranked retrieval of RDF data with keyword-augmented structured queries. The ranking method is based on statistical language models, the state-of-the-art paradigm in information retrieval. The paper develops a novel form of language models for the structured, but schema-less setting of RDF triples and extended SPARQL queries.

1 Motivation and Background

Entity-Relationship graphs are receiving great attention for information management outside of mainstream database engines. In particular, the Semantic-Web data model RDF (Resource Description Format) is gaining popularity for applications on scientific data such as biological networks [14], social Web2.0 applications [4], large-scale knowledge bases such as DBpedia [2] or YAGO [13], and more generally, as a light-weight representation for the "Web of data" [5].

An RDF data collection consists of a set of subject-property-object triples, SPO triples for short. In ER terminology, an SPO triple corresponds to a pair of entities connected by a named relationship or to an entity connected to the value of a named attribute. As the object of a triple can in turn be the subject of other triples, we can also view the RDF data as a graph of typed nodes and typed edges where nodes correspond to entities and edges to relationships (viewing attributes as relations as well). Some of the existing RDF collections contain more than a billion triples.

As a simple example that we will use throughout the paper, consider a Web portal on movies. Table 1 shows a few sample triples. The example illustrates a number of specific requirements that RDF data poses for querying:

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Subject (S)	Property (P)	Object (O)
Ennio_Morricone	hasType	Composer
Ennio_Morricone	hasWonPrize	Academy_Award
Ennio_Morricone	composedSoundtrack	C'era_una_volta_il__West
Henry_Fonda	actedIn	C'era_una_volta_il__West
Henry_Fonda	hasWonPrize	Academy_Award
Claudia_Cardinale	actedIn	C'era_una_volta_il__West
Claudia_Cardinale	hasWonPrize	Golden_Berlin_Bear
El_Laberinto_del_Fauno	hasType	Movie
El_Laberinto_del_Fauno	hasTag	Fantasy
El_Laberinto_del_Fauno	hasTag	Franco_Regime_in_Spain
Guillermo_del_Toro	directed	El_Laberinto_del_Fauno
Guillermo_del_Toro	hasWonPrize	Academy_Award
Javier_Navarette	createdFilmMusic	El_Laberinto_del_Fauno
Javier_Navarette	nominatedFor	Grammy_Award

Table 1: SPO triples in the RDF data model

1. Despite the repetitive structure in some parts of the data, there is often a high diversity of property names across the entire dataset. Thus, we need a flexible query language to search and explore *schema-less* data. The W3C-endorsed SPARQL language offers the equivalent of select-project-join queries, but in contrast to SQL, allows wildcards for property names.

2. While this is a powerful feature for schema-less data, it would still be a Boolean-match evaluation. Therefore, SPARQL queries may often produce empty results when queries are too specific, or overwhelm the user or application program with huge result sets. This calls for meaningful *ranking* of search results, based on statistics about the data and user queries.

3. While ranking is desired for structured query conditions, we should also consider that RDF triples often come with additional textual components. For example, movie portals also contain user comments, knowledge bases that are automatically built by information extraction from Web sources should also include the source texts, and biological data collections are often annotated by expert users or may reference PubMed articles. To tap on these potentially valuable texts, an RDF query language should also support text search with *keywords* and *phrases*.

This paper gives an overview on how these three requirements can be addressed. It is based on recent and ongoing research on IR-style ranking for RDF data [8, 9]. In Section 2, we discuss how to extend the SPARQL language with keyword search. In Section 3, we show how to compute principled rankings for search results in a schema-less but structured data setting. In Section 4, we sketch experimental studies, as evidence for the viability of the proposed model. We conclude with an outlook on open issues.

2 Extending the SPARQL Query Language

The SPARQL language is the W3C standard for querying RDF data. A query consists of conjunctions of elementary SPO search conditions, so-called *triple patterns*. For example, a question about prize-winning composers who have composed music for movies that feature prize-winning actors, with results consisting of composer-movie pairs, can be expressed in SPARQL as follows:

Select ?x, ?m *Where* $\{$?x hasType Composer . ?x hasWonPrize ?p .
?x composedSoundtrack ?m . ?m hasType Movie .
?a actedIn ?m . ?a hasWonPrize ?q $\}$

Each triple pattern has one or two of the SPO components replaced by variables such as ?x , the dots between the triple patterns denote logical conjunctions, and using the same variable in different triple patterns denotes join conditions. For the example data of Table 1, this query returns Ennio Morricone and C'era Una Volta il West (which stars Henry Fonda). However, the query would miss results about movies whose directors won an award or with actors or directors nominated for an award (which often is a major honor already). Moreover, as there is no prescriptive schema for property names, there are a variety of ways for expressing the appearance of musical compositions in movies; referring only to the composedSoundtrack property is unduly restrictive. Fortunately, SPARQL allows relaxing the query by having variables for property names. This way, the query could be re-phrased into the more liberal form:

Select ?x, ?m *Where* $\{$?x hasType Composer . ?x hasWonPrize ?p .
?x ?p ?m . ?m hasType Movie . ?a ?r ?m . ?a hasWonPrize ?q $\}$

This is still a highly structured query, with typed entity variables, join conditions, etc. But it is now so relaxed that we would obtain a large number of results on a realistically sized movie collection. This too-many-results situation thus calls for a ranked result list, in the IR spirit, rather than a result set merely based on Boolean matching. We will discuss our approach to ranking in the next section.

As mentioned before, RDF triples may have associated text passages, e.g., the text from which a fact was automatically extracted or user-provided comments from a Web2.0 community. In this case, that text provides extra information that we can consider in formulating queries. For example, suppose we search for film-music composers who have also written classical music and whose compositions appear in dramatic movie scenes such as gunfights in westerns or battles in easterns. This is difficult if not impossible to express by structured query conditions alone. To overcome this problem, we would prefer to simply specify keywords – but keywords within the context of a more precise triple pattern:

Select ?x, ?m *Where* $\{$?x hasType Musician $\{$*classical, composer*$\}$.
?x composedSoundtrack ?m . ?m hasType Movie $\{$*gunfight, battle*$\}$ $\}$

Here, two of the triple patterns are augmented by keyword conditions. The query should return results such as Tan_Dun composedSoundtrack Hero. Similarly, we could ask for rock musicians whose music is used in love scenes in movies (as mentioned in descriptions of the movie plot or discussed in fan communities), or we could refine our earlier condition about awards by specifying that the award citation should include certain keywords, and so on.

It is important to note that the query still has a rich structure with several triple patterns and that the association of keywords with individual triple patterns is crucial. The keyword parts have to be matched by facts whose associated text contains the specified keywords. This imposes a *semantic grouping* on the total set of query keywords. Without this structure, we would get different, possibly inappropriate, matches: for example, movies about classical composers such as Mozart with music composed by rock stars that were involved in gunfights and their personal battle with drugs (to give an outrageous example).

The outlined approach is close in spirit to languages like XQuery Full-Text [1] - except, and this is crucial, that we are dealing with RDF-style graph data and not with XML trees. The semantic grouping of keywords is impossible to express with today's Internet search engines. Separate phrase conditions such as "classical composer" and "gunfight battle" return very different results based on exact matching of consecutive words, and would miss good results such as "composer of the opera . . . a masterpiece for classical orchestra".

3 Ranking with Statistical Language Models

Whenever queries return many results, we need ranking based on the *informativeness* of the answers. Users prefer prominent entities and salient facts as answers. For example, the query about award-winning composers for movies with award-winning actors or directors should return prominent results such as "El Laberinto del Fauno" or "Hero" rather than matches such as "The Muse" (which features music by Grammy winner Elton John but is not a well-known movie).

State-of-the-art ranking models in IR are based on *statistical language models*, *LMs* for short. They have been successfully applied to passage retrieval for question answering, cross-lingual search, ranking elements in semistructured XML data, and other advanced IR tasks [7]. An LM is a generative model, where a document d is viewed as a probability distribution over a set of words $\{t_1, ..., t_n\}$ (e.g., a multinomial distribution), and a query $q = \{q_1, ..., q_m\}$ with several keywords q_i is seen as a sample from this distribution. The parameters of the distribution d are determined by maximum-likelihood estimators in combination with advanced smoothing (e.g., Dirichlet smoothing). This is not unlike $tf * idf$-style frequency-based models, but LMs are more principled and the smoothing method is often decisive for good results. Now one can estimate the likelihood of query q for different candidate documents, and the one document that maximizes this likelihood should be the highest-ranked result. Alternatively, we can also associate queries with LMs: a query is a probability distribution over keywords, derived from the query itself and, for example, a general user-interest profile or other sources for smoothing. Then, the query-likelihood model is equivalent to ranking based on the Kullback-Leibler divergence (KL-divergence for short) of the query LM with regard to the candidate document LMs [16]. KL-divergence is an information-theoretic measure (aka. relative entropy) for comparing two probability distributions. It is always non-negative and zero only for identical distributions.

3.1 Entity-Relationship Ranking

Recently, extended LMs have been developed for entity ranking in the context of expert finding in enterprises and Wikipedia-based retrieval and recommendation tasks [10–12, 15]. These models are limited to entities – the nodes in an entity-relationship graph. In contrast, general knowledge search needs to also consider the role of relations – the edges in the graph – for answering more expressive classes of queries. Moreover, the discussed work on entity IR is still based on keyword search and does not consider structured query languages like SPARQL. Ranking for structured queries has been intensively investigated for XML [1], and, to a small extent, for restricted forms of SQL queries [6]. Ranking has also been studied in the context of keyword search on relational graphs (e.g., [3]). However, these approaches do not carry over to the graph-structured, largely schema-less RDF data collections and the expressive queries discussed before.

What we need for RDF knowledge ranking is a generalization of entity LMs that considers relationships (RDF properties) as first-class citizens. Recent work on the NAGA search engine [8, 9] has addressed these issues and developed full-fledged LM-based methods for ranking the results of extended SPARQL queries.

In IR, typical LM-based ranking for documents, passages, or entities are associated with informative distributions over words (or phrases or N-grams) and queries have straightforward LMs (essentially just the words in the query). In contrast, our LMs for structured RDF search are probability distributions over facts (RDF triples). Our approach is to construct a query LM for the query and result LMs for each potential result. The KL-divergence between the query LM and a result LM gives a measure of relevance with the results being presented to the user in increasing order of KL-divergence.

3.2 Estimating the Query LM

The LM for a triple pattern, the basic unit of a structured query, considers all possible substitutions of its variables by matching triples from the underlying RDF data collection. Consider the pattern ?x directed ?y. This is satisfied by all director-movie pairs from directed triples in the data. The LM for this triple pattern is a probability distribution over these triples (with smoothing by giving a small amount of probability mass to all other triples).

The probability of a given triple in the LM of a triple pattern can be viewed as the probability that the user is interested in this particular triple as an answer to her question. As discussed above, some triples are more informative than others, because they refer to important directors, important movies, etc. We can incorporate this aspect into the LM by using statistical weights for different triples in order to construct a non-uniform distribution. To this end, we consider the *witnesses* of a given triple: how often, on the Web or in the news, do we see this triple. Alternatively, if we construct the RDF data collection by automatic information extraction from Web sources, the witnesses of a triple are the distinct sources from which we have extracted the triple. In our implementation, we issued keywords queries for each triple against a major search engine and used the reported result sizes as estimates for witness counts. These counts are pre-computed and stored in indexes.

#	**Triple** (t_i)	$c(t_i)$	$P_Q(t_i)$
t_1	Ivana_Baquero actedIn El_laberinto_del_fauno	200	$200/1000 = 0.2$
t_2	Henry_Fonda actedIn C'era_una_volta_il_West	250	$250/1000 = 0.25$
t_3	Holly_Hunter actedIn The_Piano	200	$200/1000 = 0.2$
t_4	Robert_Duvall actedIn Apocalypse_Now	350	$350/1000 = 0.35$

Table 2: Triples matching ?a actedIn ?m with witness counts

Let $\hat{q}_i$ be the set of triples which match the triple pattern q_i and $c(t_i)$ be the number of witnesses of triple t_i. The probability $P_Q(t_i)$ is then estimated as follows:

$$P_Q(t_i) = \frac{c(t_i)}{\sum_{t \in \hat{q}_i} c(t)}$$

For the triple pattern ?a actedIn ?m, Table 2 shows a very simple LM over four actedIn triples (smoothing over all triples is ignored here for simplicity).

Now consider a query with two or more patterns, for example, [?x directed ?y . ?x hasWonPrize Academy_Award] asking for movies whose directors have won the Academy Award. For queries with N triple patterns, we now define the query LM to be a probability distribution over N-tuples of triples. For tractability of both parameter estimation and query-time computation, we assume probabilistic independence among pairs of triple patterns and compute the entire query LM as:

$$P_Q(T) = \prod_i P_Q(t_i)$$

where $P_Q(T)$ is the probability of the N-tuple $(t_1, \ldots, t_N)$ in the query LM and $P_Q(t_i)$ is the probability of triple t_i. $P_Q(t_i)$ is computed as previously described.

The Query LM for Keyword-Augmented Structured Queries. For estimating the query LM of keyword-augmented queries, each triple in the RDF collection is conceptually extended by an associated keyword taken from textual annotations or witness documents. The same triple is repeated with different words if it has multiple words associated with it. $c(t_i; w_k)$ is the number of witnesses for triple t_i occurring with word w_k.

We now need to compute the probability of keyword-augmented triple patterns of the form $q_i = q\{w_1, ..., w_m\}$ where q is a simple triple pattern and w_k is an associated keyword. As before, we need to estimate $P_Q(t_i)$, the probability of a triple t_i in the query LM. However, since now we have a context, in the form of words $w_1, ..., w_m$, we actually need to estimate the probability $P_Q(t_i|w_1, ..., w_m)$. That is, the probability of the triple t_i, given the context $w_1, ..., w_m$. Assuming independence between keywords, we calculate the triple probability as

$$P_Q(t_i|w_1,...,w_m) = \prod_{k=1}^{m}[\alpha P_Q(t_i|w_k) + (1-\alpha)P(t_i)] \quad (1)$$

where $P_Q(t_i|w_k)$ is the probability of t_i given the single-keyword context w_k, $P(t_i)$ is the smoothing component, and the parameter α controls the influence of smoothing. Note that it is crucial to smooth the probabilities, since otherwise $P_Q(t_i|w_1,...,w_m) = 0$ if t_i is not associated with at least one word w_k in the context. We use uniform smoothing (i.e, a uniform probability distribution for $P(t_i)$). Now, let $\hat{q}_i$ be the set of triples which match the triple pattern q. Then,

$$P_Q(t_i|w_k) = \begin{cases} \frac{c(t_i;w_k)}{\sum_{t \in \hat{q}_i} c(t;w_k)} & \text{if } t_i \in \hat{q}_i \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

The necessary count values $c(t_i;w_k)$ are precomputed and stored in indexes (indexed on triple identifier and keyword identifier).

3.3 Result LM and Ranking

For query Q with N triple patterns, let G be a result. Now, the LM of G, denoted P_G, is estimated over all N-tuples as: $P_G(T) = \beta P(T|G) + (1-\beta)P(T|C)$, where β is another smoothing parameter. If G is the N-tuple T, then $P(T|G) = 1$, otherwise, $P(T|G) = 0$. For the smoothing component, probabilistic independence is assumed: $P(T|C) = \prod_{i=1}^{N} P(t_i|C)$ where $P(t_i|C)$ is estimated given the entire corpus or data collection: $P(t_i|C) = \frac{c(t_i)}{\sum_{t \in KB} c(t)}$

Given the query and candidate-result LMs, the KL-divergence between the query LM P_Q of query Q and a result LM P_G of result G is computed as follows:

$$KL(Q||G) = \sum_i P_Q(T_i) log \frac{P_Q(T_i)}{P_G(T_i)}$$

The results are returned to the user in ascending order of KL-divergence.

4 Experimental Studies

We conducted a user study with relevance assessments collected on the Amazon Mechanical Turk platform. Our dataset was derived from a subset of the Internet Movie Database (IMDB) and consisted of around 600,000 RDF triples. In addition, each triple was augmented with keywords derived from the data source where it was extracted from. In particular, all the terms in the plots, tag-lines and keywords fields were extracted and stored with each triple.

We constructed 10 structured queries and 10 keyword-augmented queries. We presented the top-10 results for each query to 7 different evaluators who judged the results based on a 6-point scale ranging from "highly relevant" (5) to "irrelevant" (0). To measure the ranking quality, we used the Discounted Cumulative Gain (DCG) [7], which is a measure that takes into consideration the rank of relevant documents and allows the incorporation of different relevance levels. Dividing the obtained DCG by the DCG of the ideal ranking we obtained a *Normalized DCG (NDCG)* which accounts for the variance in performance among queries.

Table 3 shows the 10 evaluation queries and their NDCG values for the case of purely structured SPARQL queries. On average, we achieved a high NDCG value of 0.88. Similarly, Table 4 shows the 10 keyword-augmented SPARQL queries we used to evaluate the quality of the result ranking in the case where the queries contained keyword conditions. The third column gives the corresponding NDCG value for each query. On average, we obtained an NDCG value of 0.866. For both types of queries, the NDCG results are excellent (1.0

would be perfect). Note that the keyword-augmented queries cannot be expressed at all in purely structured form. If we dropped the keyword conditions and merely used the structural conditions, we would obtain much weaker if not useless results.

ID	SPARQL Query	NDCG
1	?a actedIn ?m . ?a hasWonPrize ?p . ?m hasWonPrize Academy_Award	0.834
2	?a hasWonPrize ?p . ?a actedIn ?m1 . ?a actedIn ?m2	0.891
3	?d directed ?m . ?d actedIn ?m	0.901
4	?a1 isMarriedTo ?a2 . ?a1 actedIn ?m . ?a2 actedIn ?m	0.905
5	?d hasWonPrize Academy_Award_for_Best_Director . ?d directed ?m . ?a actedIn ?m . ?a hasWonPrize Academy_Award_for_Best_Actor	0.869
6	?m hasGenre Comedy . ?a actedIn ?m . ?a directed ?m	0.956
7	?m hasGenre Comedy . ?a1 actedIn ?m . ?a2 actedIn ?m	0.894
8	?a hasWonPrize Academy_Award_for_Best_Actress. ?a actedIn ?m . ?a diedOnDate ?t	0.912
9	?d directed ?m . ?d hasWonPrize ?p1 . ?m hasWonPrize ?p2	0.818
10	?m1 hasGenre Family . ?m1 hasProductionYear 1995 . ?a actedIn ?m1 . ?m2 hasGenre Comedy . ?a actedIn ?m2	0.821

Table 3: NDCG for Purely Structured Queries

ID	SPARQL Query	NDCG
1	?a actedIn ?m {spielberg} . ?a hasWonPrize ?p	0.745
2	?m hasGenre Comedy {christmas} . ?a1 actedIn ?m . ?a2 actedIn ?m	0.846
3	?a1 hasWonPrize Academy_Award_for_Best_Actor . ?a1 actedIn ?m {relationship} . ?a2 hasWonPrize Academy_Award_for_Best_Actress . ?a2 actedIn ?m {love}	0.873
4	?d hasWonPrize Academy_Award_for_Best_Director . ?d directed ?m {new, york} . ?a actedIn ?m . ?a hasWonPrize Academy_Award_for_Best_Actor	0.872
5	?m hasGenre Comedy {school, friends}	0.939
6	?m hasGenre Comedy {police} . ?a actedIn ?m . ?a directed ?m	0.933
7	?d directed ?y {true, story} . ?d hasWonPrize ?p	0.921
8	?a hasWonPrize Academy_Award_for_Best_Actress . ?a actedIn ?m {paris} . ?a diedOnDate ?t	0.920
9	?d directed ?m {love} . ?d hasWonPrize ?p1 . ?m hasWonPrize ?p2	0.849
10	?m1 hasGenre Family {wedding} . ?m1 hasProductionYear 1995 . ?a actedIn ?m1 . ?m2 hasGenre Comedy {serial, killer} . ?a actedIn ?m2	0.762

Table 4: NDCG for Keyword-augmented Queries

In Table 5 we show some example purely-structured queries. For each query, the top-5 results are given. Due to space limitations, we just show the bound entities (i.e., matches to the variables in the queries). Next to each result, we also show the average relevance value given by the evaluators (column *Rel.*). Recall that each result was given a relevance value between 0 and 5, with 5 corresponding to highly relevant results.

For Query 3 in Table 3, asking for a movie directed and acted in by the same person, the top-5 results are all well-known movies (see Table 5). Similarly, for Query 5 in Table 3, asking for a director who won an Academy award and directed a movie that an Academy-awarded actor acted in, we also obtain very prominent movies, directors, and actors.

For the case of keyword-augmented SPARQL queries also, our result ranking is superior. For instance, consider Query 2 in Table 4, asking for comedy movies about "Christmas" and their directors. The top-5 movies

Q. ID	Rank	Result	Rel.
3	1	Sylvester_Stallone, Rambo	3.29
	2	Mel_Gibson, Braveheart	3.71
	3	Clint_Eastwood, Million_Dollar_Baby	3.71
	4	Woody_Allen, Annie_Hall	3.71
	5	Ben_Stiller, Zoolander	4.14
5	1	Martin_Scorsese, Robert_De_Niro, Casino	3.43
	2	Milo_Forman, Jack_Nicholson, One_Flew_Over_the_Cuckoo's_Nest	3.29
	3	William_Wyler, Gregory_Peck, Roman_Holiday	3.29
	4	Robert_Zemeckis, Tom_Hanks, Forrest_Gump	3.00
	5	Sydney_Pollack, Dustin_Hoffman, Tootsie	2.86

Table 5: Top-ranked results for some example purely-structured queries

Q. ID	Rank	Result	Rel.
2	1	Miracle_on_34th_Street, Maureen_O'Hara, Edmund_Gwenn	2.86
	2	Love_Actually, Liam_Neeson, Alan_Rickman	3.14
	3	Bad_Santa, John_Ritter, Billy_Bob_Thornton	2.86
	4	Jingle_All_the_Way, James_Belushi, Arnold_Schwarzenegger	3.29
	5	Christmas_in_Connecticut, Sydney_Greenstreet, Barbara_Stanwyck	3.14
4	1	Martin_Scorsese, Robert_De_Niro, Mean_Streets	3.29
	2	Elia_Kazan, Marlon_Brando, On_the_Waterfront	3.29
	3	James_L._Brooks, Jack_Nicholson, As_Good_as_It_Gets	3.14
	4	Spike_Jonze, Nicolas_Cage, Adaptation.	3.29
	5	John_Schlesinger, Dustin_Hoffman, Midnight_Cowboy	3.57

Table 6: Top-ranked results for some example keyword-augmented queries

(see Table 6) are all well-known comedies that have christmas as a theme. Query 4 in Table 4 asks for a director that won an Academy award, and directed a movie that an Academy-awarded actor acted in, and in addition the movie should be related to "New York". The top answers, shown in Table 6, are all salient movies featuring New York.

5 Future Directions

In our current work, the text-search conditions are limited to keywords. However, IR systems usually offer a richer repertoire of predicates: phrase matching (i.e., several contiguous keywords), proximity search (i.e., several non-contiguous keywords within short distance), negated conditions (i.e., taboo words that must not appear in a result), query expansion (i.e., adding related words that are not explicitly given in the query), and more. Our LM-based framework is geared for this, and can compute meaningful rankings even for such richer queries with a structured "backbone". However, efficient indexing and query processing pose major challenges.

Personalizing and diversifying search results are also open issues in the extended-SPARQL setting. For example, consider a question about Oscar winners from Europe. In the standard setting, we would rank people like Bruce Willis, Anthony Hopkins, Roman Polanski, or Ingrid Bergman as top results. But suppose the user has previously expressed strong interest in music, e.g., by browsing information on contemporary composers. Then the personalized search result should prefer people like Ennio Morricone, Hans Zimmer, or Javier Navarrete (all of whom won Oscars for film music). Now consider that the query should also return salient movies by these European Oscar winners. Returning only movies with Ennio Morricone's music on the top ranks would be rather monotone, even if this is the user's favorite composer. Instead, it is highly desirable to have more diversity in the top-ranked results.

The querying and ranking models presented in this paper have shown very good results so far. The next step will be to extend our framework to address the above open issues.

References

[1] S. Amer-Yahia, M. Lalmas: XML Search: Languages, INEX and Scoring. SIGMOD Record 35(4), 2006

[2] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, Z. G. Ives: DBpedia: A Nucleus for a Web of Open Data. ISWC/ASWC 2007

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, S. Sudarshan: Keyword Searching and Browsing in Databases using BANKS. ICDE 2002

[4] J. Breslin, A. Passant, S. Decker: The Social Semantic Web, Springer, 2009

[5] C. Bizer, T. Heath, T. Berners-Lee: Linked Data - The Story So Far. To appear in: International Journal on Semantic Web and Information Systems (IJSWIS), 2010.

[6] S. Chaudhuri, G. Das, V. Hristidis, G. Weikum: Probabilistic Information Retrieval Approach for Ranking of Database Query Results. ACM Trans. Database Syst. 31(3), 2006

[7] W. B. Croft, D. Metzler, T. Strohman: Search Engines - Information Retrieval in Practice. Addison-Wesley, 2009

[8] S. Elbassuoni, M. Ramanath, R. Schenkel, M. Sydow, G. Weikum: Language-model-based Ranking for Queries on RDF-Graphs. CIKM 2009

[9] G. Kasneci, F. Suchanek, G. Ifrim, M. Ramanath, G. Weikum: NAGA: Searching and Ranking Knowledge. ICDE 2008

[10] Z. Nie, Y. Ma, S. Shi, J.-R. Wen, W.-Y. Ma: Web Object Retrieval. WWW 2007

[11] D. Petkova, W. B. Croft: Hierarchical Language Models for Expert Finding in Enterprise Corpora. ICTAI 2006

[12] P. Serdyukov, D. Hiemstra: Modeling Documents as Mixtures of Persons for Expert Finding. ECIR 2008

[13] F. Suchanek, G. Kasneci, G. Weikum: YAGO: a Core of Semantic Knowledge. WWW 2007

[14] UniProt: Universal Protein Resource, `http://www.uniprot.org/`

[15] D. Vallet, H. Zaragoza: Inferring the Most Important Types of a Query: a Semantic Approach. SIGIR 2008

[16] C. Zhai, J.D. Lafferty: A risk minimization framework for information retrieval. Inf. Process. Manage. 42(1), 2006

Weighted Set-Based String Similarity

Marios Hadjieleftheriou
AT&T Labs–Research
marioh@research.att.com

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

Abstract

Consider a universe of tokens, each of which is associated with a weight, and a database consisting of strings that can be represented as subsets of these tokens. Given a query string, also represented as a set of tokens, a weighted string similarity query identifies all strings in the database whose similarity to the query is larger than a user specified threshold. Weighted string similarity queries are useful in applications like data cleaning and integration for finding approximate matches in the presence of typographical mistakes, multiple formatting conventions, data transformation errors, etc. We show that this problem has semantic properties that can be exploited to design index structures that support very efficient algorithms for query answering.

1 Introduction

Arguably, one of the most important primitive data types in modern data processing is strings. Short strings comprise the largest percentage of data in relational database systems, long strings are used to represent protein and DNA sequences in biological applications, as well as HTML and XML documents on the web. Searching through string datasets is a fundamental operation in almost every application domain, for example SQL query processing, information retrieval, genome research, product search in eCommerce applications, and local business search on online maps. Hence, a plethora of specialized indexes, algorithms, and techniques have been developed for searching through strings.

Due to the complexity of collecting, storing and managing strings, string datasets almost always contain representational inconsistencies due to typographical mistakes, multiple formatting conventions, data transformation errors, etc. Most importantly, string keyword queries more often than not contain errors as well. Even though exact string matching has been studied extensively in the past and a variety of efficient string searching algorithms have been developed, it is clear that approximate string matching (based on some notion of string similarity) is fundamental for retrieving the most relevant results for a given query, and ultimately improving user satisfaction. For that purpose, various *string similarity* functions have been proposed in the literature.

This article discusses indexing techniques and algorithms based on *inverted indexes* and a variety of *weighted set-based* string similarity functions, that can be used for efficiently answering *all-match selection queries*, i.e., queries which ask for all data strings whose similarity with the query string is larger than a user specified threshold.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 Set-Based Similarity

Set-based similarity decomposes strings into sets of tokens (such as words or n-grams) using a variety of tokenization methods and evaluates similarity of strings with respect to the similarity of their sets. A variety of set similarity functions can be used for this purpose, all of which have a similar flavor: each set element (or token) is assigned a weight and the similarity between the sets is computed as a weighted function of the tokens in each of the sets and in their intersection. The application characteristics heavily influence the tokens to extract, the token weights, and the similarity function used.

2.1 Similarity Functions

Strings are modeled as sets of tokens from a known token set Λ. Let $s = \{a_1, a_2, \ldots, a_n\}, a_i \in \Lambda$ be a string consisting of n tokens. Let $W : \Lambda \to \mathbb{R}^+$ be a function that assigns a positive real weight value to each token. A simple function to evaluate the similarity between two strings is the weighted intersection of their token sets.

Definition 1 (Weighted Intersection on Sets): Let $s = \{a_1, \ldots, a_n\}$, $r = \{b_1, \ldots, b_m\}$, $a_i, b_i \in \Lambda$ be two sets of tokens. The weighted intersection of s and r is defined as $\mathcal{I}(s, r) = \sum_{a \in s \cap r} W(a)$.

The definition above does not take into account the weight or number of tokens that the two strings do not have in common. In some applications, the strings are required to have similar lengths (i.e., similar number of tokens or similar total token weight). One could use various forms of normalization to address this issue.

Definition 2 (Normalized Weighted Intersection, Jaccard Similarity, Dice Similarity, Cosine Similarity): Let $s = \{a_1, \ldots, a_n\}$, $r = \{b_1, \ldots, b_m\}$, $a_i, b_i \in \Lambda$ be two sets of tokens. Let $\|s\|_1 = \sum_{i=1}^{n} W(a_i)$ (i.e., the L_1-norm), and $\|s\|_2 = \sqrt{\sum_{i=1}^{n} W(a_i)^2}$ (i.e., the L_2-norm).

The *normalized weighted intersection* of s and r is defined as $\mathcal{N}(s, r) = \frac{\|s \cap r\|_1}{\max(\|s\|_1, \|r\|_1)}$.

The *Jaccard similarity* of s and r is defined as $\mathcal{J}(s, r) = \frac{\|s \cap r\|_1}{\|s \cup r\|_1} = \frac{\|s \cap r\|_1}{\|s|_1 + |r|_1 - |s \cap r\|_1}$.

The *Dice similarity* of s and r is defined as $mathcalD(s, r) = \frac{2\|s \cap r\|_1}{\|s\|_1 + \|r\|_1}$.

The *Cosine similarity* of s and r is defined as: $\mathcal{C}(s, r) = \frac{[\|s \cap r\|_2]^2}{\|s\|_2 * \|r\|_2}$.

Clearly, Normalized Weighted Intersection, Jaccard, Dice and Cosine similarity are strongly related in a sense that they normalize the similarity with respect to the weights of the token sets, and result in a similarity value between 0 and 1. The similarity is minimized (i.e., equal to 0) only if the two sets share no tokens in common, and is maximized (i.e., equal to one) only if the two sets are the same. In general, the larger the weight of the tokens that the two strings do not have in common, the smaller is their similarity. Which function works better depends heavily on application and data characteristics.

2.2 Token Weights

An important consideration for weighted similarity functions is the token weighting scheme used. The simplest weighting scheme is to use unit weights for all tokens. A more meaningful weighting scheme though, should assign larger weights to tokens that carry more information content. As usual, the information content of a token is application and data dependent. For example, a specific sequence of phonemes might be very rare in the English language but a common sequence in Greek. A commonly used weighting scheme for text processing is based on inverse document frequency weights.

Definition 3 (Inverse Document Frequency Weight): Consider a set of strings D and a universe of tokens Λ. Let $df(a), a \in \Lambda$ be the number of strings $s \in D$ that have at least one occurrence of a. The inverse document frequency weight of a is defined as $idf(a) = \log(1 + \frac{|D|}{df(a)})$.

String	Tokens			
s_1	Children's	Food	Fund	
s_2	One	Laptop	per	Child
s_3	Feed	the	Children	
s_4	International	Children's	Found	
s_5	UNICEF			

Token	String Identifiers
child	s_1, s_2, s_3, s_4
food	s_1, s_3
found	s_4
fund	s_1
international	s_4
one	s_2
laptop	s_2
unicef	s_5

Table 7: (a) A collection of strings represented as sets of tokens. (b) An inverted representation of these sets of tokens. It is assumed that common stop words and special characters have been removed, stemming of words has occurred, and all tokens have been converted to lowercase.

Alternative definitions of idf weights are also possible. Nevertheless, they all have a similar flavor. The idf weight is related to the probability that a given token a appears in a random string $s \in D$. Frequent tokens have a high probability of appearing in many strings, hence they are assigned small weights. Infrequent tokens have a small probability of appearing in any string, hence they are assigned large weights. The intuition is that two strings that share a few infrequent tokens must have a high similarity.

2.3 Selection Queries

We consider all-match selection queries, which return all data strings whose similarity with the query string is larger than a user specified threshold.

Definition 4 (All-Match Selection Query): Given a string similarity function $\mathcal{S}$, a set of strings D, a query string q, and a positive threshold τ, the result of an *all-match selection query* is the answer set $A = \{s \in D : \mathcal{S}(q, s) \geq \tau\}$.

3 Algorithms for Set-Based Similarity

The brute-force approach for evaluating selection queries requires computing all pairwise similarities between the query string and the data strings, which can be expensive. A significant savings in computation cost can be achieved by using an index structure (resulting in a typical computation cost versus storage cost trade-off).

All the set-based similarity functions introduced in Section 2 can be computed easily using an inverted index. First, the data strings are tokenized and an inverted index is built on the resulting tokens, one list per token; an example using word tokens is shown in Table 7. Each list element is simply a string identifier. Depending on the algorithm used to evaluate the chosen similarity function, the lists can be stored sequentially as a flat file, using a B-tree, or a hash table. They can also be sorted according to string identifiers or any other information stored in the lists (e.g., the L_p-norm of the strings) as will become clear shortly.

All-match selection queries can be answered quite efficiently using an inverted index. Recall that an all-match selection query retrieves all data strings whose similarity with the query exceeds a query threshold τ. To answer the query, first the query string is tokenized using exactly the same tokenization scheme that was used for the data strings. The data strings satisfying the similarity threshold need to have at least one token in common with the query, hence only the token lists corresponding to query tokens need to be involved in the search. This results in significant pruning compared to the brute-force approach.

3.1 Sorting in String Identifier Order

Let $q = \{q_1, \ldots, q_n\}$ be a query string and $L_q = \{l_1, \ldots, l_n\}$ be the n lists corresponding to the query tokens. By construction, each list l_i contains all string identifiers of strings $s \in D$ s.t. $q_i \in s$. The simplest algorithm for evaluating the exact similarity between the query and all the strings in L_q is to perform a `multiway merge` on the string identifiers over the lists to compute all intersections $q \cap s$. This will directly yield the *weighted intersection similarity*. Clearly, the `multiway merge` computation can be performed very efficiently if the lists are already sorted in increasing order of string identifiers. However, the algorithm has to exhaustively scan all lists in L_q in order to identify all strings with similarity exceeding threshold τ.

Computing *Normalized Weighed Intersection* is also straightforward, provided that the L_1-norm of each data string is stored in the token lists. Hence, the lists store tuples (string-identifier, L_1-norm). Once again, if lists are sorted in increasing order of string identifiers, a `multiway merge` algorithm can evaluate the normalized weighted intersection between the query and all relevant strings very efficiently. Similar arguments hold for Jaccard, Dice and Cosine similarity.

3.2 Sorting in L_p-norm Order

The `multiway merge` algorithm has to exhaustively scan all the lists in L_q in order to compute the similarity score of every data string and determine whether the similarity exceeds threshold τ. For Normalized Weighed Intersection similarity we make the following observation:

Lemma 5 (Normalized Weighted Intersection L_1-norm Filter): Given sets s, r and Normalized Weighted Intersection threshold $\tau \in (0, 1]$ the following holds:

$$\mathcal{N}(s, r) \geq \tau \Leftrightarrow \tau \|r\|_1 \leq \|s\|_1 \leq \frac{\|r\|_1}{\tau}.$$

Essentially, we can use the L_1-norm filter to prune strings appearing in any list in L_q without the need to compute the actual similarity with the query. For that purpose, we sort token lists in increasing order of the L_1-norms of strings, rather than in string identifier order. Using Lemma 5, we can directly skip over all candidate strings with L_1-norm $\|s\|_1 < \tau\|q\|_1$ (where $\|q\|_1$ is the L_1-norm of the query) and stop scanning a list whenever we encounter the first candidate string with L_1-norm $\|s\|_1 > \frac{\|q\|_1}{\tau}$. Given that the lists are not sorted in increasing string identifier order, there are two options available: The first is to sort the strings within the appropriate norm intervals in string identifier order and perform the `multiway merge` in main memory. The second is to use classic `threshold` based algorithms to compute the similarity of each string to the query string.

`Threshold` algorithms utilize special terminating conditions that enable processing to stop before exhaustively scanning all token lists. This is easy to show if the similarity function is a monotone aggregate function. Let $w_i(s, q) = \frac{W(q_i)}{\max(\|s\|_1, \|q\|_1)}$ be the partial similarity of data string s and query token q_i. Then, $\mathcal{N}(s, q) = \sum_{q_i \in s \cap q} w_i(s, q)$. It can be shown that $\mathcal{N}(s, q)$ is a monotone aggregate function, i.e., $\mathcal{N}(s, q) \leq \mathcal{N}(s', q)$ if $\forall i \in [1, n] : w_i(s, q) \leq w_i(s', q)$.

There are three `threshold` algorithm flavors. The first scans lists sequentially and in a round robin fashion and computes the similarity of strings incrementally. The second uses random accesses to compute similarity aggressively every time a new string identifier is encountered in one of the lists. The third uses combination strategies of both sequential and random accesses.

Since the sequential round robin algorithm computes similarity incrementally, it has to temporarily store in main memory information about strings whose similarity has only been partially computed. Hence, the algorithm has a high book keeping cost, given that the candidate set needs to be maintained as a hash table organized by string identifiers. The aggressive random access based algorithm computes the similarity of strings in one step and hence does not need to store any information in main memory. Hence it has a very small book keeping cost, but on the other hand it has to perform a large number of random accesses to achieve this. This could be a drawback on traditional storage devices (like hard drives) but unimportant in modern, solid state based devices, like flash storage. Combination strategies try to strike a balance between low book keeping and a small number of random accesses. A simple round robin strategy is shown in Algorithm 3.1.

Algorithm 3.1: NRA(q, τ)

Tokenize the query: $q = \{q_1, \ldots, q_n\}$
M is a map of $(\ddot{s}, \mathcal{N}_s, B_1, \ldots B_n])$ tuples (B_i are bits, $\ddot{s}$ is the string identifier)
for $i \leftarrow 1$ **to** n
  do seek l_i to first element s.t. $\|s\|_1 \geq \tau\|q\|_1$
while not all lists l_i are inactive
  $f = \infty, w = 0$
  for $i \leftarrow 1$ **to** n
  do
    if l_i is inactive : *Continue*
      else $(\ddot{s}, \|s\|_1) \leftarrow l_i$.next
    if $\|s\|_1 > \frac{\|q\|_1}{\tau}$: Flag l_i as inactive and *Continue*
    $f = \min(f, \|s\|_1), w = w + W(q_i)$
    $(\ddot{s}, \mathcal{N}_s, B_1, \ldots, B_n) \leftarrow M$.find$(\ddot{s})$
    for $j \leftarrow 1$ **to** n
      do if l_j is inactive : $B_j = 1$
    if $\ddot{s} \in M$
      then
        if $B_1 = 1, \ldots, B_n = 1\}$ **and** $\mathcal{N}_s + W(q_i) > \tau \max(\|s\|_1, \|q\|_1)$
          then report $\ddot{s}, \mathcal{N}_s + W(q_i)$
          else $M \leftarrow (\ddot{s}, \mathcal{N}_s + W(q_i), B_1, \ldots, B_n)$
      else $\{M \leftarrow (\ddot{s}, W(q_i), B_1, \ldots, B_n)$
  for all $(\ddot{s}, \mathcal{N}_s, B_1, \ldots, B_n) \in M$
  do
    for $j \leftarrow 1$ **to** n
      do if l_j is inactive : $B_j = 1$
    if $B_1 = 1, \ldots, B_n = 1$
      then
        if $\mathcal{N}_s > \tau \max(\|s\|_1, \|q\|_1)$
          then report $\ddot{s}, \mathcal{N}_s$
          else remove from M
  if $\frac{w}{\max(f, \|q\|_1)} < \tau$ **and** M is empty : *Stop*

The algorithm keeps a candidate set M containing tuples consisting of a string identifier, a partial similarity score, and a bit vector containing zeros for all query tokens that have not matched with the particular string identifier yet, and ones for those that a match has been found. The candidate set is organized as a hash table on string identifiers for efficiently determining whether a given string has already been encountered or not. Lemma 5 states that for each list we only need to scan elements within a narrow L_1-norm interval. We skip directly to the first element in every list with L_1-norm $\|s\|_1 \geq \tau\|q\|_1$. The algorithm proceeds by reading one element per list in every iteration, from each active list. According to Lemma 5, if the next element read from list l_i has L_1-norm larger than $\frac{\|q\|_1}{\tau}$ the algorithm flags l_i as inactive. Otherwise, if the string identifier read is already contained in the candidate set M, its entry is updated to reflect the new partial similarity score. In addition, the bit vector is updated to reflect the fact that the candidate string contains the particular query token. Also, the algorithm checks whether any other lists have already become inactive, which implies one of two things: If the candidate string contains the query token, then the bit vector is already set to one and the partial similarity score has been updated to reflect that fact; or the candidate string does not contain the query token, hence the bit vector can be set to one without updating the partial similarity score (essentially adding zero to the similarity score). Finally, if the bit vector is fully set and the similarity score exceeds the query threshold, the string is reported as an answer. If the new string read is not already contained in M, a new entry is created and reported as an answer or inserted in M, using reasoning similar to the previous step. After one round robin iteration, the bit vector of each candidate in the candidate set is updated, and candidates with fully set bit vectors are reported as answers or evicted from the candidate set accordingly.

The algorithm can terminate early if two conditions are met. First the candidate set is empty, which means no more viable candidates whose similarity has not been completely computed yet exist. Second, the maximum possible score of a conceptual frontier string that appears at the current position in all lists cannot exceed the desired threshold.

Lemma 6: Let $L_a \subseteq L_q$ be the set of active lists. The terminating condition

$$\mathcal{N}^f = \frac{\sum_{l_i \in L_a} W(q_i)}{\max(\min_{l_i \in L_a} \|f_i\|_1, \|q\|_1)} < \tau,$$

does not lead to any false dismissals.

It is easy to see that a tighter bound for $\mathcal{N}^f$ exists. This can be achieved by examining the L_1-norm of all frontier elements simultaneously and is based on the observation that

Observation 7: Given a string s with L_1-norm $\|s\|_1$ and the L_1-norms of all frontier elements $\|f_i\|_1$ we can immediately deduce whether s potentially appears in list l_i or not by a simple comparison: If $\|s\|_1 < \|f_i\|_1$ and s has not been encountered in list l_i yet, then s does not appear in l_i (given that lists are sorted in increasing order of L_1-norms).

Let l_j be a list s.t. $\|f_j\|_1 = \min_{l_i \in L_a} \|f_i\|_1$. Based on Observation 7, a conceptual string s with L_1-norm $\|s\|_1 = \|f_j\|_1$ can appear only in list l_j. Thus

Lemma 8: Let $L_a \subseteq L_q$ be the set of active lists. The terminating condition

$$\mathcal{N}^f = \max_{l_i \in L_a} \frac{\sum_{l_j \in L_a : \|f_j\|_1 \leq \|f_i\|_1} W(q_j)}{\max(\|f_i\|_1, \|q\|_1)} < \tau,$$

does not lead to any false dismissals.

We can also use Observation 7 to improve the pruning power of the `nra` algorithm by computing a best-case similarity for all candidate strings before and after they have been inserted in the candidate set. Strings whose best-case similarity is below the threshold can be pruned immediately, reducing the memory requirements and the book keeping cost of the algorithm. The best-case similarity of a new candidate uses Observation 7 to identify lists that do not contain that candidate.

Lemma 9: Given query q, candidate string s, and a subset of lists $L_{q'} \subseteq L_q$ potentially containing s (based on Observation 7 and the frontier elements f_i), the best-case similarity score for s is:

$$\mathcal{N}^b(s, q) = \frac{\|q'\|_1}{\max(\|s\|_1, \|q\|_1)}.$$

An alternative `threshold` algorithm strategy is to assume that token lists are sorted in increasing L_1-norms for efficient sequential access, but that there also exists one index per token list on the string identifiers. Then, a random access algorithm can scan token lists sequentially and for every element probe the remaining lists using the string identifier index to immediately compute the exact similarity of the string. Similar terminating conditions as those used for the `nra` algorithm can be derived for the random access only algorithm.

The last threshold based strategy is to use a combination of `nra` and random accesses. We run the `nra` algorithm as usual but after each iteration we do a linear pass over the candidate set and use random accesses to compute the exact similarity of the strings and empty the candidate set.

It turns out that exactly the same length based algorithms can be used for Dice and Cosine similarity. It is easy to show that Dice similarity is a monotone aggregate function with partial similarity defined as $w_i(s, q) = \frac{W(q_i)}{\|s\|_1 + \|q\|_1}$. Hence, all `threshold` algorithms presented above can be adapted to Dice. In addition, it is easy to prove the following lemmas.

Lemma 10 (Dice L_1-norm Filter): Given sets s, r and Dice similarity threshold $\tau \in (0, 1]$ the following holds:

$$\mathcal{D}(s, r) \geq \tau \Leftrightarrow \frac{\tau}{2 - \tau}\|r\|_1 \leq \|s\|_1 \leq \frac{2 - \tau}{\tau}\|r\|_1.$$

Lemma 11: Let $L_a \subseteq L_q$ be the set of active lists. The terminating condition

$$\mathcal{D}^f = \max_{l_i \in L_a} \frac{\sum_{l_j \in L_a : \|f_j\|_1 \leq \|f_i\|_1} 2W(q_i)}{\|f_i\|_1 + \|q\|_1} < \tau,$$

does not lead to any false dismissals.

Lemma 12: Given query q, candidate string s, and a subset of list $L_{q'} \subseteq L_q$ potentially containing s, the best-case similarity score for s is:

$$\mathcal{D}^b(s, q) = \frac{2\|q'\|_1}{\|s\|_1 + \|q\|_1}.$$

Exactly the same arguments hold for cosine similarity as well. Recall that cosine similarity is computed based on the L_2-norm of the strings. Once again, an inverted index is built where this time each list element contains tuples (string-identifier, L_2-norm). Clearly, Cosine similarity is a monotone aggregate function with partial similarity $w_i(s, q) = \frac{W(q_i)^2}{\|s\|_2\|q\|_2}$. An L_2-norm filter also holds.

Lemma 13 (Cosine similarity L_2-norm Filter): Given sets s, r and Cosine similarity threshold $\tau \in (0, 1]$ the following holds:

$$\mathcal{C}(s, r) \geq \tau \Leftrightarrow \tau\|r\|_2 \leq \|s\|_2 \leq \frac{\|r\|_2}{\tau}.$$

In addition

Observation 14: Given a string s with L_2-norm $\|s\|_2$ and the L_2-norms of all frontier elements $\|f_i\|_2$ we can immediately deduce whether s potentially appears in list l_i or not by a simple comparison: If $\|s\|_2 < \|f_i\|_2$ and s has not been encountered in list l_i yet, then s does not appear in l_i (given that lists are sorted in increasing order of L_2-norms).

Hence

Lemma 15: Let $L_a \subseteq L_q$ be the set of active lists. The terminating condition

$$\mathcal{C}^f = \max_{l_i \in L_a} \frac{\sum_{l_j \in L_a : \|f_j\|_2 \leq \|f_i\|_2} W(q_i)^2}{\|f_i\|_2 * \|q\|_2} < \tau,$$

does not lead to any false dismissals.

Lemma 16: Given query q, candidate string s, and a subset of lists $L_{q'} \subseteq L_q$ potentially containing s, the best-case similarity score for s is:

$$\mathcal{C}^b(s,q) = \frac{[\|q'\|_2]^2}{\|s\|_2 * \|q\|_2}.$$

The actual algorithms in principle remain the same.

Jaccard similarity presents some difficulties. Notice that Jaccard is a monotone aggregate function with partial similarity $w_i(s,q) = \frac{W(q_i)}{\|s \cup q\|_1}$. Nevertheless, we cannot use this fact for designing termination conditions for the simple reason that w_i's cannot be evaluated on a per token list basis since $\|s \cup q\|_1$ is not known in advance (knowledge of $\|s \cup q\|_1$ implies knowledge of the whole string s and hence knowledge of $\|s \cap q\|_1$ which is equivalent to directly computing the similarity). Recall that an alternative expression for Jaccard is $\mathcal{J}(s,q) = \frac{\|s \cap q\|_1}{\|s\|_1 + \|q\|_1 - \|s \cap q\|_1}$. This expression cannot be decomposed into aggregate parts on a per token basis, and hence is not useful either. Nevertheless, we can still prove various properties of Jaccard that enable us to use all `threshold` based algorithms. In particular

Lemma 17 (Jaccard L_1-norm Filter): Given sets s, r and Jaccard similarity threshold $\tau \in (0, 1]$ the following holds:

$$\mathcal{J}(s,r) \geq \tau \Leftrightarrow \tau\|r\|_1 \leq \|s\|_1 \leq \frac{\|r\|_1}{\tau}.$$

Lemma 18: Let $L_a \subseteq L_q$ be the set of active lists. Let $I_i = \sum_{l_j \in L_a : \|f_j\|_1 \leq \|f_i\|_1} W(q_i)$. The terminating condition

$$\mathcal{J}^f = \max_{l_i \in L_a} \frac{I_i}{\|f_i\|_1 + \|q\|_1 - I_i} < \tau,$$

does not lead to any false dismissals.

Lemma 19: Given query q, candidate string s, and a subset of list $L_{q'} \subseteq L_q$ potentially containing s, the best-case similarity score for s is:

$$\mathcal{J}^b(s,q) = \frac{\|q'\|_1}{\|s\|_1 + \|q\|_1 - \|q'\|_1}.$$

Generally speaking, comparing the *threshold* based algorithms with the *multiway merge* algorithm, the relative cost savings between sorting according to string identifiers or sorting according to the L_p-norm heavily depends on a variety of factors most important of which are: the algorithm used, the specific query (whether it contains tokens with very long lists for which the L_p-norm filtering will yield significant pruning), the overall token distribution of the data strings, the storage medium used to store the lists (e.g., disk drive, main memory, solid state drive), and the type of compression used for the lists, if any.

It is important to state here that most list compression algorithms are designed for compressing string identifiers, which are natural numbers. The idea being that natural numbers in sorted order can easily be represented by delta coding. In delta coding only the first number is stored, and each consecutive number is represented by its difference with the previous number. This representation decreases the magnitude of the numbers stored (since deltas are expected to have small magnitude when the numbers are in sorted order) and enables very efficient compression. A significant issue with all normalized similarity functions is that, given arbitrary token weights, token lists have to store real valued L_1-norms (or L_2-norms for Cosine similarity). Lists containing real valued attributes cannot be compressed as efficiently as lists containing only natural numbers.

3.3 Processing Lists in Token Weight Order

Recall that the `threshold` based algorithms scan lists in a random order, e.g., in the order that tokens appear in the query string. However, the distribution of tokens in the data strings can help determine a more meaningful ordering that can yield significant performance benefits. The idea is based on the observation that the order and the strategy used to scan the lists affects the frontier elements, and hence the pruning power of subsequent steps.

Consider Normalized Weighted Intersection first. Let the lists in L_q be sorted according to their respective token weights, from heaviest to lightest. Without loss of generality, let this ordering be $L_q = \{l_1, \ldots, l_n\}$ s.t. $W(q_1) \geq \ldots \geq W(q_n)$. Since $w_1(s,q) \geq \ldots \geq w_n(s,q)$, the most promising candidates appear in list l_1; the second most promising appear in l_2; and so on. This leads to an alternative sequential algorithm, which we refer to here as `heaviest first`. The algorithm exhaustively reads the heaviest token list first, and stores all strings in a candidate set. The second to heaviest list is scanned next. The similarity of strings that have already been encountered in the previous list is updated, and new candidates are added in the candidate set, until all lists have been scanned and the similarity of all candidates has been computed. While a new list is being traversed the algorithm prunes the candidates in the candidate set whose best-case similarity is below the query threshold. The best-case similarity for every candidate is computed by taking the partial similarity score already computed for each candidate after list l_i has been scanned, and assuming that the candidate exists in all subsequent lists $l_{i+1}, \ldots, l_n$.

The important observation here is that we can compute a tighter L_1-norm filtering bound each time a new list is scanned. The idea is to treat the remaining lists as a new query $q' = \{q_{i+1}, \ldots, q_n\}$ and recompute the bounds using Lemma 5. The intuition is that the most promising new candidates in lists $l_{i+1}, \ldots, l_n$, are the ones containing all tokens $q_{i+1}, \ldots, q_n$ and hence potentially satisfying $s = q'$. Care needs to be taken though in order to accurately complete the partial similarity scores of all candidates already inserted in the candidate set from previous lists, which can have L_1-norms that do not satisfy the recomputed bounds for q'. The algorithm identifies the largest L_1-norm in the candidate set and scans subsequent lists using that bound. If a string already appears in the candidate set its similarity is updated. If a string does not appear in the candidate set (this is either a new string or an already pruned string) then the string is inserted in the candidate set if and only if its L_1-norm satisfies the recomputed bounds based on q'.

To reduce the book keeping cost of the candidate set, the algorithm stores the set as a linked list, sorted primarily in increasing order of L_1-norm and secondarily in increasing order of string identifiers. Then the algorithm can do a merge join of the currently processed token list and the candidate list very efficiently in one pass. The complete algorithm is shown in Algorithm 3.2.

The biggest advantage of the `heaviest first` algorithm is that it can benefit from long sequential accesses, one list at a time. Notice that both the `nra` and the `multiway merge` strategies have to access lists in parallel. For traditional disk based storage devices, as the query size increases and the number of lists that need to be accessed in parallel increases, a larger buffer is required in order to prefetch entries from all the lists and the head seek time increases, moving from one list to the next (this is not an issue for solid state devices).

Notice that the `heaviest first` strategy has another advantage when token weights are assigned according to inverse token popularity, as is the case for idf weights. In that case, the heaviest tokens are rare tokens that correspond to the shortest lists and the `heaviest first` strategy is equivalent to scanning the lists in order of their length, shortest list first. The advantage is that this keeps the size of the candidate set to a minimum, since as the algorithm advances to lighter tokens, the probability that newly encountered strings (strings that do not share any of the heavy tokens with the query) will satisfy the similarity threshold significantly decreases. Such strings are therefore immediately discarded. In addition, the algorithm terminates with high probability before long lists have to be examined exhaustively.

It is easy to see that the `heaviest first` strategy can be adapted straightforwardly to all other normalized similarity measures. Moreover, the `heaviest first` strategy can help prune candidates even for simple Weighted Intersection similarity, based on the observation that strings containing the heaviest query tokens are more likely to exceed the threshold.

Algorithm 3.2: HEAVIEST FIRST(q, τ)

Tokenize query: $q = \{q_1, \ldots, q_n\}$ s.t. $W(q_i) \geq W(q_j), i < j$
M is a list of $(\ddot{s}, \mathcal{N}_s, \|s\|_1)$ sorted in $\|s\|_1, \ddot{s}$ order
$\lambda = \sum_{i=1}^{n} W(q_i)$

for $i \leftarrow 1$ **to** n
do
- $\lambda = \lambda - W(q_i)$
- Seek to first element of l_i with $\|s\|_1 \geq \tau\lambda$
- $(\ddot{r}, \mathcal{N}_r, \|r\|_1) \leftarrow M$.last
- $\mu = \max(\|r\|_1, \frac{\lambda}{\tau}$
- $(\ddot{r}, \mathcal{N}_r, \|r\|_1) \leftarrow M$.first
- **while** not at end of l_i
  do
  - $(\ddot{s}, \|s\|_1) \leftarrow l_i$.next
  - **if** $\|s\|_1 > \mu$: *Break*
  - **while** $\|r\|_1 \leq \|s\|_1$ **and** $\ddot{r} \neq \ddot{s}$
    do
    - **if** $\mathcal{N}_r + \lambda \leq \tau \max(\|r|_1, \|q\|_1)$: Remove from M
    - $(\ddot{r}, \mathcal{N}_r, \|r\|_1) = M$.next
  - **if** $\ddot{r} = \ddot{s}$
    then $\{M \leftarrow (\ddot{r}, \mathcal{N}_r + W(q_i), \|r\|_1)$
    else if $W(q_i) + \lambda \geq \tau \max(\|s\|_1, \|q\|_1)$
    then Insert $(\ddot{s}, W(q_i), \|s\|_1)$ at current position in M

Let query string $q = \{q_1, \ldots, q_n\}$ and threshold $\tau \in (0, \sum_{i=1}^{n} W(q_i)]$, and assume without loss of generality that lists are sorted in decreasing order of token weights (i.e., $W(q_1) \geq \ldots \geq W(q_n)$). Assume that there exists a string s that is contained only in a suffix $l_k, \ldots, l_n$ of token lists, whose aggregate weight $\sum_{i=k}^{n} W(q_i) < \tau$. Clearly such a string cannot exceed the query threshold. An immediate conclusion is that

Lemma 20: Let query $q = q_1, \ldots, q_n$ s.t. $W(q_1) \geq \ldots \geq W(q_n)$, and threshold $\tau \in (0, \sum_{i=1}^{n} W(q_i)]$. Let $\pi = \arg\max_{1 \leq \pi \leq n} \sum_{i=\pi}^{n} W(q_i) \geq \tau$. Call $P(q) = \{q_1, \ldots, q_\pi\}$ the prefix of q and $S(q) = \{q_{\pi+1}, \ldots, q_n\}$ the suffix of q. Then, for any string s s.t. $s \cap P(q) = \emptyset, \mathcal{I}(s, q) < \tau$.

Hence, the only viable candidates have to appear in at least one of the lists in the prefix $l_1, \ldots l_\pi$. The `heaviest first` algorithm exhaustively scans the prefix lists to find all viable candidates and then probes the suffix lists to complete their scores. If an index on string identifiers is available on each of the lists in the suffix, the algorithm needs to perform, on average, one random access per candidate per list in $S(q)$. If an index is not available, then assuming that lists are sorted in increasing string identifier order, the algorithm keeps track of the smallest and largest string identifier in the candidate set, seeks to the first list element with identifier larger than or equal to the smallest identifier and scans each list as deep as the largest identifier. Alternatively, binary search can be performed. This strategy will help potentially prune a long head and tail from each list in the suffix, which can be very beneficial when token weights are assigned according to token popularity (e.g., idf weights) where the lightest tokens (the most popular ones) correspond to the longest lists (which by design become part of the suffix).

4 Index Updates

Typically, inverted indexes are used for mostly static data. More often than not, token lists are represented either by simple arrays in main memory, or flat files in secondary storage, since these representations are space efficient and also enable very fast sequential processing of list elements. On the other hand, inserting or deleting elements in the token lists becomes expensive.

In situations where fast updates are important, token lists can be represented by linked lists or binary trees in main memory, or B-trees in external memory. The drawback is the increased storage cost both for main memory and external memory structures, and also the fact that external memory data structures do not support sequential accesses over the complete token lists as efficiently as flat files. On the other hand, the advantage is that insertions and deletions can be performed very efficiently.

A subtle but very important point regarding updates arises when considering token lists that contain L_p-norms. Recall that L_p-norms are computed based on token weights. For certain token weighting schemes, like idf weights, the weights depend on the total number of strings and the number of occurrences of tokens within the strings. As strings get inserted, deleted, or updated, both the total number of strings and the number of occurrences of tokens might change, and as a result the L_p-norms of strings will change.

Consider for example idf weights, where the weight of a token is a function of the total number of strings $|D|$ (see Definition 3). A single string insertion or deletion will affect the idfs of all tokens, and hence render the L_p-norms of all entries in the inverted lists in need for updating. Similar effects occur when a string is updated, resulting in the document frequency of a set of tokens to change. Since the idf weight of tokens is also a function of document frequency, the L_p-norm of every string containing one or more of these tokens will change. Hence, a single string update can result in a cascading update effect on the token lists of the inverted index that could be very expensive.

To alleviate the cost of updates in such scenarios, a technique based on delayed propagation of updates has been proposed. The idea is to keep stale information in the inverted lists as long as certain guarantees on the answers returned can be provided. Once the computed L_p-norms have deviated far enough from the true values such that the guarantees no longer hold, the pending updates are applied in a batch fashion.

The algorithm essentially computes lower and upper bounds between which the weight of individual tokens can vary, such that a selection query with a well defined reduced threshold $\tau' < \tau$ will return all true answers (i.e., will not result in any false negatives). The additional cost is an increased number of false positives. The reduced threshold τ' is a function of τ, the particular weighting scheme, and the relaxation bounds. The tighter the relaxation bounds chosen, the smaller the number of false positives becomes, but the more frequently the inverted lists need to be batch updated.

5 Related Work

Baeza-Yates and Ribeiro-Neto [2] and Witten et al. [14] discuss selection queries for various similarity measures using `multiway merge` and inverted indexes. The authors also discuss various compression algorithms for inverted lists sorted according to string identifiers.

Various strategies for evaluating queries based on sorting strings according to their L_p-norms appear in Hadjieleftheriou et al. [8]. Similar observations using L_p-norms (in a more general context) have been made by Bayardo et al. [3]. L_p-norm based filtering has also been used by Xiao et al. [17].

A detailed analysis of threshold based algorithms is conducted by Fagin et al. [6]. Improved termination conditions for these algorithms are discussed by Sarawagi and Kirpal [13], Bayardo et al. [3] and Hadjieleftheriou et al. [8]. The `heaviest first` strategy was introduced by Hadjieleftheriou et al. [8]. The `heaviest first` algorithm for weighted intersection based on prefix and suffix lists is based on ideas introduced by Chaudhuri et al. [4].

Index construction and update related issues with regard to L_p-norm computation is discussed in detail by Hadjieleftheriou et al. [9].

Apart from selection queries, string similarity join queries have also received a lot of attention [1, 3, 4, 7, 13, 16, 17]. A related line of work has concentrated on edit-based (as opposed to set-based) similarity functions (e.g., edit distance) [1, 4, 5, 7, 10–12, 15, 17].

References

[1] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, May 1999.

[3] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[4] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.

[5] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *SIGMOD*, pages 707–718, 2009.

[6] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[7] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[8] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, 2008.

[9] M. Hadjieleftheriou, N. Koudas, and D. Srivastava. Incremental maintenance of length normalized indexes for approximate string matching. In *SIGMOD*, pages 429–440, 2009.

[10] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 371–380, 2009.

[11] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[12] C. Li, B. Wang, and X. Yang. Vgram: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314, 2007.

[13] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, pages 743–754, 2004.

[14] I. H. Witten, T. C. Bell, and A. Moffat. *Managing Gigabytes: Compressing and Indexing Documents and Images*. John Wiley & Sons, Inc., 1994.

[15] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

[16] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009.

[17] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

Search-As-You-Type: Opportunities and Challenges

Chen Li† and Guoliang Li‡
†Department of Computer Science, University of California, Irvine, USA
chenli@ics.uci.edu
‡Department of Computer Science and Technology, Tsinghua University, Beijing, China
liguoliang@tsinghua.edu.cn

Abstract

Traditional information systems return answers after a user submits a complete query. Users often feel "left in the dark" when they have limited knowledge about the underlying data, and have to use a try-and-see approach for finding information. A recent trend of supporting autocompletion in these systems is a first step towards solving this problem. In this paper, we study a new information-access paradigm, called "search-as-you-type" or "type-ahead search," in which the system searches the underlying data on the fly as the user types in query keywords. It extends traditional prefix-based autocompletion interfaces by supporting full-text search on the data using tokenzied query keywords. We give an overview of this information-access paradigm, discuss research challenges and opportunities, and report recently developed techniques.

1 Introduction

Traditional information systems allow users to compose and submit a query to retrieve relevant answers. This information-access paradigm requires the user to have certain knowledge about the structure and content of the underlying data repository. With limited knowledge about the data, a user often feels "left in the dark" when issuing queries, and the user and has to use a try-and-see approach for finding information. For instance, Figure 1 shows a traditional interface to search on the people directory of an organization. To find a person, a user needs to fill in the form by providing information for multiple attributes, such as name, phone, department, and title. If the user has limited information about the person she is looking for, such as the exact spelling of the person's name, the user needs to try a few possible keywords, go through the returned results, modify the keywords, and reissue a new query. She needs to repeat this step multiple times to find the person, if lucky enough. This search interface is neither efficient nor user friendly.

To solve this problem, many systems provide instant feedback as users formulate search queries. Most search engines and many online search forms support autocompletion, which shows suggested queries or even answers "on the fly" as a user types in a keyword query letter by letter. For instance, consider the Web search interface at Netflix (http://www.netflix.com/BrowseSelection), which allows a user to search for movies by their titles, actors, directors, and genres. If a user types in a partial query "`mad`", the system shows movies matching this keyword as a prefix, such as "`Madagascar`" and "`Mad Men: Season 1`".

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

The quick feedback helps the user not only in formulating the query, but also in understanding the underlying data. Most autocompletion systems make suggestions by simply treating a query as a *prefix* condition. As a result, a Netflix user cannot type in "`Spielberg sci-fi`" to find sci-fi movies directed by Steven Spielberg, since the system treats the query as a prefix string, which does not exist in any movie record.

Figure 1: A traditional directory-search form.

Recently a new type-ahead-search paradigm has arisen that generalizes prefix-based autocompletion. In this paradigm, a system supports *full-text* search on the underlying data to find answers as a user types in query keywords. We have developed several prototypes using this paradigm. The first one, called PSearch, supports search on the UC Irvine people directory. A screenshot is shown in Figure 2. In the figure, a user has typed in a query string "`professor smyt`." Even though the user has not typed in the second keyword completely, the system can already find person records that might be of interest to the user. Notice that the two keywords in the query string (including a partial keyword "`smyt`") can appear in different attributes of the records. In particular, in the first record, the keyword "`professor`" appears in the "title" attribute, and the partial keyword "`smyt`" appears in the "name" attribute. The matched prefixes are highlighted. The system also utilizes a-priori knowledge such as synonyms. For instance, given the fact that "`william`" and "`bill`" are synonyms, the system can find a person called "`William Kropp`" when the user has typed in "`bill crop`." This search prototype has been used regularly by many people at UCI, and received positive feedback due to the friendly user interface and high efficiency.

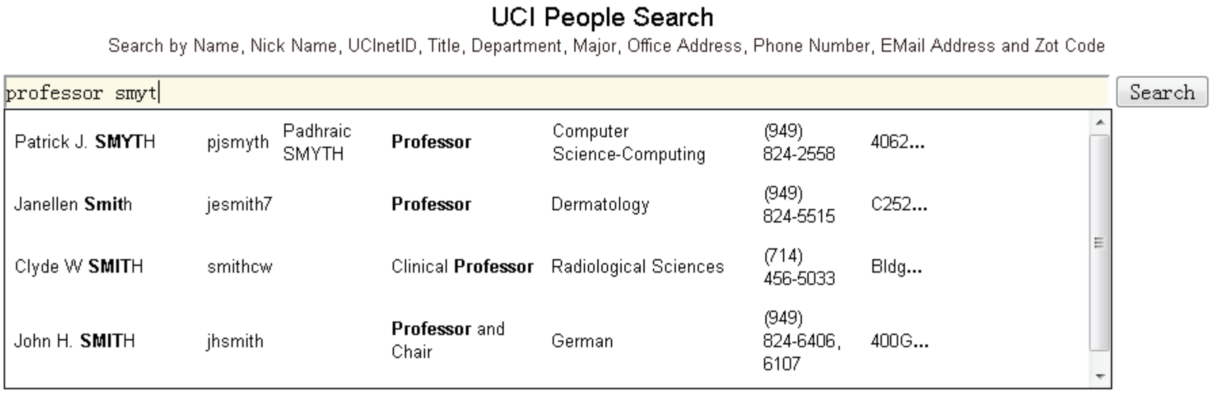

Figure 2: Fuzzy type-ahead search on the UC Irvine people directory (http://psearch.ics.uci.edu).

There are several other systems developed using this search paradigm: (1) The "Search" box on the page http://www.ics.uci.edu that searches on the people of the school of ICS at UCI and its important internal pages; (2) A system called "iPubMed" (http://ipubmed.ics.uci.edu) that supports interactive, fuzzy search on more than 18 million MEDLINE publications; (3) A prototype (http://tastier.cs.tsinghua.edu.cn/urlsearch/) that supports searches on 10 million popular URLs [11]; and (4) A search interface (http://fr.ics.uci.edu/haiti) for family reunification for the recent Haiti earthquake.

In this paper, we give an overview of this information-access paradigm, discuss research challenges and opportunities, and report recently developed techniques.

2 Overview of Type-ahead Search

Figure 3 illustrates a client-server architecture of a system supporting search-as-you-type. The underlying data residing on the server can be a relational database [10], a collection of documents, or XML data [9]. For simplicity, in this paper we focus on the case where the server has a set of relational records. The client has a browser, using which a user can send requests to the server to retrieve results. Each keystroke of the user could invoke a query, which includes the current string the user has typed in. The browser sends the query to the server.

The server tokenizes the query string, computes and returns to the user the best answers ranked by their relevancy to the query. For each query, the server treats the last keyword as a *partial keyword* the user is completing, and other earlier keywords as *complete keywords*. For a keyword query, we want to find the records that contain every complete keyword in the query and a keyword with the partial keyword as a prefix. For example, in Figure 2, for a keyword query "`professor smyt`", the keyword "`smyt`" is a partial keyword, while "`professor`" is a complete keyword. The first record in the figure is an answer since it contains keyword "`professor`" and a keyword "`smyth`" with the prefix "`smyt`".

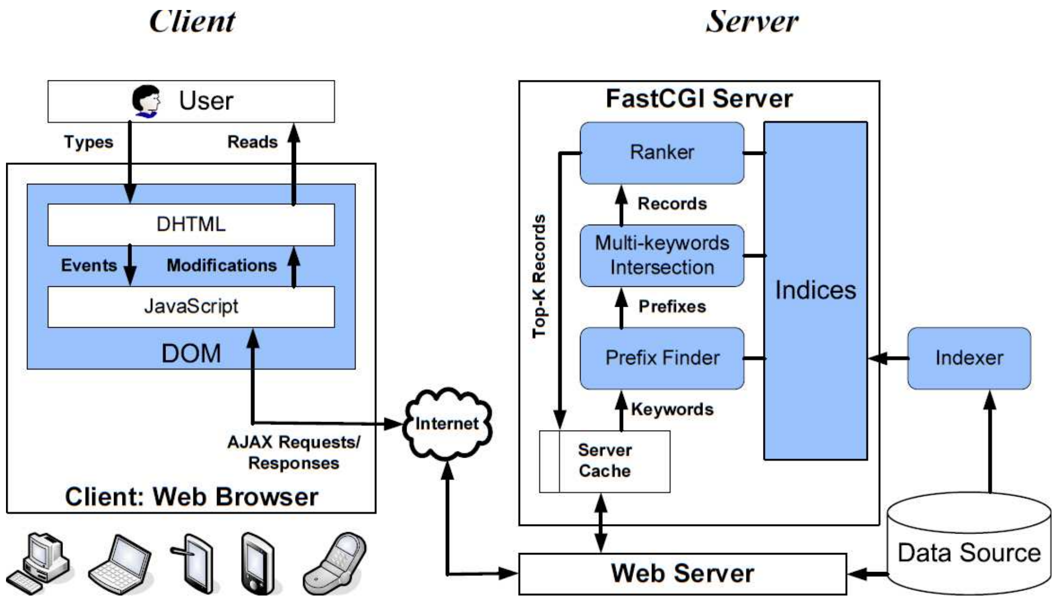

Figure 3: Search-as-you-type system architecture.

Supporting *fuzzy search* is very important especially when users do not remember the exact spelling of the right keywords. To support this feature, for each complete keyword in a query, we identify the keywords in the data that are similar to the keyword. For the partial keyword, we identify its *similar keywords* as those in the data with a prefix similar to the partial keyword. We compute the relevant records that contain a similar keyword for every query keyword. We can use edit distance to quantify the similarity between keywords. The *edit distance* between two words is the minimum number of edit operations (i.e., insertion, deletion, and substitution) of single characters needed to transform the first one to the second. For example, the edit distance of "`feloutose`" and "`faloutsos`" is 3. We say two keywords are *similar* if their edit distance is within a given threshold τ. This threshold could be proportional to the length of a query keyword to allow more errors for longer keywords. For example, suppose the edit-distance threshold $\tau = 1$. In Figure 2, for keyword query "`professor smyt`", the second record is an answer, since it contains keyword "`professor`" and a keyword "`smith`" with a prefix "`smit`" similar to input keyword "`smyt`".

2.1 Client

The client side contains HTML contents with JavaScript code executed in the browser. When the user types in a query, if there is no pending request being processed by the server, the JavaScript code issues an AJAX query to the server. Otherwise, it waits until the request has been answered. In this way, we can avoid overloading the server if the user types very fast.

2.2 Server

As shown in Figure 3, there are several components on the server side. We use a FastCGI module to store the data and indices. (Other programming languages such as Java Servlets are also possible.) The FastCGI server module is loaded once when the Web server starts, and continually handles queries without spawning more instances. The server loads the data and indices from disks, and searches on the data. The FastCgi Server waits for queries from the client, and caches query results. The Server Cache component checks whether the query can be answered using the cached results. If not, the server incrementally answers the query by using the cached information. For each query keyword, the Prefix Finder incrementally computes keywords of partial keywords. The Multi-keyword Intersection module computes the relevant answers. The Ranker module ranks the answers to identify the best answers. The Indexer component indexes the underlying data. Now we explain the modules in more detail.

Prefix Finder: For exact search, it finds the keywords with a prefix of the partial keyword. However, to support fuzzy search, we need to compute multiple prefixes that are similar to the partial keyword, and retrieve their corresponding complete keywords as the similar keywords. In the query "`professor smyt`", for exact search, this module finds complete keywords with the prefix "`smyt`", such as "`smyth`". For fuzzy search, it finds complete keywords with a prefix similar to "`smyt`," such as "`smyth`" and "`smith`".

Multi-keyword Intersection: This module takes the sets of keywords produced by the prefix finder as input (for multiple keywords), and computes the relevant answers, which contain a matching keyword from each set. For the partial keyword, there could be multiple keywords, and each similar partial prefix has multiple similar keywords. The union of the inverted lists of those keywords similar to one keyword is called the "union list" for this keyword. A straightforward method to identify the relevant answers is to first construct the union list for every query keyword, and compute the intersection of the union lists. In our running example, for exact search, based on keywords "`professor`" and "`smyth`", this module computes the intersection of inverted lists of the two keywords. For fuzzy search, we first compute the union lists of "`smyt`", which is the union of inverted lists of similar keywords of "`smyt`" (e.g., "`smyth`" and "`smith`"). Then we compute the intersection of the union lists of "`professor`" and "`smyt`". More efficient algorithms have been developed for solving this problem [7].

Ranker: In order to compute high-quality results, we need to use a good ranking function for the candidates. The function should consider various factors such as the similarity between a query keyword and its similar prefixes, the weight of each keyword, term frequencies, inverse document frequencies, importance of each record, etc.

Server Cache: After finding the answers to a query, we can cache some information, and incrementally answer the subsequent keyword queries using the cached information. For instance, suppose the results of the keyword query "`professor smyt`" have been cached. If the user types in one more letter and submits a new query "`professor smyth`", we can use the cached information to answer the new query.

Indexer: To improve performance, we can create index structures for efficiently answering type-ahead search, which will be discussed in the next section.

3 Research Challenges

There are several unique challenges in type-ahead search, mainly due to the requirement on a high interactive speed and the capability of relaxing keyword conditions. Each keystroke from the user can invoke a query on the backend server. The total round-trip time between the client browser and the backend server includes the network delay and data-transfer time, query-execution time on the server, and the javascript-execution time on the client browser. In order to achieve an interactive speed, this total time should be short (typically within 100ms). The query-execution time on the server should be even shorter. This high speed is more challenging to achieve when we need to relax the keywords on-the-fly. At a high level, a main challenge is: *in type-ahead search we need to support the features available in traditional search systems as the user types in keywords character by character.* Notice that in traditional autocompletion, we can easily recommend good queries by traversing a trie structure. However, when we allow keywords to appear at different places in the answers, the problem becomes "search the data on the fly," rather than just recommending a few queries. As a consequence, the join nature of the online-search problem can be computationally costly.

In this section, we present several recently developed techniques to address these challenges and discuss open problems that need more research investigation.

3.1 Efficient Indexing and Ranking

To facilite prefix search, we can construct a trie structure with inverted lists on the leaf nodes [3, 7]. Each word in the data set 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 the word. For each leaf node, we store an inverted list of IDs of records that contain the corresponding word. Figure 4 gives an example trie structure, in which node 12 has an inverted list of records 3 and 4, when records 3 and 4 contain the corresponding keywords "`lin`".

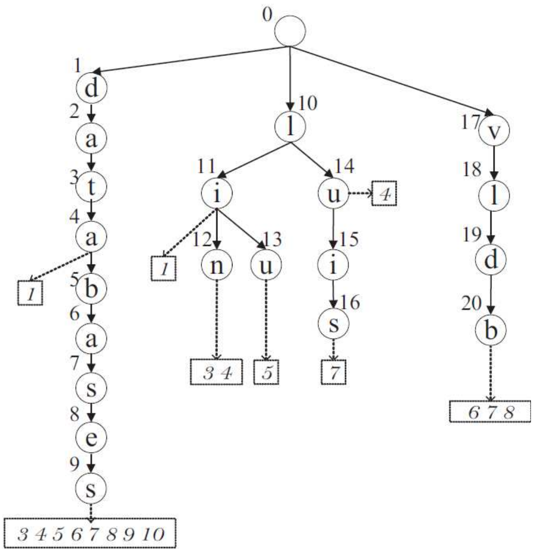

Figure 4: Trie structure with inverted lists on leaf nodes.

Exact Prefix Search: Given a partial keyword, we want to find the keywords with a prefix of the partial keyword. We can use the trie structure to answer prefix queries as follows. We first find the corresponding trie node of the prefix, and then traverse the subtrie to get the keywords with a prefix of the input keyword. Finally, we compute the records on the inverted lists of the leaf nodes. For example, consider the trie structure in Figure 4. When a user types in a keyword "lu", we find the trie node 14, and retrieve the corresponding leaf nodes "luis," and get record 7. When the user types in a new keyword "lui", we find trie node 15. In this way, we can use the trie structure to efficiently find the complete keywords of the partial keyword.

Fuzzy Prefix Search: For the case of exact prefix search using a trie index, there can be at most one trie node corresponding to a query prefix. The solution to the problem of fuzzy search is more challenging since a keyword prefix can have multiple similar prefixes, and we need to compute them efficiently. Traditional gram-based methods [8] are inefficient in this case due to their poor pruning power for short strings.

There are recent studies on supporting fuzzy prefix search efficiently [5, 7]. The idea is the following. When the user types in one more letter after a query prefix p, the similar prefixes of p can be used to compute the similar prefixes of the new query. Specifically, when the user types in one more letter after the partial keyword p, only the similar prefixes of p and their descendants could be similar prefixes of the new query keyword. We can use this property to incrementally compute the similar prefixes of a new query. For a new query, we first find similar prefixes of previous queries from the server cache, compute similar prefixes for the current query incrementally, and store the results in the cache for future computation. For example, consider the trie structure in Figure 4. Assume a user types in a query "nlis" letter by letter. Suppose two strings are considered to be similar if their edit distance is within 2. First, the similar prefixes of the empty string are nodes 0, 10, 11, and 14. When the user types in the first character "n", we compute its similar prefixes based on that of the empty string as follows. For node 0, since we can delete the letter "n", it is a similar prefix of "n". For node 10, which is a child of node 0 with a letter "l", as we can substitute "l" for "n", it is a similar prefix of "n". In this way, we can compute the similar prefixes of "n".

Multi-Keyword Intersection: The goal of multi-keyword intersection is to efficiently and incrementally compute the relevant records based on the keywords generated from the prefix finder. In [7] we studied the following problems. (1) *Intersection of multiple lists of keywords*: Each query keyword (treated as a prefix) has multiple predicted complete keywords, and the union of the lists of these complete keywords includes potential answers. The union lists of multiple query keywords need to be intersected in order to compute the answers to the query. These operations can be computationally costly, especially when each query keyword can have multiple similar prefixes. (2) *Cache-based incremental intersection*: Users tend to type in a query letter by letter. Thus we can use the cached results of earlier queries to answer a query incrementally. We use an example to illustrate how to cache query results and use them to answer subsequent queries. Suppose a user types in a keyword query Q_1 = "cs co". All the records in the answers to Q_1 are computed and cached. For a new query Q_2 = "cs conf" that appends two letters to the end of Q_1, we can use the cached results of Q_1 to answer Q_2, because the second keyword "conf" in Q_2 is more restrictive than the corresponding keyword "co" in Q_1. Each record in the cached results of Q_1 is verified to check if "conf" can appear in the record as a prefix. In this way, Q_2 does not need to be answered from scratch.

3.2 Ranking

A good ranking function needs to consider various factors to compute an overall relevance score of a record to a query. The following are important factors. (1) *Matching prefixes*: We consider the similarity between a query keyword and its best matching prefix. The more similar a record's matching keywords are to the query keywords, the higher this record should be ranked. The similarity is also related to keyword length. For example, when a user types in a keyword "circ", the word "circle" is possibly more similar to the query keyword than "circumstance". Therefore records containing the word "circle" could be ranked higher than those with the word "circumstance". Exact matches on the query should have a higher weight than fuzzy matches. (2) *Predicted keywords*: Different predicted keywords for the same prefix can have different weights. One way to assign a score to a keyword is based on its inverse document frequency (IDF). (3) *Record weights*: Different records could have different weights. For example, a publication record with many citations could be ranked higher than a less cited publication.

As an example, the following is a scoring function that combines the above factors. Suppose the query Q has keywords $p_1, p_2, \ldots, p_\ell$, while p'_i is the best matching prefix for p_i, and k_i is the best predicted keyword for p'_i. Let $sim(p_i, p'_i)$ be an edit similarity between p'_i and p_i. The score of a record r for Q can be defined as:

$$Score(r, Q) = \sum_i [sim(p_i, p'_i) + \alpha \cdot (|p'_i| - |k_i|) + \beta \cdot score(r, k_i)],$$

where α and β are coefficients ($0 < \beta < \alpha < 1$), $sim(p_i, p'_i) = \frac{\mathsf{ed}(p_i, p'_i)}{|p'|}$, and $score(r, k_i)$ is a score of record r for keyword k_i.

One technical challenge is how to utilize a ranking function in the search process to compute the top answers efficiently. While traditional top-k algorithms (e.g., [6]) could be utilized, new techniques need to be developed specifically for the unique index structure in type-ahead search to support efficient list access and pruning.

3.3 Additional Features

Keyword Highlighting: When displaying records to users, we want to highlight the most similar prefixes for an input prefix. This highlighting feature is straightforward for the exact-match case. For fuzzy search, a query prefix could be similar to several prefixes of the same predicted keyword. Thus, there could be multiple ways to highlight the predicted keyword. For example, suppose a user types in "`lus`", and there is a predicted keyword "`luis`". Both prefixes "`lui`" and "`luis`" are similar to "`lus`". There are several ways to highlight them, such as "<u>**lui**</u>s" or "<u>**luis**</u>", where underlined characters are highlighted. To address this issue, we use the concept of *normalized edit distance*. Formally, given two prefixes p_i and p_j, their normalized edit distance is:

$$\mathsf{ned}(p_i, p_j) = \frac{\mathsf{ed}(p_i, p_j)}{max(|p_i|, |p_j|)}, \tag{3}$$

where $|p_i|$ denotes the length of p_i. Given an input prefix and one of its predicted keywords, the prefix of the predicted keyword with the minimum ned to the query is highlighted. We call such a prefix a *best matched prefix*, and call the corresponding normalized edit distance the "minimal normalized edit distance," denoted as "mned." This prefix is considered to be most similar to the input keyword. For example, for the keyword "`lus`" and its predicted word "`luis`," we have ned("`lus`", "`l`") = $\frac{2}{3}$, ned("`lus`", "`lu`") = $\frac{1}{3}$, ned("`lus`", "`lui`") = $\frac{1}{3}$, and ned("`lus`", "`luis`") = $\frac{1}{4}$. Since mned("`lus`", "`luis`") = ned("`lus`", "`luis`"), "<u>**luis**</u>" will be highlighted.

Supporting Synonyms: We can also utilize a-priori knowledge about synonyms to find relevant records. For example, "`William` = `Bill`" is a common synonym in the domain of person names. Suppose in the underlying data, there is a person called "`Bill Gates`". If a user types in "`William Gates`", we want to find this person. One way to support this feature is the following. On the trie, the node corresponding to "`Bill`" has a link to the node corresponding to "`William`", and vice versa. When a user types in "`Bill`", in addition to retrieving the records for "`Bill`", we also identify those of "`William`" following the link.

3.4 Supporting Type-Ahead Search on Relational Databases

In [10] we studied how to support type-ahead search on a relational database with multiple tables. We model the underlying data as a graph, and propose efficient index structures and algorithms for finding relevant answers on-the-fly by joining tuples in the database. This join operation is more challenging than the single-table case due to the complexity of the data model. We devised a partition-based method to improve query performance by grouping relevant tuples and pruning irrelevant tuples efficiently. We also developed a technique to answer a query efficiently by predicting highly relevant complete queries for the user. The idea is to first predict the query the user may want to type, then use it to compute answers.

3.5 Open Problems

Reducing Memory Requirements: There have been studies on disk-based index structures and algorithms for type-ahead search [1–4]. To achieve a very high interactive speed, especially for Web servers with a lot of queries, ideally we want to store in memory the data, index structures, and cached query results. This memory requirement will increase as the data set increases. Furthermore, it is expensive to answer queries with short prefixes, since these prefixes can have many complete keywords. To make such queries more efficient, we can also cache the results for short-keyword queries, which also require a lot of memory. Thus new techniques are needed to solve this problem of high memory requirement.

One way to solve the problem is utilize solid-state drives (SSDs or flash drives). SSDs serve as a storage layer between traditional hard disks and memory. They are much faster but more expensive than traditional hard disks, yet cheaper and slower than memory. Compared to hard disks, an SSD has several advantages such as smaller startup time, extremely low read latency times, and relatively deterministic read performance. Due to these advantages, there have been many recent studies on how to use this new storage medium to do efficient data management. It is worth studying how to use SSDs to support efficient type-ahead search.

Type-ahead Search in non-English Languages: Considering the global reach of the Internet and its internationalization trend, it is important to study how to extend type-ahead-search techniques mainly designed for English to other languages, and solve unique, common challenges specific to these languages. For instance, there are many languages using the Latin alphabet such as French, Greek, German, and Turkish. They are similar to English in the way words are separated by delimiters such as space. Therefore, existing techniques for search on queries with multiple keywords can be naturally applied to these languages. At the same time, we need to consider their language-specific characteristics. For example, these languages often have special characters that do not exist in English, and these letters often have corresponding English letters. Many special letters are obtained by adding a diacritical mark to an English letter, such as the letter `ä` in German corresponding to the English letter `a`. In most cases these special characters are used to change the accent of their corresponding base letters. On an English keyboard, users often substitute these special characters with their base letters.

We use Turkish as an example to show language-specific characteristics and the corresponding search-related issues. Turkish has six special characters: `ç`, `ğ`, `ı`, `ö`, `ş`, and `ü`, which correspond to characters `c`, `g`, `i`, `o`, `s`, and `u`, respectively. When typing in a special letter, users often type in the base letter since it is easier. This common input behavior affects the way a search should be done. In a search engine such as Google, if a user types in a keyword `koruk`, the search engine will return both `koruk` pages and `körük` pages, since the engine may not know if the user typed the word exactly or substituted special characters with base letters. On the other hand, if the user types in the word `körük` with special characters, the system may still find documents containing the word `koruk`, but possibly give them a lower rank compared to those documents with the word `körük`. The reason is that the user may specifically want to use the special letters to find documents with this word. As a result, the similarity between these two words becomes asymmetric. Similar issues need to considered in type-ahead search in these languages.

Go Beyond String Data: Consider the case where we have a publication database, and a user types in a keyword "`2001`" in type-ahead search. Based on edit distance, we may find a record with a keyword "`2009`". On the other hand, if we knew this keyword is about the year of a publication, then we may relax the condition in a different way, e.g., by finding publications in a year between 1999 and 2003. This kind of relaxation and matching depends on the type and semantics of the corresponding attribute, and requires new techniques to do indexing and searching.

Supporting Large Scale Data: For large amounts of data that cannot be handled by a single machine, we need to use multiple machines to support type-ahead search. For structured data, such as relational records, we can partition them to different machines, and let each machine process its local data. A keyword query is sent to these machines, which answer the query locally and send the results to a master machine to do the aggregation. New techniques are needed to support type-ahead search, especially related to how to partition complex data models such as graphs.

4 Conclusion

In this paper, we gave an overview of the new information-access paradigm, called search-as-you-type or type-ahead search, which finds answers to queries as a user types in keywords character by character. We discussed technical challenges related to achieving a high interactive search speed, reported recently developed results, and presented open problems that need more research investigation.

5 Acknowledgment

The work was partially supported by the National Science Foundation of the US under the awards IIS-0742960 and IIS-0844574, gift funds from Google and Microsoft, the National Science Foundation of China under the grant numbers 60828004 and 60873065, the National High Technology Development 863 Program of China under grant numbers 2007AA01Z152 and 2009AA011906, and the National Grand Fundamental Research 973 Program of China under the grant number 2006CB303103.

References

[1] Holger Bast, Alexandru Chitea, Fabian M. Suchanek and Ingmar Weber. ESTER: efficient search on text, entities, and relations. SIGIR, pages 671-678, 2007.

[2] Holger Bast Christian Worm Mortensen and Ingmar Weber. Output-Sensitive Autocompletion Search. SPIRE, 2006.

[3] Holger Bast and Ingmar Weber. Type less, find more: fast autocompletion search with a succinct index. SIGIR, pages 364-371, 2006.

[4] Holger Bast and Ingmar Weber. The CompleteSearch Engine: Interactive, Efficient, and Towards IR& DB Integration. CIDR, pages 88-95, 2007.

[5] Surajit Chaudhuri and Raghav Kaushik. Extending Autocompletion to Tolerate Errors. SIGMOD, pages 707-718, 2009.

[6] Ronald Fagin. Combining Fuzzy Information from Multiple Systems. PODS, pages 216-226, 1996.

[7] Shengyue Ji, Guoliang Li, Chen Li and Jianhua Feng. Interative Fuzzy Keyword Search. WWW, pages 142-153, 2009.

[8] Chen Li, Jiaheng Lu and Yiming Lu. Efficient Merging and Filtering Algorithms for Approximate String Searches. ICDE, pages 257-266, 2008.

[9] Guoliang Li, Jianhua Feng and Lizhu Zhou. Interactive Search in XML Data. WWW, pages 1063-1064, 2009.

[10] Guoliang Li, Shengyue Ji, Chen Li and Jianhua Feng. Efficient Type-Ahead Search on Relational Data: a TASTIER Approach. SIGMOD, pages 695-706, 2009.

[11] Jiannan Wang, Guoliang Li and Jianhua Feng. Automatic URL completion and prediction using fuzzy type-ahead search. SIGIR, pages 634-635, 2009.

Query Results Ready, Now What?

Ziyang Liu Yi Chen
Arizona State University
{ziyang.liu, yi}@asu.edu

Abstract

A major hardness of processing searches issued in the form of keywords on structured data is the ambiguity problem. A set of keywords itself is not a complete piece of information and may imply different information needs from different users. Although state-of-the-art keyword search engines are usually able to automatically identify meaningful connections among keywords in the data, users may still be frequently overwhelmed by the huge number of results and face much trouble in selecting the relevant ones. Many search engines attempt to rank the results in order of their inferred relevance, however, it is virtually impossible for a ranking scheme to be perfect and works properly for all users and all queries.

In this paper we discuss several post-processing methods for keyword searches on structured data, which ease the users' task of finding and digesting relevant results from all results returned for a keyword query. The methods to be discussed include generating result snippets, differentiating selected results as well as query expansion. Each of these methods helps users gracefully get the relevant information in its own way. At last we discuss the remaining challenges in post-processing keyword searches on structured data.

1 Introduction

Keyword searches are usually ambiguous. A set of keywords with no further information of how they should be related/connected in the results, as is the typical scenario in most keyword searches, can have many different ways to be interpreted. It may very well be the case that different users with different intensions issue the same set of keywords. As a result, it is impossible for a search engine which simply provides a set of ranked results to guarantee that the order of the results exactly matches the user's need. This may cause the users to spend lots of time navigating through the results in order to find the relevant ones. Moreover, it is unlikely that the user always comes up with a good set of keywords; often times the user may miss important keywords or use some "bad" keywords which may harm the quality of the retrieved results, and it will be desirable if the search engine can provide help to the user if so happen.

In this paper we discuss several post-processing techniques that work beyond generating a ranked list of results for keyword searches on structured data. Each of these techniques helps the user get insights of the results in its own way. Let us look at a few desirable ways of postprocessing.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

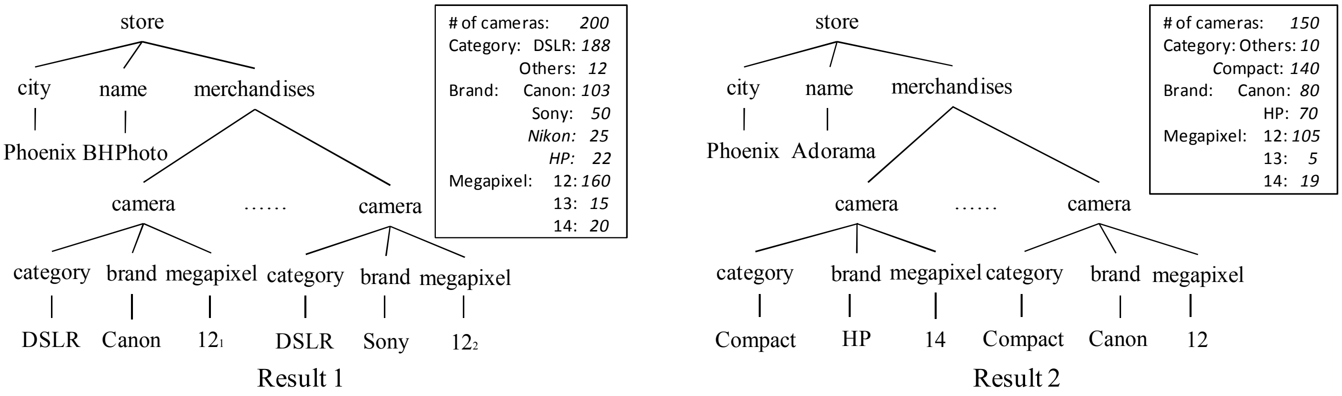

Figure 1: Two Results of Query "*Phoenix, camera, store*"

Result snippet generation is one of the most helpful postprocessing methods for keyword search and is used by almost all web search engines. Although search engines attempt to rank the results in terms of their relevance to the query, all existing ranking schemes are heuristics; the user usually has no idea based on what criteria the results are ranked and cannot possibly fully trust the ranking. Indeed, when we issue a web search, very often the top results consist of a mixture of relevant and irrelevant results. Result snippets are very helpful in these circumstances as they help the users quickly judge the relevance of each result.

After the user roughly understands the content of each result, the user may be interested in the relationship of a set of results by comparing them. Such needs are common in applications like online shopping, employee hiring, job/institution hunting, etc. It is shown in [2] that 50% of keyword searches on the web are for information exploration purposes, and inherently have multiple relevant results. Such queries are classified as informational queries, where a user would like to investigate, evaluate, compare, and synthesize multiple relevant results for information discovery and decision making, rather than reaching a particular website. Generating a comparison table which highlights the key differences of the selected results will be very helpful.

Furthermore, although result snippets and differentiation tables are helpful for users to judge the relevance of results, yet if the keywords selected by the user is imperfect to begin with, the query may retrieve so many irrelevant results that it is still very difficult for the user to pick the relevant ones even with the help of snippets and differentiation techniques. By suggesting a set of related queries to the keyword query the user has issued (also called query expansion), the search engine can help the user get more insights of what is the important information contained in the results, and narrow down the search scope to get more precise results.

Sometimes the keywords have multiple relationships in the data, and in this case clustering the search results and showing one representative for each cluster makes it easy for the user to quickly browse all types of results. Last but not the least, the search engine can also work interactively with the user to improve the result quality through relevance feedbacks.

The rest of the paper is organized as follows. Section 2 discusses result snippet generation techniques proposed in eXtract [4]; Section 3 introduces the approach for generating comparison tables for selected results proposed in XRed [7]; in Section 4 we introduce two related approaches, Data Clouds [5] and [10], which selects important keywords from the search results for query expansion. At last we discuss the remaining challenges of processing keyword searches on structured data and potential future research directions.

2 Generating Result Snippets

Snippets help the user quickly understand the essence of a result without reading through its entirety. As an example, let us look at fragments of two tree-structured search results of query "*Phoenix, camera, store*" shown

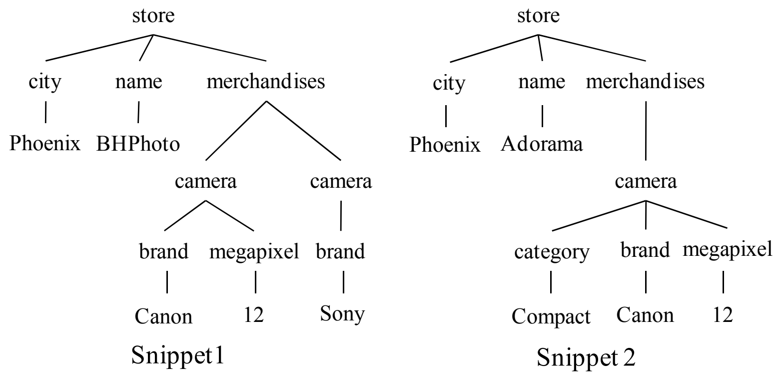

Figure 2: Snippets of the Result in Figure 1

in Figure 1. Some nodes in the results are omitted due to space constraint, and the box next to each result records the statistics information of the result, e.g., *DSLR: 188* indicates there are 188 DSLR cameras in this result.

Two sample snippets of the results are shown in Figure 2. As can be seen, each snippet captures the heart of the result in a small tree, e.g., from snippet 1, we know that the result is about a store named BHPhoto in Phoenix, which features Canon and Sony cameras and mainly sells cameras with 12 megapixel. Users can judge the relevance of the query result from this brief subtree of carefully selected nodes.

Snippet generation on structured data faces several unique challenges comparing with snippet generation on plain text documents. Most importantly, unlike results on text documents, results on structured data typically have a tree structure, which is not composed of sentences or paragraphs. Therefore, selecting representative sentences as snippets is not a valid solution. Instead, the selection should be made at the node level. Except nodes that match keywords, it is not directly obvious which nodes are significant and helpful for the user to judge the result relevance. In this subsection we introduce a system eXtract [4], which adopts techniques for snippet generation in XML keyword search.

eXtract attempts to achieve four goals in generating a snippet: (1) the snippet should be self-contained (i.e., represent a semantic unit); (2) the snippet should distinguish the result from other results; (3) the snippet should be a reasonable summary of the result; (4) the snippet should be small to be easily comprehensible. Note that the first three goals interfere with the last one: the more nodes a snippet has, the better it achieves the first three goals but the harder it is for the user to comprehend it. eXtract takes all four goals into account by creating a list of important *features* in the result, called IList. eXtract then adds the features in the IList into the snippet one by one, until the snippet size has reached a limit. Each feature is a triplet consisting of an entity, an attribute and a value, e.g., (*camera, category, DSLR*). Entities and attributes are inferred using the heuristics proposed in [6].

The IList is initialized to include the keywords in the query. To make the snippet self-contained, eXtract adds all entities in the result into IList. Take result 1 in Figure 1 as an example. The entities in the result are store and camera.

To make the snippet distinguishable, eXtract identifies the key of a result and adds it to the IList. The key of a result is considered to be the key attribute of an entity (called return entity) selected by eXtract. In result 1 in Figure 1, the return entity is *store*. The key attribute can be retrieved from the DTD or schema (if available), otherwise the most selective attribute can be used. eXtract adds the key attribute value of the return entity to the IList.

To make the snippet a reasonable summary of the result, eXtract identifies features whose occurrences are dominant. A feature is dominant if its number of occurrences in the result is large compared with features with the same type, where feature type refers to the (entity, attribute) pair. For example, according to the statistics of result 1 in Figure 1, *DSLR*, *Canon*, *Sony* and *12* are dominant features. A quantitative measure is adopted to assess the dominance of a feature, i.e., the dominance score of feature f wrt result r, defined as following:

Phoenix, camera, store, BHPhoto, Canon, 12, Sony, DSLR, …

← *keywords* → ← *key* → ← *features* →

← *entities* →

Figure 3: IList for Result 1 in Figure 1

Feature Type	DFS of Result 1 (D_1)	DFS of Result 2 (D_2)
store:name	BHPhoto	Adorama
camera:brand	Canon Canon Sony	Canon HP
camera:category	DSLR	Compact

Figure 4: Comparison Table of the Result in Figure 1

$$DS(f, r) = \frac{N(f)}{\frac{N(f.t)}{D(f.t)}}$$

where $N(\cdot)$ is the number of occurrences of a feature or feature type in the result, $f.t$ is the feature type of f, $D(\cdot)$ is the domain size (i.e., number of distinct features) of a feature type. For example, suppose there are 3 camera categories (DSLR, compact, single-use), then the dominance score of feature *DSLR* is 188/(200/3) = 2.82. eXtract adds the features in the result into the IList in the order of their dominance score. The final IList of result 1 in Figure 1 looks like the one in Figure 3.

eXtract then includes the items in the IList one by one into the snippet until it reaches a preset size limit in terms of number of edges. The challenge hereby is to choose the instance of each item so that the snippet can accommodate as many items as possible. For example, to include items *canon* and *12*, using 12_1 in Figure 1 is better than using 12_2 as the former choice results in a smaller tree size, thus leaving more space for other items. The problem of including the most number of items in the snippet is proved to be NP-hard, and eXtract uses a greedy algorithm for snippet generation. The details of the algorithm are omitted in this paper and can be learned from [4].

3 Query Result Differentiation

From reading the snippets, the users will understand the most important features in the results. However, many snippets may look similar to each other, and in order for the user to further judge the relevance of results, he/she needs to learn the key differences of these results by comparing and analyzing them. From Figure 1, the store in result 1 mainly sells Canon and Sony DSLR cameras, with roughly twice as many Canon cameras as Sony cameras; while the store in result 2 mainly sells Canon and HP Compact cameras. From their snippets, we know result 2 focuses on Compact cameras, but have no idea whether or not result 1 focuses on Compact or DSLR, since the category information about the store is missing in its snippet. Similarly, result 1 has many Sony cameras, but we do not have information about whether result 2 has many Sony cameras or not. As we can see, snippets are designed to summarize a single result, rather than compare and differentiate multiple results. Figure 4 shows the comparison table of the two results, which highlights the key difference of the two results, e.g., the first store focuses on DSLR and the second one focuses on compact camera, which is helpful for the user to make his/her decision.

In this section we introduce XRed [7], a structured search result differentiation system that generates a

Differentiation Feature Set (DFS) for each result to help users compare and contrast results. Similar as eXtract, XRed considers each (entity, attribute, value) triplet as a feature in the result. XRed allows users to select a set of results for comparison, then it automatically selects a set of features from each result to generate a DFS that shows the differences of the result from others.

XRed generates DFS according to the following 3 desiderata:

(1) *Being Small.* Similar as a result snippet, to enable users quickly differentiate query results, a DFS should be small, so that users can quickly browse and understand them. XRed allows the user to specify a number which is used as the upper bound of the number of features in each DFS.

(2) *Summarizing Query Results.* For the comparisons based on DFSs to be valid, a DFS should be a reasonable summary of the corresponding result by capturing the main characteristics in the result. Otherwise, the differences shown in two DFSs do not reflect the actual differences between the corresponding query results.

Specifically, a feature that has more occurrences in the result should have a higher priority to be selected in the DFS, so that the DFS reflects the most important feature in the result, and the differences among DFSs correctly reflect the main differences of their corresponding results. Consider again the two results of query "*Phoenix, camera, store*" in Figure 1(a). Both results mainly sell *Canon* cameras. The store in result 1 also sells a couple of *HP* cameras. Suppose we have the DFS for result 1, D_1={store:brand:HP}, and the DFS for result 2, D_2={store:brand:Canon}. Obviously these two DFSs are different. However, the difference is meaningless as they give users the impression that store 1 differs from store 2 by mainly selling HP cameras instead of Canon cameras, which is untrue.

Furthermore, the distributions of features of the same type in a DFS should roughly reflect their distribution in the data (up to the size limit). For example, although both stores in these results sells *Canon* and *HP*, it is undesirable to have a single occurrence of *Canon* and *HP* in the DFS of each result. Such DFSs give users the impression that the two stores are similar in terms of their speciality on *Canon* and *HP*. In fact, the store in result 1 mainly focuses on *Canon* with just a couple of *HP*; whereas the store in result 2 focuses on both *Canon* and *HP*, with roughly the same number of cameras.

(3) *Differentiating Query Results.* Obviously, a DFS should be able to differentiate the result it represents from others. Since different results are compared on *feature types*, a differentiation unit is a feature type, e.g., (*camera, category*). A feature type t can differentiate two DFSs if the features or the order of features of type t are different in the two DFSs.

As an example, consider feature type (*camera, category*) in the DFSs in Figure 4. This feature type can differentiate the two DFSs, as feature *DSLR* is the most common feature of this type in DFS 1, while in DFS 2 the most common one is *compact*. For feature type (*camera, brand*), it can also differentiate the two DFSs as the second most common feature in the two DFSs are *Sony* and *HP*, respectively. Assuming that we use feature *HP* to replace *Sony* in DFS 1, then they are still differentiable, as the ratios between the number of occurrences of *Canon* and that of *HP* are about 2:1 in DFS 1 and 1:1 in DFS 2, respectively.

Using the concept of differentiable feature type, XRed judges the quality of two DFSs by the number of feature types that can differentiate, which is called their *degree of differentiation* (DoD). The DoD of multiple DFSs is the total DoD of all pairs of DFSs.

The DFS generation problem is formulated as the problem of generating a set of DFSs, one for each result, such that each DFS should satisfy the size limit, be valid and maximize their DoD. The problem is proved to be NP-hard. XRed adopts greedy algorithms that achieve two local optimality criteria efficiently. The detailed algorithms and proofs can be found in [7].

4 Query Expansion

Sometimes since the keywords issued by the user are imperfect, a large number of irrelevant results may be retrieved. Even with the help of snippets and differentiation, it is still a tough task for the user to quickly locate the relevant ones. In this case, the search engine can help user narrow down the search scope and improve the result quality by suggesting a set of keywords related to the original query, which the user can use to revise the query. For example, the sample query "*Phoenix, camera, store*" may retrieve a large number of results as the predicates are not very specific. The user issues this query possibly because he/she is just interested in buying a digital camera, but is unfamiliar with detailed specifications of cameras. If the search engine is able to automatically recommend to the user a set of keywords such as "DSLR", "compact", "12 megapixel", etc., the user will be able to pick the desired keywords and reduce the number of retrieved results.

Major search engines provide query expansion functionality in the form of query auto-completion. Most of them are implemented based on user query logs [1]. For instance, [3] extracts the personal information from users' desktops and query logs, and provides adaptive factors to evaluate the relevance of words. Although these techniques are designed for web search engines, they can be applied to keyword search on structured data as well. We do not further elaborate these approaches as they are not based on post-processing the query results.

In this section we mainly focus on a paper that studies query expansion for keyword search on structured data without the aid of query logs: Data Clouds [5]. Data Clouds allows user to issue keyword queries on relational databases, and ranks the terms in all results or selected top results, then returns the top ranked terms, which can be used to form expanded queries.

To generate query results, Data Clouds first identifies a set of entities (e.g., product) in the relational database which is done by domain experts or database administrators. Then it asks the user to specify which entity to search. Each result is an entity that contains all keywords in the tuples related to it.

To find interesting terms as suggest keywords, Data Clouds discusses three alternative ranking schemes for ranking terms in the results: popularity based ranking scheme, relevance based ranking scheme, and query-dependent ranking scheme. Data Clouds returns terms with high scores for query expansion. The ranking schemes are illustrated below.

(1) *Popularity Based Ranking Scheme.* This ranking schemes is based on the observation that the keywords that appears many times in a result are likely important.. Given query q, a result r and a term $t \in r$, the popularity score of term t is:

$$score(t, q, r) = \sum_{e \in r} \sum_{a \text{ of } e} n_a$$

where e is an entity, a is an attribute of e and n_a is the number of times t occurs in a.

(2) *Relevance Based Ranking Scheme.* Simply considering the numbers of occurrences of terms may lead to selecting terms that occur universally frequently, which is undesirable as those terms are not representative for the results. The relevance based ranking scheme additionally incorporates the idea of inverse document frequency into the ranking score. Given query q, a result r and a term $t \in r$, the relevance score of term t is:

$$score(t, q, r) = \sum_{e \in r} tf_{t,e} \times idf_t$$

where $tf_{t,e}$ is the frequency of term t within entity e, and idf_t is the log of the ratio of the number of entities in the data over the number of entities containing t. Note that relevance based ranking scheme uses term frequency $tf_{t,e}$ instead of number of occurrences in the popularity based ranking scheme, as term frequency is normalized over the size of the entities and is more fair.

(3) *Query Dependent Ranking Scheme.* Compared with the relevance based ranking scheme, this ranking scheme additionally favors keywords occurring in entities that are highly related to the query. By combining TFIDF with the scores of entities, the query dependent score of a term t is:

$$score(t, q, r) = \sum_{e \in r} (tf_{t,e} \times idf_t) \times score(e, q)$$

where $score(e, q)$ is the relevance of entity e to query q, computed using traditional TFIDF metrics.

[8] also proposes a set of ranking factors for finding important keywords related to a user query. It is designed specifically for queries related to online shopping: the keywords in the expanded queries are features of the products. [8] uses three factors to judge the importance of a candidate query. The first factor is based on the co-occurrence of the words in the candidate query with the keywords in the original query; high co-occurrence is preferred. The other two factors rely on the users' reviews of a product. They favor the attributes of a product with extreme ratings (i.e., highly positive/negative) and consistent ratings (i.e., receiving the same rating from many users), respectively. Top-k queries with the highest scores are computed and returned to the user. Another difference from Data Clouds is that [8] considers a ranked list of queries, rather than keywords. For example, for a user query "Canon", Data Clouds returns suggested keywords such as "lens", "DSLR", "zoom", etc., while [8] returns suggested queries such as "camera, lens, zoom".

Different from the two approaches introduced above, [10] ranks the terms in the order of their number of occurrences in the results. It requires the input data to be a relational database with a schema. [10] proposes an efficient algorithm that is able to compute precisely the top-k frequently occurred terms without even processing the query. This enables the query processing and query expansion to be parallelized and shorten the response time of a search engine. The detailed algorithm is omitted due to space constraint and we direct readers to the paper for more insights.

5 Conclusions and Future Work

In this paper we discuss the benefits of keyword query post-processing on structured data, and introduce several post-processing techniques which, from different angles, help users retrieve the relevant results: result snippets, result differentiation and query expansion. Result snippets help user quickly get the essence of the results without the need to read the entire results; result differentiation highlights the key differences of the selected results which help the user compare and prioritize them; query expansion suggests a set of keywords based on the results which gives the user insights of the results and helps the user narrow down the search scope.

There are other desirable post-processing methods which are open problems, for instance, query result clustering and utilizing user feedbacks.

(1) *Result Clustering.* Results of a keyword search can have many subcategories. For example, for query "Phoenix, camera, store", some stores sell mainly DSLR cameras, some sell compact cameras, and some focus on selling used cameras. Moreover, keyword "Phoenix" may have different semantics in different results: it may match a city in Arizona, appear in a review, or even appear in the description of a camera. Note that ranking schemes tend to give similar ranking scores to similar results, thus results in the same category will likely get similar ranks. If the user only browse the top results (which is the case most of the time), he/she may only see one category and miss many potentially relevant results. A natural way of solving this issue is to cluster the query results. By grouping similar results together and showing either one result per cluster or a natural language description for each cluster, the user will be able to conveniently view the result categories and navigate into the relevant ones.

Much research has been done for clustering structured data; however, they are not query-aware, thus their clustering criteria does not necessarily correspond to different interpretations of the query semantics. Furthermore, existing clustering criteria cannot be easily understood by the user, thus by looking at one result in a cluster, it can be very difficult for the user to tell in what way they are similar. Query result clustering is a useful post-processing method whose techniques await development.

(2) *Utilizing Relevance Feedback.* After the results are generated, the search engine can also interact with the user to improve the result quality. For example, the search engine can ask the user to provide some quality judgement, e.g., specifying which results are relevant or assigning satisfaction scores to some results. Based on the feedback from the user, the search engine can adjust its strategy in composing and ranking the results. The

search engine may also utilize implicit feedbacks, e.g., the results clicked by the user and the time each result is browsed.

Utilization of user feedback is widely studied and adopted in information retrieval, whereas for keyword search on structured data, the problem is not much studied. The structure of the data poses new challenges for utilizing relevance feedback, e.g., the retrieval units are nodes rather than documents, and the ranking schemes for structured data can usually be more complicated than those for text documents, requiring novel techniques for incorporating feedbacks. [9] briefly discusses the problem of learning weights of edges in the schema graph for keyword searches on relational databases. Whenever the user specifies a desirable way to connect the keywords, the edge weights are adjusted, so that the score of any other way of connecting keywords is lower by a margin that is equal to the distance of the two trees. Barring this work, there are still many open problems on utilizing feedbacks. For example, it remains unclear how to adjust the weight of state-of-the-art ranking factors, such as TFIDF, number of keywords appearing in the result, height of the result tree, etc., in face of user feedbacks. User feedbacks can also be used by the search engine to either automatically refine the original query, or suggest a set of new queries that can retrieve user preferred results.

6 Acknowledgement

This material is based on work partially supported by NSF CAREER award IIS-0845647, IIS-0740129 and IIS-0915438.

References

[1] Ziv Bar-Yossef and Maxim Gurevich. Mining Search Engine Query Logs via Suggestion Sampling. VLDB, pages 54–65, 2008.

[2] Andrei Broder. A Taxonomy of Web Search. ACM SIGIR Forum, volume 36, number 2, pages 3-10, 2002.

[3] Paul - Alexandru Chirita, Claudiu S. Firan and Wolfgang Nejdl. Personalized Query Expansion for the Web. SIGIR, 2007.

[4] Yu Huang, Ziyang Liu and Yi Chen. Query Biased Snippet Generation in XML Search. SIGMOD, 2008.

[5] Georgia Koutrika, Zahra Mohammadi Zadeh and Hector Garcia-Molina. Data Clouds: Summarizing Keyword Search Results over Structured Data. EDBT, 2009.

[6] Ziyang Liu and Yi Chen. Identifying Meaningful Return Information for XML Keyword Search. SIGMOD, 2007.

[7] Ziyang Liu, Peng Sun and Yi Chen. Structured Search Result Differentiation. SIGMOD, 2009.

[8] Nikos Sarkas, Nilesh Bansal, Nilesh Bansal, Gautam Das and Nick Koudas. Measure-driven Keyword-Query Expansion. VLDB, 2009.

[9] Partha Pratim Talukdar, Marie Jacob, Muhammad Salman Mehmood, Koby Crammer, Zachary G. Ives, Fernando Pereira and Sudipto Guha. Learning to Create DataIntegrating Queries. VLDB, 2008.

[10] Yufei Tao and Jeffrey Xu Yu. Finding Frequent Co-occurring Terms in Relational Keyword Search. EDBT, 2009.

Evaluating the Effectiveness of Keyword Search

William Webber
Computer Science and Software Engineering
The University of Melbourne
Victoria 3010, Australia
wew@csse.unimelb.edu.au

Abstract

The prevalence of free text search in web search engines has inspired recent interest in keyword search on relational databases. Whereas relational queries formally specify matching tuples, keyword queries are imprecise expressions of the user's information need. The correctness of search results depends on the user's subjective assessment. As a result, the empirical evaluation of a keyword retrieval system's effectiveness is essential. In this paper, we examine the evolving practices and resources for effectiveness evaluation of keyword searches on relational databases. We compare practices with the longer-standing full-text evaluation methodologies in information retrieval. In the light of this comparison, we make some suggestions for the future development of the art in evaluating keyword search effectiveness.

1 Introduction

The rise of search engines as gateways to the Internet has made searching an everyday activity. The predominant mode of search is through the use of *keywords*, a small number of highly discriminating terms that the user anticipates will identify the web pages they are looking for. Keyword search offers a straightforward, intuitive, and flexible method of retrieving information. The success of keyword search on the web has generated interest in keyword search interfaces to relational databases and similar structured data sources. The traditional method of querying structured data stores is through formal query languages such as SQL. Such query languages, however, require much time to learn, and knowledge of a store's data schema to use. Keyword search interfaces offer a simple and flexible alternative, with (it is hoped) minimal loss of querying power.

Keyword search on unstructured text data has long been studied in the information retrieval community, where it goes under the name of free text search. Keyword searches provide only an approximate specification of the information items to be retrieved. Therefore, the correctness of the retrieval cannot be formally verified, as it can with query languages such as SQL. Instead, retrieval effectiveness is measured by user perception and experience. The empirical assessment of keyword-based retrieval systems is therefore imperative. Such assessment is the topic of this paper. We begin in Section 2 with a description of the characteristics of keyword search. Section 3 surveys the history of effectiveness evaluation on unstructured text in information retrieval. In Section 4, we examine the resources and methods used to date for keyword search evaluation over structured data. Finally, we consider in Section 5 some future directions for the evaluation of keyword-based retrieval on relational databases.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 Characteristics of Keyword Search

Users perform searches to satisfy *information needs*. A keyword query is an expression of such an information need, and it is the task of the retrieval system to return information items that are *relevant* to that need. For unstructured text, the information items are discrete documents. For relational data, however, the information items are (possibly joined) tuples. The relational search system therefore has the additional responsibility of determining the candidate tuple joins. Additionally, the keyword query contains no schema information, so that each keyword potentially must be matched against each field of the joined tuple [1, 7].

A keyword query does not precisely define which information items are relevant to the user's information need. For instance, items may be relevant even though they do not contain all query terms, and conversely items containing all query terms may not be relevant. Instead, retrieval involves computing the *similarity* between the user's query and each information item. Similarity metrics generally take into account textual features such as the number of times a keyword occurs in an item, and the overall discrimination of the keyword in the collection [8]. With relational data, structural features can also be incorporated in the similarity metric. For instance, tuples that have fewer joins may be preferred as more coherent than tuples with many joins [1, 9]; or query keywords could be matched with schema terms [10]. The metric assigns a similarity score to each candidate item. The system then ranks the items by decreasing similarity, and returns some prefix of the ranking to the user. The goal of ranking by similarity is to bring more relevant items to the top, thus minimizing the amount of effort the user must expend to find the information they want.

In a structured query language like SQL, there is only one correct answer set. In contrast, there are many plausible similarity metrics, each with its own way of inferring a user's information need from a query, and of calculating the query's similarity to information items, to generate a ranking of answers. The *effectiveness* of a response to a keyword query, and hence of the similarity metric, is not something that can be formally proved; rather, it is determined by the user who realized the information need, formulated the query, and perused the response. This effectiveness must be empirically assessed.

3 Evaluation in Information Retrieval

The empirical approach to information retrieval began with the field itself, in the experiments conducted at the library of the Cranfield Aeronautical College, England, in the late 1950s and early 1960s, under the direction of the librarian, Cyril Cleverdon [4, 17]. The orthodoxy of the time in information science was that complex, hierarchical indexing schemes were essential to effective retrieval. The question the Cranfield experiments set out to answer was, which indexing scheme was best; and the answer the experiments arrived at was: none. It made no great difference which scheme was used; simply indexing documents by plain keywords was as good a method as any; what mattered was the process of retrieval. Cleverdon himself described these as "results which seem to offend against every canon on which we were trained as librarians" [4, p.252]. These findings turned attention away from information classification, and towards information retrieval.

The experimental method developed in the Cranfield tests has been highly influential. The first component of this method is the *corpus* of documents to index and retrieve. Against this corpus, user information requests must be processed; and a key insight of Cranfield was to divide these requests into two further components: first, the request statements themselves (what later became termed the *topics*); and second, assessments of which documents in the corpus were relevant to each request (called *qrels* in the jargon). These three components – corpus, topics, and qrels – together form a test collection; and the use of such a test collection in evaluation is often termed the "Cranfield methodology" or even the "Cranfield paradigm" [19].

The Cranfield experiments themselves were carried out entirely, and rather heroically, by manual means; the two hours required to process each of the 361 searches by hand was regarded as "relatively cheap compared to what would have been the cost for any form of machine searches" [4, p.91]. The test collection method is,

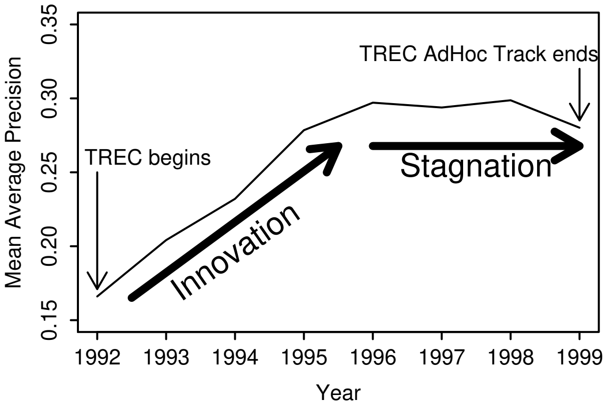

Figure 1: Retrieval effectiveness of the SMART versions from the first eight years of TREC, averaged across the first eight TREC collections [3].

though, ideally suited to automation and computerization. Relevance assessments are made in advance and are reusable, so experiments can be performed automatically and cheaply. The first such computerized retrieval systems were developed in the early 1960s, the most famous being that of the SMART project at Cornell University [13, 14]. Early progress was brisk, driven by the fast turnaround of collection-based evaluation, and the foundations of statistical information retrieval were laid down within a decade, including term weighting, query expansion, and result ranking. Over time, however, the field suffered from a lack of consolidation of results, due in part to the small and ageing test collections employed [16]. The largest collection used by SMART in 1990 had under 13,000 documents and was already 20 years old [15]. The credibility of experimental findings was undermined, impeding the adoption of research technologies in operational systems [13].

The second great impetus to empirical IR research came with the institution of the Text REtrieval Conferences (TREC) in 1992 [6, 20]. TREC produces large-scale, up-to-date test collections, encourages collaborative experiments upon them, and provides a venue for publishing and discussing results. The first collection used at TREC, known as TIPSTER, contained some 750,000 documents, a fifty-fold leap over what was available previously. The collaborative experiments run at TREC also provided a forum for the direct comparison, on equal terms, of many different research ideas. The first TREC experiment involved 22 research groups; by the fifth year, this has reached 38; and at its (apparent) peak in 2005, 117 research groups participated in TREC [18].

The impact that the TREC effort had upon the effectiveness of retrieval systems can be gauged from Figure 1. Even though SMART had been under active development for three decades, the first five years of TREC saw its retrieval effectiveness almost double, as measured by mean average precision, one common evaluation metric. And SMART's experience is typical of that of other participating groups. A number of important innovations were made during these early years of TREC, from new similarity metrics such as BM25 [12], now the standard retrieval formula, to smaller refinements that nevertheless had significant impacts, such as document length normalization. Few of these innovations were revolutionary in their nature; rather, the existence of a large and standard test environment allowed existing ideas to be extended, refined, and tuned. Figure 1 also suggests that improvements had plateaued by TREC's fifth year, leading to the retirement of the ad-hoc (plain text) task in favour of fresher tasks in 1998; and a recent survey has found little improvements in ad-hoc retrieval effectiveness during the following decade [2]. This underlines the impact that the standard, large-scale test collections produced by TREC had: in just a few years, they took retrieval technology from half of its potential, arrived at through three decades of piecemeal research, up to the effectiveness limits of current approaches.

4 Evaluation of Keyword Search

The earliest work in keyword search on relational databases was concerned with the practicality of performing it in a reasonably efficient manner. At this stage, any and only tuples that contained all of the query keywords were considered correct matches for the query. Results were ranked simply by the increasing number of joins (on the principle that the fewer joins, the more coherent the answer), and the evaluation of effectiveness was not considered [1,7]. As technology developed, researchers became interested in not just the tractability of keyword retrieval, but the quality of the results. Proper metrics of similarity between query and answer tuple were introduced. Some of these metrics were specific to structured data, and were concerned with the conciseness or coherence of the retrieved answer [5]. Other similarity metrics were adopted from full-text information retrieval, treating either whole answer tuples or their individual fields as virtual documents [8, 10]. The most fruitful of the similarity metrics combined both a structural and a full-text component [9].

With the development of proper similarity metrics to process queries came the need to assess the effectiveness of the results. A number of different test datasets have been employed by different researchers. Two in particular have become widely used: the IMDB movie database, and the DBLP database of academic publications and citations. Such agreement is not to be found on query sets, though. Queries are generally formulated by the authors themselves, rather than taken from a query log, say, or written by independent third parties. Self-authored queries have a strong potential for bias: it is too easy to formulate queries that are favourable to your own over other algorithms. Query sets have not been re-used between experiments and experimenters, making comparison of results difficult, unless the researcher provides that comparison directly through re-implementing existing approaches as baselines – an important practice which, fortunately, is fairly common in the published keyword work. Query sets are frequently quite small, rarely more than 20 per dataset, and sometimes as few as 5. This is somewhat short of the 50 queries used in TREC collections, and even that figure of 50 is regarded by many as insufficient [21]. An exception is [10]: explicitly following TREC practice, they use a set of 50 queries; additionally, these queries are not created by the authors, but sampled from the log of a commercial search engine.

Just as queries are generally authored by the researchers, so too relevance assessment is generally performed by them or their colleagues. Like self-authoring, self-assessment has the potential for biasing results, with the same interpretation of relevance determining both algorithm design and relevance assessment – and query construction too, if self-authored. What should be a (correctly) subjective assessment of relevance can easily verge on an (incorrectly) objective verification of correctness. The suspicion that this has occurred is strongest where abnormally high effectiveness scores are achieved. For instance, the best fully automatic TREC participant systems achieve scores under the precision at depth 100 metric of around metric of around 0.25; but [9] report around 0.9. Similarly, top-end scores at TREC for mean reciprocal rank (MRR, an admittedly unstable metric) are around 0.8; but [11] achieve the rather astonishing, perfect MRR score of 1.

Given the disparity of query sets, corpora, and assessment methods, it is not straightforward to determine how retrieval effectiveness in keyword search has progressed. The only method is to follow the chain of baselines. So, [10] employ a variety of tuning mechanisms to improve the vanilla IR baseline of [8] from a MRR of 0.245 to 0.871. In turn, [11] give MRR scores of 0.243 and 0.333 respectively for baselines derived from the two aforementioned papers, and then report their impressive perfect score of 1. [22] count instead the proportion of top-ranked documents that are relevant (P@1); they report a mean P@1 score for their [8] baseline of 0, and for [11] baseline of 0.2; their own systems achieves a mean P@1 of 0.95. It is rather difficult to know what to make of this sequence of results, with each researcher tripling their predecessor's baseline to achieving perfect or near-perfect scores. There is persuasive *prima facie* evidence that substantial improvements in effectiveness are being made from a reasonable, standard IR beginning; [10] tantalizing report better effectiveness than Google (and their effectiveness has been tripled twice since!). But the reliable verification of such improvements would require a larger, more thorough, and more impartial, experimental environment.

5 Future Directions In Keyword Search Evaluation

The field of keyword search on structured data is well poised for growth towards maturity. The fundamental technical and formal problems of performing such search have been solved, and many important theoretical results have been achieved (in, for instance, graph theory). Concern is now turning to questions of the end-user effectiveness of such search systems. Traditional IR similarity metrics have been ported to the new domain, and combined with domain-specific structural features. There is also evidence of significant improvements in effectiveness, both through developing new methods and tuning existing ones.

Keyword search on structured data is therefore at roughly the same stage that information retrieval was pre-TREC; or, to be less sanguine, the stage that information retrieval had reached by the mid-seventies, and was not to clearly surpass for another two decades. There is much promise in the field, but more needs to be done to set it on a firm basis, to validate its results, and to inspire the confidence needed to convert this research technology into deployed tools. And most of these desiderata depend upon improvements in evaluation method.

What is needed is a standard method, combined with large-scale, independently curated test collections. Some straightforward quantitative points are immediately apparent – for instance, test sets of only 20 queries are scarcely adequate. Test collections also need to be more reusable; not merely document corpora should be made available, but the query sets and relevance judgments to go with them, even if the last of these are incomplete. And rather than unadorned keyword queries, test collections should have properly formulated topics; that is, fuller statements of information need. Such fuller topics form a number of useful functions. They guide additional relevance assessment, if such assessment should prove necessary. Moreover, they allow for different query formulations to be made for the one information need, with each formulation assessable by the same set of relevance judgments. The ability to reformulate queries is particularly important for a new and fluid field, as new retrieval methods may require different query methods to draw out their features.

The field of keyword search may, however, still be too young, and the technology too fluid, for a full TREC-style collaborative experiment to be achievable or even appropriate. Instead, the way forward would seem to be for individual research groups to create more thorough, credibly independent, and re-usable test collections, incorporating all three components – corpus, topics, and qrels. Such an undertaking requires a non-trivial amount of effort. But the experience of TREC demonstrates how much leverage standard collections and a standard methodology can achieve.

References

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: a system for keyword-based search over relational databases. In *Proc. 18th International Conference on Data Engineering*, pages 5–16, San Jose, California, Feb. 2002.

[2] T. Armstrong, A. Moffat, W. Webber, and J. Zobel. Improvements that don't add up: ad-hoc retrieval results since 1998. In *Proc. 18th ACM International Conference on Information and Knowledge Management*, pages 601–610, Hong Kong, China, Nov. 2009.

[3] C. Buckley and J. Walz. SMART in TREC 8. In E. Voorhees and D. Harman, editors, *Proc. 8th Text REtrieval Conference*, pages 577–582, Gaithersburg, Maryland, US, Nov. 1999. NIST Special Publication 500-246.

[4] C. Cleverdon, J. Mills, and E. Keen. *Factors determining the performance of indexing systems*. Aslib Cranfield Research Project, Cranfield, 1966.

[5] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: ranked keyword search over XML documents. In *Proc. 29th ACM SIGMOD International Conference on Management of Data*, pages 16–27, San Diego, California, 2003.

[6] D. Harman. Overview of the first text REtrieval conference (TREC-1). In D. Harman, editor, *Proc. 1st Text REtrieval Conference*, pages 1–30, Gaithersburg, Maryland, US, Nov. 1992. NIST Special Publication 500-207.

[7] V. Hristidis and Y. Papakonstantinou. DISCOVER: keyword search in relational databases. In *Proc. 28th International Conference on Very Large Data Bases*, pages 670–681, Hong Kong, China, Aug. 2002.

[8] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *Proc. 29th International Conference on Very Large Data Bases*, pages 850–861, Berlin, Germany, 2003.

[9] G. Li, B. 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 *Proc. 34th ACM SIGMOD International Conference on Management of Data*, pages 903–914, Vancouver, Canada, June 2008.

[10] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *Proc. 32nd ACM SIGMOD International Conference on Management of Data*, pages 563–574, Chicago, IL, USA, 2006.

[11] Y. Luo, X. Lin, W. Wang, and X. Zhou. SPARK: top-k keyword query in relational databases. In *Proc. 33rd ACM SIGMOD International Conference on Management of Data*, pages 115–126, Beijing, China, 2007.

[12] S. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In D. Harman, editor, *Proc. 3rd Text REtrieval Conference*, pages 109–126, Gaithersburg, Maryland, US, Nov. 1994. NIST Special Publication 500-225.

[13] G. Salton. The Smart environment for retrieval system evaluation–advantages and problem areas. In K. Sparck Jones, editor, *Information Retrieval Experiment*, chapter 15, pages 316–329. Butterworths, 1981.

[14] G. Salton, editor. *Information Storage and Retrieval*, volume Scientific Report No. ISR-11. Cornell University, Ithaca, New York, 1966.

[15] G. Salton and C. Buckley. Improving retrieval performance by relevance feedback. *Journal of the American Society for Information Science*, 44(4):288–297, 1990.

[16] K. Sparck Jones. Retrieval system tests 1958–1978. In K. Sparck Jones, editor, *Information Retrieval Experiment*, chapter 12, pages 213–255. Butterworths, 1981a.

[17] K. Sparck Jones. The Cranfield tests. In K. Sparck Jones, editor, *Information Retrieval Experiment*, chapter 13, pages 256–284. Butterworths, 1981b.

[18] E. Voorhees. Overview of TREC 2007. In E. Voorhees and L. Buckland, editors, *Proc. 16th Text REtrieval Conference*, pages 1:1–16, Gaithersburg, Maryland, US, Nov. 2007. NIST Special Publication 500-274.

[19] E. Voorhees. The philosophy of information retrieval evaluation. In *Proc.2ndWorkshop of the Cross-Lingual Evaluation Forum*, volume 2406 of *Lecture Notes in Computer Science*, pages 355–370, Darmstadt, Germany, Sept. 2002.

[20] E. Voorhees and D. Harman, editors. *TREC: Experiment and Evaluation in Information Retrieval*. The MIT Press, 2005.

[21] W. Webber, A. Moffat, and J. Zobel. Statistical power in retrieval experimentation. In *Proc. 17th ACM International Conference on Information and Knowledge Management*, pages 571–580, Napa, USA, Oct. 2008.

[22] Y. Xu, Y. Ishikawa, and J. Guan. Effective top-k keyword search in relational databases considering query semantics. In *Proc. APWeb-WAIM 2009 International Workshops*, volume 5731/2009 of *LNCS*, pages 172–184, Suzhou, China, Apr. 2009.

RSEARCH: Enhancing Keyword Search in Relational Databases Using Nearly Duplicate Records

Xiaochun Yang Bin Wang Guoren Wang Ge Yu
Key Laboratory of Medical Image Computing (Northeastern University), Ministry of Education
School of Information Science and Engineering, Northeastern University, China
{yangxc,binwang,wanggr,yuge}@mail.neu.edu.cn

Abstract

The importance of supporting keyword searches on relations has been widely recognized. Different from the existing keyword search techniques on relations, this paper focuses on nearly duplicate records in relational databases due to abbreviation and typos. As a result, processing keyword searches with duplicate records involves many unique challenges. In this paper we discuss the motivation and present a system, RSEARCH*, to show challenges in supporting keyword search using nearly duplicate records and key techniques including identifying nearly duplicate records and generating results efficiently.*

1 Introduction

RDBMSs are very popular to store a huge amount of data due to their rigid schema specifications and mature query processing techniques. Conventional RDBMSs provide SQL interfaces and require users to understand how data is stored in it. However, some users might not know how to write an SQL to get what they are interested in, instead, they hope RDBMSs provide IR-style facilities that allow users to access the database using a set of keywords. Many recent work have studied the problem of keyword search in relational databases [1, 2, 5–7, 11–14]. They mainly focus on the mapping between keywords and data in relations and explore different semantic meanings to explain the query results without considering correlations among data.

In relational databases, duplication among tuples is one of typical data correlation. Informally, we say two records are *nearly duplicate* if they identify the same real-world entity. *Nearly duplicate records* exists in many databases due to data was collected from heterogenous sources. Figure 1 shows part of records in a real database: Science Citation Index Expanded (SCI-E)[1]. The data was collected from different publishers who use different format of author names and journal/conference names in their reference list. It is not surprising to see many *nearly duplicate records* appear as individual tuples in relations, since a real-world entity might be expressed using different values due to abbreviation, type errors, and etc.

For instance, in Figure 1, relation *Source* contains two duplicate records s_4 and s_5. Although journal names are different, we can easily find the journal name of s_4 is an abbreviation of s_5. The situation becomes a little complex in relation *Author*: three records a_2, a_3, and a_4 contain the same author name, however, if we examine

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1] http://isiknowledge.com

Author

id	name	email
a_1	Alon Y. Halevy	alon@cs.washington.edu
a_2	Halevy A	alon@cs.washington.edu
a_3	Halevy A	avinoams@clalit.org.il
a_4	Halevy A	
a_5	Halevy Alon	halevy@google.com
a_6	Halevy AY	halevy@google.com
a_7	Dong XL	lunadong@research.att.com

Write

id	aid	pid
w_1	a_1	p_6
w_2	a_2	p_7
w_3	a_3	p_5
w_4	a_4	p_1
w_5	a_4	p_2
w_6	a_4	p_4
w_7	a_5	p_2
w_8	a_6	p_3
w_9	a_7	p_2

Source

id	name	ISSN
s_1	Communication of the ACM	0001-0782
s_2	IEEE Intelligent System	1541-1672
s_3	Journal of Child Neurology	0883-0738
s_4	VLDB J.	1066-8888
s_5	VLDB Journal	1066-8888

Paper

id	title	sid	year	vol(number)	page	citation-times
p_1	Data integration with uncertainty	s_5	2009	18(2)	469-500	0
p_2	Representing uncertain data...	s_5	2009	18(5)	989-1019	0
p_3	The Claremont Report on Database Research	s_1	2009	52(6)	56-65	0
p_4	The Unreasonable Effectiveness of Data	s_2	2009	24(2)	8-12	0
p_5	... Complex Visual Hallucinations	s_3	2009	24(8)	1005-1007	0
p_6	Schema mediation ... semantic data sharing	s_4	2005	14(1)	68-83	11
p_7	MiniCon: ... answering queries using views	s_4	2001	10(2-3)	182-198	33

Figure 1: Sample database in Science Citation Index Expanded (SCI-E).

(a) Input *Halevy A* and *2009*. (b) Input *Halevy AY* and *2009*.

Figure 2: Keyword search results.

their published paper manually, we found a_2 and a_4 represent the same author Alon Y. Halevy in a_1, whereas a_3 does not. Such phenomenons result in the following two problems:

(i) users might retrieve wrong search results, or

(ii) users might miss some information that they are really interested in.

We use Example 1 to illustrate the problems.

Example 1: We used a 2-keyword query, *Alon Y. Halevy* and *2009* to search the SCI-indexed paper of Alon Y. Halevy published in 2009 in the Science Citation Index Expanded database, unfortunately we got nothing. Instead, when using *Halevy A* and *2009* we could get 5 results, 3 of them were written by Alon Y. Halevy (marked by "$\surd$" in Figure 2(a)), and the other two results were not correct. When we used *Halevy AY* and *2009*, we got a different result shown in Figure 2(b).

Ideally, we want to exclude irrelevant results and collect all correct results from relational databases to increase both precision and recall. In this paper, we present a system, RSEARCH, to analyze data between tuples in a relation and identify nearly duplicate records to enhance the keyword search ability.

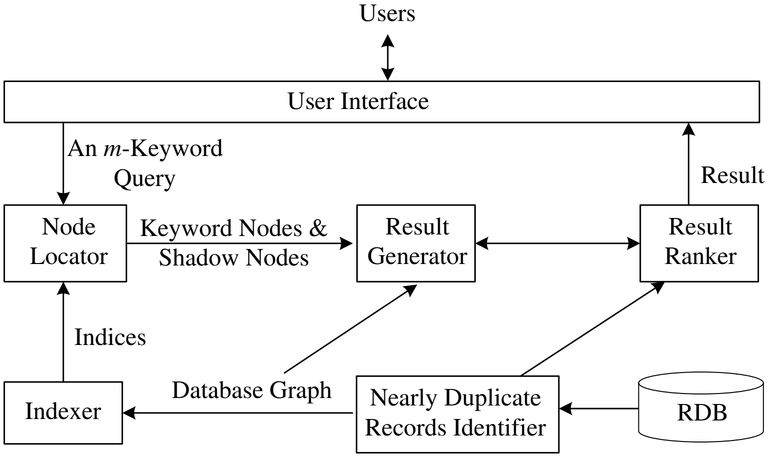

Figure 3: Architecture of RSEARCH.

In the remaining part of this paper, we introduce the architecture of RSearch in Section 2. In Section 3 we study several challenges that arise naturally when considering nearly duplicate records in the keyword search processing, and introduce the basic idea of our techniques. Finally, we discuss future directions in Section 4.

2 The RSEARCH System: Architecture

Figure 3 shows the system architecture of RSEARCH. It mainly consists of five modules: *Nearly Duplicate Records Identifier*, *Indexer*, *Node Locator*, *Result Generator*, and *Result Ranker*. We describe the process followed by a brief overview of the various modules.

- *Nearly Duplicate Records Identifier*. The *Nearly Duplicate Records Identifier* analyzes data correlation in relations, identifies nearly duplicate records, and generates a database graph.

  A relational database can be modeled as a database graph $G = (V, E_f, E_d)$. Each tuple in the database corresponds to a vertex in V, and the vertex is associated with all attribute values in the tuple. An edge $e \in E_f$ from one vertex to another one represents a foreign-key relationship. Different from the existing graph-based approaches [2, 5, 9, 13], we use E_d to express another type of edges, which we call *duplicate edges* to express the relationship between two nearly duplicate records. For simplicity, the graph can also be modeled as an undirected graph. The graph can have weights based on different semantics [2, 13]. For example, Figure 4 shows the database graph of the database tuples in Figure 1. The dotted edges represent duplicate edges.

- *Indexer*. The *Indexer* constructs indices based on database graph $G = (V, E_f, E_d)$. In addition, the *Indexer* maintains nearly duplicate records in indices. The index structure is a trie. Any token of string type in the relational database can be expressed as a path from root to a leaf in the trie. We use a node to express each of values of other data types, like integer, float, date, and etc.

- *Node Locator*. Once a user issues a query, the *Node Locator* accesses *Indexer* and retrieves matches to each token in the given keywords in the database graph $G = (V, E_f, E_d)$. We call such nodes *keyword nodes*. Besides locating keywords nodes, the *Indexer* uses duplicate edges E_d in the database graph G to locate nodes that donot contain keywords but are regarded as the same entities with the keyword nodes. We call them *shadow nodes*. There is an duplicate edge in between a keyword node and a shadow node.

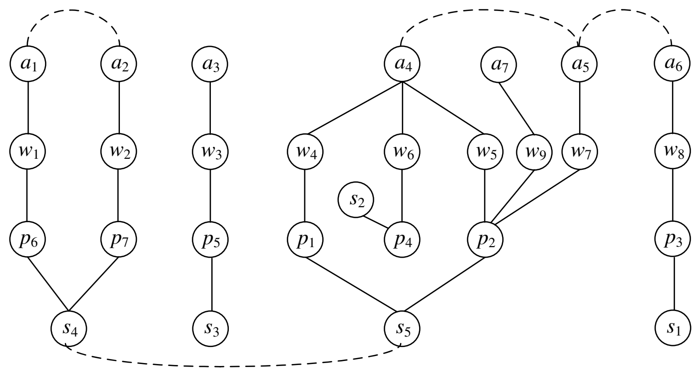

Figure 4: The database graph of the database tuples in Figure 1.

- *Result Generator*. Based on the two kinds of located nodes: keyword nodes and shadow nodes, the *Result Generator* decides how to connect them to generate results. Notice that, a search result might not contain any keyword nodes, but is also interested by users. For example, the path $a_2 - w_2 - p_7 - s_4$ in Figure 4 should be an interested result to the 2-keyword query *Alon Halevy* and *VLDB Journal*, although the result does not contain any keyword nodes. Here, both a_2 and s_4 are shadow nodes. Without considering duplicate edges in the database graph, the system could not generate this result.
- *Result Ranker*. The *Result Ranker* ranks the results generated by the *Result Generator* based on different ranking functions, such as variants of term frequency, the number of keyword matches, query result size, weight in the database graph, and etc. For simplicity, in this paper, let the weight of duplicate edges be 0. Notice that, when using the number of keyword matches, a shadow node is also a match to its corresponding keyword.

3 Key Modules in RSEARCH

In this section, we discuss key modules in RSEARCH and address several challenges of keyword search in relational databases with nearly duplicate records.

3.1 Identifying Nearly Duplicate Records

Two nearly duplicate records might look similar, for example, in Figure 1 records s_2 and s_3 in relation *Source* are similar in *name* and *ISSN* columns. In fact, only partial attributes in a record could identify duplicates. For example, the attribute *email* in *Author*, *ISSN* in *Source*, and the combination of attributes *sid*, *vol(number)*, and *page* in *Paper* could be used to identify duplicates. We call such attributes *duplicate identifiers*.

Duplicate identifier can help us to discriminate most of the nearly duplicate records. For instance, *email* is a typical duplicate identifier to distinguish a person from others. Using *email* in *Author*, we know a_1 and a_2 are duplicate, and the two records a_5 and a_6 are also duplicate.

Formally, given a relation R with a set of attributes $U = \{id, A_1, \ldots, A_n\}$, a similarity function δ, and a similarity threshold λ. A subset of attributes $X \subseteq U - \{id\}$ is a duplicate identifier on a relation R, if the following statement holds: "If for any two tuples t_i and t_j of R agree on X (i.e. $t_i[X] = t_j[X]$), their corresponding values $t_i[A_1, \ldots, A_n]$ and $t_j[A_1, \ldots, A_n]$ are similar (i.e. $\delta(t_i[A_1, \ldots, A_n], t_j[A_1, \ldots, A_n]) \leq \lambda$)." We say X approximately determines R, denoted $X \rightsquigarrow R$.

Many similarity functions have been used to evaluate the closeness of two records, such as edit distance, jaccard similarity, cosine similarity, and etc [3,4]. However, it is hard to determine which similarity function(s) and threshold value(s) should be used to evaluate similarity for different attributes.

In addition, we cannot use duplicate identifier to discriminate all nearly duplicate records. For example, we donot know whether a_4 and a_5 are duplicate since a_4 has no corresponding email address. In this case, we need more complex way to do the discrimination. For example, *Halevy A* is the abbreviation of *Halevy Alon*, and both of them write the same paper p_2. Therefore, we infer that a_4 and a_5 are also duplicate.

The above observations show that automatically identifying duplicates is technically very nontrivial, however a manually solution is labor intensive and obviously undesirable.

RSEARCH provides a smart way to identify nearly duplicate records using following steps.

(1) Finding candidate duplicate identifers. For each relation R, the *Nearly Duplicate Record Identifier* firstly finds possible duplicate identifer X (if any) in R by extracting $X \rightsquigarrow R$. Obviously, an *id* attribute of R is not a duplicate identifer. Because the number of possible duplicate identifers is exponential in the number of remaining n attributes $A_1, \ldots, A_n$ in R, efficient search techniques and pruning heuristics are essential ingredients of these approaches. Different from the soft key discovery algorithm developed in [8], we examine each attribute A_i in R and prune attributes based on the following two heuristic rules:

   (i) If at least one distinct value in an attribute A_i is frequent, then A_i should not be a duplicate identifer. For example, neither attribute *year* nor *citation-times* in Figure 1 could be a duplicate identifer.

   (ii) If the cardinality of an attribute A_i is approximately equals to the cardinality of R, then A_i could not be a duplicate identifer. For example, *title* of papers is not a duplicate identifer.

   All these operations can be done easily using SQLs inside the RDBMS. For example, we can easily use the expression COUNT(DISTINCT A_i) to count the number of distinct values in A_i and determine whether A_i should be pruned according to the maximum count number.

(2) Verification of duplicate identifers using single attribute. For the remaining attributes in R, we verify $A_i \rightsquigarrow A_j$ $(i \neq j)$ by computing similarity between values in A_j. If $A_i \rightsquigarrow A_j$ does not hold, we prune A_i from the candidate set of duplicate identifers. The basic idea is to partition values in A_i firstly. For each partition in A_i, we find the corresponding values in A_j and adopt our approximate string matching approaches [10, 15] on them to verify whether these values are similar. If they are not similar, we then conclude $A_i \rightsquigarrow A_j$ does not hold.

   Notice that, as we mentioned in the above, it is hard to choose a suitable similarity function and a threshold to determine whether two records on A_j are close enough. RSEARCH begins with a "Show me more similar samples" and learns a good similarity function and threshold based on the samples.

   If A_i approximately determines each attribute in $U - \{id\}$, we say A_i is the duplicate identifier. We conclude records in each partition of A_i are duplicates.

3.2 Result Generator: Propagation along Duplicate Edges

There are many approaches to generate results in a database graph without duplicate edges. L. Qin et al. in [12] summarizes those approaches into three types of semantics: connected tree semantics, distinct root semantics, and distinct core semantics.

Duplicate edges make the database graph more complicated and we need traverse the graph along these duplicate edges. We use connected tree semantics [2, 7, 11] to explain the problem[2]. Consider a 3-keyword query, *Luna*, *Alon Halevy*, and *uncertain*. Figure 5 shows the query results. Figure 5(a) is a result without considering any duplicate edges, which means *Luna* and *Alon Halevy* coauthored a paper p_2 on *uncertain*. When considering duplicate edges, RSEARCH generates the other four results. The result in Figure 5(b) shows

[2]Notice that our approach is orthogonal to the existing graph-based keyword search approaches, which can be adopted easily by considering duplicate edges in the database graph.

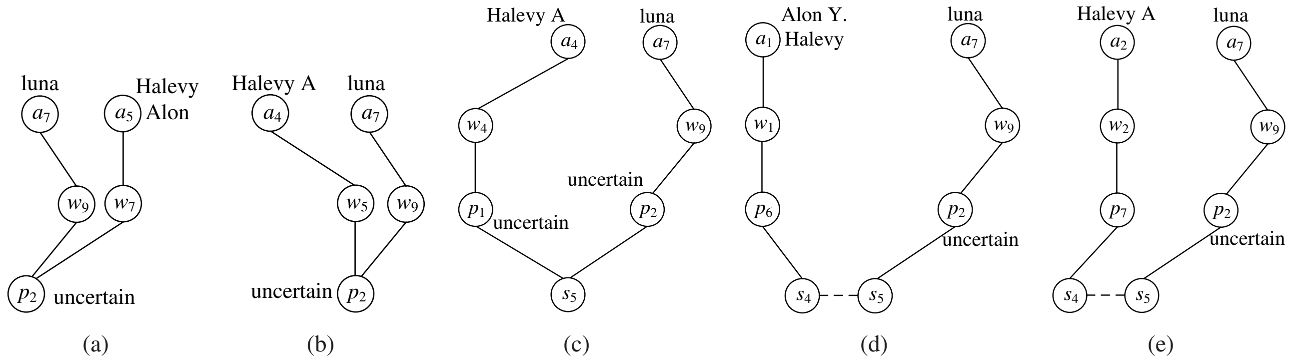

Figure 5: Query results.

the same meaning with the one in Figure 5(a) but using a keyword node a_7 and a shadow node a_4, which is propagated from the keyword node a_5. The result in Figure 5(c) means *Luna* and *Halevy A* write paper p_2 and p_1 separately. The title of these two papers contain *uncertain*. The result in Figure 5(d) means *Luna* published a paper about uncertain in a journal, meanwhile the journal publishes another paper wrote by *Alon Y. Halevy*. The result in Figure 5(e) has the similar meaning with the fourth result.

Different from the existing approaches, when a database graph contains duplicate edges, a critical problem is to decide when and how to propagate along duplicate edges. In RSEARCH, the *Node Locator* firstly locates all keyword nodes and traverses the database graph G using its foreign-key edges E_f. The *Result Generator* finds all results using keyword nodes, if any. For each result T, it expands it by propagating along duplicate edges. It uses a shadow node n_s to replace the corresponding keyword node n_k in T through a duplicate edge e. If n_s is reachable to the result through foreign-key edges E_f, we say it is a valid propagation. The *Result Generator* then generate another results using E_f. If the *Result Generator* could not generate a result using keyword nodes and E_f, it needs use duplicate edges E_d to generate more results by using the same policy of the existing approaches.

4 Conclusions and Future Work

We motivated our work by considering nearly duplicate records in relational databases and show challenges in developing a keyword search system for RDBMSs. We introduce RSEARCH system that can provide keyword search results with higher precision and recall when considering nearly duplicate records in relations. We briefly present two key techniques in RSEARCH including automatically identifying nearly duplicate records and generating query results efficiently.

Future research directions include further analyzing effects on search results of rich data correlations inside relational database. Besides nearly duplicate records, there are many types of data correlations, such as mutual exclusive of tuples, association rules in relational database. Such data correlations will greatly affect the precision and recall of search results, as well as the search efficiency. However, none of the existing approaches on relational keyword search considers data correlations. Another critical issue is to evaluate relational database search systems. So far, there are many approaches and semantics of generating search results for given keywords. The quality of a relation keyword search system greatly depends on users preference. We expect to allow maximum flexibility for using different semantics according to different preferences.

5 Acknowledgment

The work is partially supported by the National Nature Science of China (Nos. 60828004, 60973018) and the Center Education Fundamental Scientific Research (No. N090504004).

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] 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.

[3] R. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW Conference*, pages 131-140, 2007.

[4] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491-500, 2001.

[5] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *SIGMOD*, pages 305-316, 2007.

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

[7] V. Hristidis and Y. Papakonstantinou. DISCOVER: keyword search in relational databases. In *VLDB*, pages 670-681, 2002.

[8] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. CORDS: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, pages 647-658, 2004.

[9] 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.

[10] C. Li, B. Wang, and X. Yang. Vgram: Improving performance of approximate queries on string collections using variablelength grams. In *VLDB*, pages 303-314, 2007.

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

[12] L. Qin, J. X. Yu, and L. Chang. Keyword search in databases: the power of RDBMs. In *SIGMOD*, pages 681-694, 2009.

[13] L. Qin, J. X. Yu, L. Chang, and Y. Tao. Querying communities in relational databases. In *ICDE*, pages 724-735, 2009.

[14] A. Simitsis, G. Koutrika, and Y. E. Ioannidis. Précis: from unstructured keywords as queries to structured databases as answers. *VLDB J.*, 17(1): 117-149, 2008.

[15] X. Yang, B. Wang, and C. Li: Cost-Based Variable-Length-Gram Selection for String Collections to Support Approximate Queries Efficiently. In *SIGMOD*, pages 353-364, 2008.

Keyword Search in Relational Databases: A Survey

Jeffrey Xu Yu, Lu Qin, Lijun Chang
Chinese University of Hong Kong
{yu,lqin,ljchang}@se.cuhk.edu.hk

1 Introduction

The integration of DB and IR provides flexible ways for users to query information in the same platform [2, 3, 5–7, 28]. On one hand, the sophisticated DB facilities provided by RDBMSs assist users to query well-structured information using SQL. On the other hand, IR techniques allow users to search unstructured information using keywords based on scoring and ranking, and do not need users to understand any database schemas.

We survey the developments on finding *structural information* among tuples in an ***RDB*** using an l-keyword query, Q, which is a set of keywords of size l, denoted as $Q = \{k_1, k_2, \cdots, k_l\}$. Here, an ***RDB*** is viewed as a data graph $G_D(V, E)$, where V represents a set of tuples, and E represents a set of edges between tuples. An edge exists between two tuples if at least there is a foreign key reference from one to the other. A tuple consists of attribute values and some of them are strings or full-text. The structural information to be returned for an l-keyword query is a set of connected structures, $\mathcal{R}$, where a connected structure represents how the tuples, that contain the required keywords, are interconnected in a database G_D. $\mathcal{R}$ can be either all trees or all subgraphs. When a function $score(\cdot)$ is given to score a structure, we can find the top-k structures instead of all structures in G_D. Such a $score(\cdot)$ function can be based on either the text information maintained in tuples (node weights), or the connections among tuples (edge weights), or both.

In Section 2, we focus on supporting keyword search in an RDBMS using SQL. Since this implies making use of the database schema information to issue SQL queries in order to find structures for an l-keyword query, it is called the schema-based approach. The two main steps in the schema-based approach are how to generate a set of SQL queries that can find all the structures among tuples in an ***RDB*** completely, and how to evaluate the generated set of SQL queries efficiently. Due to the nature of set operations used in SQL and the underneath relational algebra, a data graph G_D is considered as an undirected graph by ignoring the direction of references between tuples, and therefore a returned structure is of undirected structure (either tree or subgraph). The existing algorithms use a parameter to control the maximum size of a structure allowed. Such a size control parameter limits the number of SQL queries to be executed. Otherwise, the number of SQL queries to be executed for finding all or even top-k structures is too large. The $score(\cdot)$ functions used to rank the structures are all based on the text information on tuples.

In Section 3, we focus on supporting keyword search in an RDBMS from a different viewpoint, by materializing an ***RDB*** as a directed graph G_D. Unlike an undirected graph, the fact that a tuple v can reach to another tuple u in a directed graph does not necessarily mean that the tuple v is reachable from u. In this context, a returned structure (either steiner tree, distinct rooted tree, r-radius steiner graph, or multi-center subgraph) is directed. Such direction handling provides users with more information on how the tuples are interconnected.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

On the other hand, it requests higher computational cost to find such structures. Many graph-based algorithms are designed to find top-k structures, where the $score(\cdot)$ functions used to rank the structures are mainly based on the connections among tuples. This type of approach is called schema-free in the sense that it does not request any database schema assistance.

2 Schema-Based Keyword Search on Relational Databases

Consider a relational database schema as a directed graph $G_S(V, E)$, called a *schema graph*, where V represents the set of relation schemas $\{R_1, R_2, \cdots, R_n\}$ and E represents the set of edges between two relation schemas. Given two relation schemas, R_i and R_j, there exists an edge in the schema graph, from R_i to R_j, denoted $R_i \rightarrow R_j$, if the primary key defined on R_i is referenced by the foreign key defined on R_j. There may exist multiple edges from R_i to R_j in G_S if there are different foreign keys defined on R_j referencing the primary key defined on R_i. In such a case, $R_i \stackrel{X}{\rightarrow} R_j$ is used, where X is the foreign key attribute names. We use $V(G)$ and $E(G)$ to denote the set of nodes and the set of edges of a graph G, respectively. In a relation schema R_i, we call an attribute, defined on strings or full-text, a text attribute, to which keyword search is allowed. A relation on relation schema R_i is an instance of the relation schema (a set of tuples) conforming to the relation schema, denoted $r(R_i)$. We use R_i to denote $r(R_i)$ if the context is obvious. A relational database (***RDB***) is a collection of relations. An ***RDB*** can be viewed as a data graph $G_D(V, E)$ on the schema graph G_S. Here, $V(G_D)$ represents a set of tuples, and $E(G_D)$ represents a set of edges between tuples. There is an edge between two tuples t_i and t_j in G_D, if there exists a foreign key reference from t_i to t_j or vice versa (undirected) in the ***RDB***. In general, two tuples, t_i and t_j are reachable if there exists a sequence of connections between t_i and t_j in G_D. The distance $dist(t_i, t_j)$ between two tuples t_i and t_j is defined as the minimum number of connections between t_i and t_j.

An l-keyword query is given as a set of keywords of size l, $Q = \{k_1, k_2, \cdots, k_l\}$, and searches interconnected tuples that contain the given keywords, where a tuple contains a keyword if a text attribute of the tuple contains the keyword. To select all tuples from a relation R that contain a keyword k_1, a predicate $contain(A, k_1)$ is supported in SQL in IBM *DB2*, *ORACLE*, and Microsoft SQL-SERVER, where A is a text attribute in R. The following SQL query, finds all tuples in R containing k_1 provided that the attributes A_1 and A_2 are all and the only text attributes in relation R. We say a tuple contains a keyword, for example k_1, if the tuple is included in the result of such a selection.

select * **from** R **where** $contain(A_1, k_1)$ **or** $contain(A_2, k_1)$

An l-keyword query returns a set of answers, where an answer is a minimal total joining network of tuples (***MTJNT***) [1, 16] that is defined as follows. Given an l-keyword query and a relational database with schema graph G_S, a joining network of tuples (***JNT***) is a connected tree of tuples where every two adjacent tuples, $t_i \in r(R_i)$ and $t_j \in r(R_j)$ can be joined based on the foreign key reference defined on relational schema R_i and R_j in G_S (either $R_i \rightarrow R_j$ or $R_j \rightarrow R_i$). An ***MTJNT*** is a joining network of tuples that satisfy the following two conditions, total and minimal. By total, each keyword in the query must be contained in at least one tuple of the joining network. By minimal, a joining network of tuples is not total if any tuple is removed.

Because it is meaningless if two tuples in an ***MTJNT*** are too far away from each other, a size control parameter, Tmax, is introduced to specify the maximum number of tuples allowed in an ***MTJNT***.

Given an ***RDB*** on the schema graph G_S, in order to generate all the ***MTJNT***s for an l-keyword query, Q, *keyword relation* and *Candidate Network* (***CN***) are defined as follows. A keyword relation $R_i\{K'\}$ is a subset of relation R_i containing tuples that only contain keywords $K' (\subseteq Q)$) and no other keywords, as defined below:

$$R_i\{K'\} = \{t | t \in r(R_i) \land \forall k \in K', t \text{ contains } k \land \forall k \in (K - K'), t \text{ does not contain } k\}$$

where K is the set of keywords in Q, i.e. $K = Q$. K' can be $\emptyset$. In such a situation, $R_i\{\}$ consists of tuples that do not contain any keywords in Q and is called an empty keyword relation. A candidate network (*CN*) is a connected tree of keyword relations where for every two adjacent keyword relations $R_i\{K_1\}$ and $R_j\{K_2\}$, we have $(R_i, R_j) \in E(G_S)$ or $(R_j, R_i) \in E(G_S)$. A candidate network must satisfy the following two conditions, total and minimal. By total, each keyword in the query must be contained in at least one keyword relation of the candidate network. By minimal, a candidate network is not total if any keyword relation is removed.

A *CN* can produce a set of (possibly empty) *MTJNT*s, and it corresponds to a relational algebra that joins a sequence of relations to obtain *MTJNT*s over the relations involved. Given a keyword query Q and an *RDB* with schema graph G_S, let $\mathcal{C} = \{C_1, C_2, \cdots\}$ be the set of all candidate networks for Q over G_S, and let $\mathcal{T} = \{T_1, T_2, \cdots\}$ be the set of all *MTJNT*s for Q over the *RDB*. For every $T_i \in \mathcal{T}$, there is exactly one $C_j \in \mathcal{C}$ that produces T_i.

For an l-keyword query over an *RDB*, the number of *MTJNT*s can be very large even if Tmax is small. It is ineffective to present users a huge number of results for a keyword query. In order to handle the effectiveness, for each *MTJNT*, T, for a keyword query Q, it also allows a score function $score(T, Q)$ defined on T in order to rank results. A top-k keyword query retrieves k *MTJNT*s $\mathcal{T} = \{T_1, T_2, ..., T_k\}$ such that for any two *MTJNT*s T and T' where $T \in \mathcal{T}$ and $T' \notin \mathcal{T}$, $score(T, Q) \leq score(T', Q)$.

Ranking issues for *MTJNT*s are discussed in many papers [14, 22, 23]. They aim at designing effective ranking functions that capture both the textual information (e.g., IR-Styled ranking) and structural information (e.g., the size of the *MTJNT*) for an *MTJNT*. There are two categories of ranking functions, namely, the attribute level ranking function and the tree level ranking function. Given an *MTJNT* T and a keyword query Q, the attribute level ranking function first assigns each text attribute for tuples in T an individual score and then combines them together to get the final score [14, 22]. In other words, in the attribute level ranking functions, each text attribute of an *MTJNT* is considered as a virtual document. Tree level ranking functions consider the whole *MTJNT* as a virtual document rather than each individual text attribute [23].

In the framework of RDBMS, the two main steps of processing an l-keyword query are candidate network generation and candidate network evaluation.

1. **Candidate Network Generation:** In the candidate network generation step, a set of candidate networks $\mathcal{C} = \{C_1, C_2, \cdots\}$ is generated over a graph schema G_S. The set of *CN*s shall be complete and duplication-free. The former ensures that all *MTJNT*s are found, and the latter is mainly for efficiency consideration.
2. **Candidate Network Evaluation:** In the candidate network evaluation step, all $C_i \in \mathcal{C}$ are evaluated.

We will introduce the two steps one by one in the next two sections.

2.1 Candidate Network Generation

In order to generate all candidate networks for an l-keyword query Q over an *RDB* with schema graph G_S, algorithms are designed to generate candidate networks $\mathcal{C} = \{C_1, C_2, ...\}$ that satisfy the following two conditions:

- **Complete:** For each solution T of the keyword query, there exists a candidate network $C_i \in \mathcal{C}$ that can produce T.
- **Duplication-Free:** For every two *CN*s $C_i \in \mathcal{C}$ and $C_j \in \mathcal{C}$, C_i and C_j are not isomorphic to each other.

The complete and duplication-free conditions ensure that (1) all results (*MTJNT*s) for a keyword query will be produced by the set of *CN*s generated (due to completeness); and (2) any result T for a keyword query will be produced only once, i.e., there does not exist two *CN*s $C_i \in \mathcal{C}$ and $C_j \in \mathcal{C}$ such that C_i and C_j both produce T (due to the duplication-free condition).

The first algorithm to generate all *CN*s was proposed in *DISCOVER* [16]. It expands the partial *CN*s generated to larger partial *CN*s until all *CN*s are generated. As the number of partial *CN*s can be exponentially large, arbitrarily expanding will make the algorithm extremely inefficient. In *DISCOVER* [16], there are three pruning rules for partial *CN*s.

- **Rule-1:** Duplicated *CN*s are pruned (based on tree isomorphism).
- **Rule-2:** A *CN* can be pruned if it contains all the keywords and there is a leaf node, $R_j\{K'\}$, where $K' = \emptyset$, because it will generate results that do not satisfy the condition of minimality.
- **Rule-3:** When there only exists a single foreign key reference between two relation schemas (for example, $R_i \rightarrow R_j$), *CN*s including $R_i\{K_1\} \rightarrow R_j\{K_2\} \leftarrow R_i\{K_3\}$ will be pruned, where K_1, K_2, and K_3 are three subsets of Q, and $R_i\{K_1\}$, $R_j\{K_2\}$, and $R_i\{K_3\}$ are keyword relations.

The Rule-3 reflects the fact that the primary key defined on R_i and a tuple in the relation of $R_j\{K_2\}$ must refer to the same tuple appearing in both relations $R_i\{K_1\}$ and $R_i\{K_3\}$. As the same tuple cannot appear in two sub-relations in a *CN* (otherwise, it will not produce a valid *MTJNT* because the minimal condition will not be satisfied), the join results for $R_i\{K_1\} \rightarrow R_j\{K_2\} \leftarrow R_i\{K_3\}$ will not contain any valid *MTJNT*.

The algorithm in [16] can generate a complete and duplication-free set of *CN*s, but the cost of generating the set of *CN*s is high. *S-KWS* [24] proposes an algorithm (1) to reduce the number of partial results generated by expanding from part of the nodes in a partial tree and (2) to avoid isomorphism testing by assigning a proper expansion order.

2.2 Candidate Network Evaluation

After generating all candidate networks (*CN*s) in the first phase, the second phase is to evaluate all candidate networks in order to get the final results. *DBXplorer* [1], *DISCOVER* [16], *S-KWS* [24], *KDynamic* [27] and *KRDBMS* [25] compute all *MTJNT*s upon the set of *CN*s generated by specifying a proper execution plan. *DISCOVER-II* [14] and *SPARK* [23] compute top-k *MTJNT*s.

2.2.1 Getting all *MTJNT*s in a relational database

In RDBMS, the problem of evaluating all *CN*s in order to get all *MTJNT*s is a multi-query optimization problem. There are two main issues: (1) How to share common subexpressions among *CN*s generated in order to reduce computational cost when evaluating. (2) How to find a proper join order to fast evaluate all *CN*s. For a keyword query, the number of *CN*s generated can be very large. Given a large number of joins, it is extremely difficult to obtain an optimal query processing plan, because one best plan for a *CN* may slow down others, if its subtrees are shared by other *CN*s. As studied in *DISCOVER* [16], finding the optimal execution plan is an NP-complete problem.

In *DISCOVER* [16], an algorithm is proposed to evaluate all *CN*s together using a greedy algorithm based on the following observations: (1) subexpressions that are shared by most *CN*s should be evaluated first; and (2) subexpressions that may generate the smallest number of results should be evaluated first. In *S-KWS* [24], in order to share the computational cost of evaluating all *CN*s, Markowetz et al. construct an operator mesh. In a mesh, there are $n \cdot 2^{l-1}$ clusters, where n is the number of relations in the schema graph G_S and l is the number of keywords. A cluster consists of a set of operator trees (left-deep trees) that share common expressions. When evaluating all *CN*s in a mesh, a projected relation with the smallest number of tuples is selected to start and to join. In *KDynamic* [27], a $\mathcal{L}$-Lattice is introduced to share computational cost among *CN*s. Given a set of *CN*s, $\mathcal{C}$, it defines the root of each *CN* to be the node r such that the maximum length of the path from r to all leaf nodes of the *CN* is minimized. There are three main differences between the Mesh and the $\mathcal{L}$-Lattice. (1) The maximum depth of a Mesh is $\mathsf{Tmax} - 1$ and the maximum depth of an $\mathcal{L}$-Lattice is $\lfloor \mathsf{Tmax}/2 + 1 \rfloor$. (2) In a mesh, only the left part of two *CN*s can be shared (except for the leaf nodes), while in an $\mathcal{L}$-Lattice multiple parts of two *CN*s can be shared. (3) The number of leaf nodes in a mesh is $O((|V(G_S)| \cdot 2^l)^2)$ because there are $O(|V(G_S)| \cdot 2^l)$ clusters in a mesh and each cluster may contain $O(|V(G_S)| \cdot 2^l)$ leaf nodes. The number of leaf nodes in an $\mathcal{L}$-Lattice is $O(2^l)$.

After sharing computational cost using either the Mesh or the $\mathcal{L}$-Lattice, all *CN*s are evaluated using joins in *DISCOVER* or *S-KWS*. In *KRDBMS* [25], the authors observe that evaluating all *CN*s using only joins may always generate a large number of temporary tuples. They propose to use semijoin/join sequences to evaluate a *CN*.

Besides evaluating all *CN*s in a static environment, *S-KWS* and *KDynamic* focus on monitoring all *MTJNT*s in a relational data stream, where tuples can be inserted/deleted frequently. In this situation, it is necessary to find new *MTJNT*s or expire *MTJNT*s in order to monitor events that are implicitly interrelated over an open-ended relational data stream for a user-given l-keyword query. In other words, it reports new *MTJNT*s when new tuples are inserted, and, in addition, reports the *MTJNT*s that become invalid when tuples are deleted. A sliding window (time interval), W, is specified. A tuple, t, has a lifespan from its insertion into the window at time $t.start$ to $W + t.start - 1$, if t is not deleted before then. Two tuples can be joined if their lifespans overlap.

2.2.2 Getting top-k *MTJNT*s in a relational database

A naive approach to answer the top-k keyword queries is to first generate all *MTJNT*s using the algorithms proposed in Section 2.2.1, and then calculate the score for each *MTJNT*, and finally output the top-k *MTJNT*s with the highest scores. In *DISCOVER-II* [14] and *SPARK* [23], several algorithms are proposed to get top-k *MTJNT*s efficiently. The aim of all the algorithms is to find a proper order of generating *MTJNT*s in order to stop early before all *MTJNT*s are generated.

In *DISCOVER-II*, three algorithms are proposed to get top-k *MTJNT*s, namely, the Sparse algorithm, the Single-Pipelined algorithm, and the Global-Pipelined algorithm. All algorithms are based on the attribute level ranking function, which has the property of *tuple monotonicity*, defined as follows. For any two *MTJNT*s $T = t_1 \bowtie t_2 \bowtie ... \bowtie t_l$ and $T' = t'_1 \bowtie t'_2 \bowtie ... \bowtie t'_l$ generated from the same *CN* C, if for any $1 \leq i \leq l$, $score(t_i, Q) \leq score(t'_i, Q)$, then we have $score(T, Q) \leq score(T', Q)$. With the tuple monotonicity, the algorithms are designed to stop early where possible.

In *SPARK* [23], the authors study the tree level ranking function which does not satisfy tuple monotonicity. In order to handle such non-monotonic score functions, a new monotonic upper bound function is introduced. The intuition behind the upper bound function is that, if the upper bound score is already smaller than the score of a certain result, then all the upper bound scores of unseen tuples will be smaller than the score of this result due to the monotonicity of the upper bound function. Two new algorithms are proposed in *SPARK*: Skyline-Sweeping and Block-Pipelined.

2.3 Other Keyword Search Semantics

In the above discussions, for an l-keyword query on an *RDB*, each result is an *MTJNT*. This is referred to as the *connected tree semantics*. There are two other semantics to answer an l-keyword query on an *RDB*, namely *distinct root semantics* and *distinct core semantics*.

Distinct Root Semantics: An l-keyword query finds a collection of tuples that contain all the keywords and that are reachable from a root tuple (center) within a user-given distance (Dmax). The distinct root semantics implies that the same root tuple determines the tuples uniquely [9, 13, 15, 21, 25]. Suppose that there is a result rooted at tuple t_r. For any of the l keywords, say k_i, there is a tuple t in the result that satisfies the following conditions: (1) t contains keyword k_i, (2) among all tuples that contain k_i, the distance between t and t_r is minimum[1], and (3) the minimum distance between t and t_r must be less than or equal to a user given parameter Dmax.

[1] If there is a tie, then a tuple is selected with a predefined order among tuples in practice.

Distinct Core Semantics: An l-keyword query finds multi-center subgraphs, called communities [25]. A community, $C_i(V, E)$, is specified as follows. V is a union of three subsets of tuples, $V = V_c \cup V_k \cup V_p$, where, V_k is a set of keyword-tuples where a keyword-tuple $v_k \in V_k$ contains at least a keyword and all l keywords in the given l-keyword query must appear in at least one keyword-tuple in V_k; V_c is a set of center-tuples where there exists at least a sequence of connections between $v_c \in V_c$ and every $v_k \in V_k$ such that $dist(v_c, v_k) \leq \mathsf{Dmax}$; and V_p is a set of path-tuples that appear on a shortest sequence of connections from a center-tuple $v_c \in V_c$ to a keyword-tuple $v_k \in V_k$ if $dist(v_c, v_k) \leq \mathsf{Dmax}$. Note that a tuple may serve several roles as keyword/center/path tuples in a community. E is a set of connections for every pair of tuples in V if they are connected over shortest paths from nodes in V_c to nodes in V_k. A community, C_i, is uniquely determined by the set of keyword tuples, V_k, which is called the core of the community, and denoted as $\mathsf{core}(C_i)$.

In [25], the authors showed tuple-reduction approaches to process different semantics using SQL in RDBMSs.

3 Graph-Based Keyword Search

A data graph G_D can be considered as materialization of an *RDB*. In this section, we show how to answer keyword queries using graph algorithms. We consider a weighted directed graph in this section, $G_D(V, E)$. Weights are assigned to edges to reflect the (directional) proximity of the corresponding tuples, denoted as $w_e(\langle u, v\rangle)$. A commonly used weighting scheme [4, 10] is as follows. For a foreign key reference from t_u to t_v, the weight for the directed edge $\langle u, v\rangle$ is given as Eq. 4, and the weight for the backward edge $\langle v, u\rangle$ is given as Eq. 5.

$$w_e(\langle u, v\rangle) = 1 \tag{4}$$
$$w_e(\langle v, u\rangle) = \log_2(1 + N_{in}(v)) \tag{5}$$

where $N_{in}(v)$ is the number of tuples that refer to t_v, which is the tuple corresponding to node v. Nodes can have weights. But, because the algorithms that deal with edge-weighted graphs can be easily modified to handle additional node-weights, below we assume that only edges have weights. We denote the number of nodes and the number of edges in graph, G_D, using $n = |V(G_D)|$ and $m = |E(G_D)|$.

There are different structures of tuples to be returned: (1) a reduced tree that contains all the keywords, that we refer to as *tree-based semantics*; (2) a subgraph, such as r-radius steiner graph [21], and multi-center induced graph [26], we call this *subgraph-based semantics*. In the following, we focus the tree-based semantics, and we will discuss the subgraph-based semantics in Section 3.3.

In the tree-based semantics, an answer to Q (called a Q-SUBTREE) is defined as any subtree T of G_D that is reduced with respect to Q. Formally, there exists a sequence of l nodes in T, $\langle v_1, \cdots, v_l\rangle$ where $v_i \in V(T)$ and v_i contains keyword term k_i for $1 \leq i \leq l$, such that the leaves of T can only come from those nodes, i.e. $leaves(T) \subseteq \{v_1, v_2, \cdots, v_l\}$, the root of T should also be from those nodes if it has only one child, i.e. $root(T) \in \{v_1, v_2, \cdots, v_l\}$.

The Q-SUBTREE is popularly used to describe answers to keyword queries. Two different weight functions are proposed in the literature to rank Q-SUBTREEs in increasing weight order, and two semantics are proposed based on the two weight functions, namely *steiner tree-based semantics*, and *distinct root-based semantics*.

Steiner Tree-Based Semantics: In this semantics, the weight of a Q-SUBTREE is defined as the total weight of the edges in the tree; formally,

$$w(T) = \sum_{\langle u,v\rangle \in E(T)} w_e(\langle u, v\rangle) \tag{6}$$

where $E(T)$ is the set of edges in T. The l-keyword query finds all (or top-k) Q-SUBTREEs in weight increasing order, where the weight denotes the cost to connect the l keywords. Under this semantics, finding the Q-SUBTREE with the smallest weight is the well-known *optimal steiner tree problem* which is NP-complete [11].

Distinct Root-Based Semantics: Since the problem of keyword search under the steiner tree-based semantics is generally a hard problem, many works resort to easier semantics. Under the distinct root-based semantics, the weight of a Q-SUBTREE is the sum of the shortest distance from the root to each keyword node; more precisely,

$$w(T) = \sum_{i=1}^{l} dist(root(T), k_i) \tag{7}$$

where $root(T)$ is the root of T, $dist(root(T), k_i)$ is the shortest distance from the root to the keyword node k_i.

There are two differences between the two semantics. First is the weight function as shown above. The other difference is the total number of Q-SUBTREEs for a keyword query. In theory, there can be exponentially many Q-SUBTREEs under the steiner tree semantics, i.e., $O(2^m)$ where m is the number of edges in G_D. But, under the distinct root semantics, there can be at most n, which is the number of nodes in G_D, Q-SUBTREEs, i.e. zero or one Q-SUBTREE rooted at each node $v \in V(G_D)$. The potential Q-SUBTREE rooted at v is the union of the shortest path from v to each keyword node k_i.

3.1 Steiner Tree-Based Keyword Search

In this section, we show three categories of algorithms under the steiner tree-based semantics. First is the backward search algorithm, where the first tree returned is an l-approximation of the optimal steiner tree. Second is a dynamic programming approach, which finds the optimal (top-1) steiner tree in time $O(3^l n + 2^l((l + \log n)n + m))$. Third is enumeration algorithms with polynomial delay.

3.1.1 Backward Search

BANKS-I [4] enumerates Q-SUBTREEs using a backward search algorithm searching backwards from the nodes that contain keywords. Given a set of l keywords, they first find the set of nodes that contain keywords, S_i, for each keyword term k_i, i.e. S_i is exactly the set of nodes in $V(G_D)$ that contain the keyword term k_i. This step can be accomplished efficiently using an inverted list index. Let $\mathcal{S} = \bigcup_{i=1}^{l} S_i$. Then, the backward search algorithm concurrently runs $|\mathcal{S}|$ copies of Dijkstra's single source shortest path algorithm, one for each keyword node v in $\mathcal{S}$ with node v as the source. The $|\mathcal{S}|$ copies of Dijkstra's algorithm run concurrently using iterators. All the Dijkstra's single source shortest path algorithms traverse graph G_D in reverse direction. When an iterator for keyword node v visits a node u, it finds a shortest path from u to the keyword node v. The idea of concurrent backward search is to find a common node from which there exists a shortest path to at least one node in each set S_i. Such paths will define a rooted directed tree with the common node as the root and the corresponding keyword nodes as the leaves.

The connected trees computed by BANKS-I are approximately sorted in increasing weight order. Computing all the connected trees followed by sorting would increase the computation time and also lead to a greatly increased time to output the first result. A fixed-size heap is maintained as a buffer for the computed connected trees. Newly computed trees are added into the heap. Whenever the heap is full, the top result tree is output and removed. With BANKS-I, the first Q-SUBTREE output is an l-approximation of the optimal steiner tree, and the Q-SUBTREEs are computed in increasing height order. The Q-SUBTREEs computed by BANKS-I is not complete, as BANKS-I only considers the shortest path from the root of a tree to nodes containing keywords.

3.1.2 Dynamic Programming

Although finding the optimal steiner tree (top-1 Q-SUBTREE under the steiner tree-based semantics) or group steiner tree is NP-complete in general, there are efficient algorithms to find the optimal steiner tree for l-keyword queries [10,19], because l is small. The algorithm [10] solves the group steiner tree problem. Note that the group steiner tree in a directed (or undirected) graph can be transformed into steiner tree problem in directed graph.

Let $\mathbf{k}, \mathbf{k1}, \mathbf{k2}$ denote a non-empty subset of the keyword nodes $\{k_1, \cdots, k_l\}$. Let $T(v, \mathbf{k})$ denote the tree with the minimum weight (called it *optimal tree*) among all the trees rooted at v and containing all the keyword nodes in $\mathbf{k}$. We can find the optimal tree $T(v, \mathbf{k})$ for each $v \in V(G_D)$ and $\mathbf{k} \subseteq Q$. Initially, for each keyword node k_i, $T(k_i, \{k_i\})$ is a single node tree consisting of the keyword node k_i with tree weight 0. For a general case, the $T(v, \mathbf{k})$ can be computed by the following equations.

$$T(v, \mathbf{k}) = \min(T_g(v, \mathbf{k}), T_m(v, \mathbf{k})) \quad (8)$$

$$T_g(v, \mathbf{k}) = \min_{\langle v,u\rangle \in E(G_D)} \{\langle v, u\rangle \oplus T(u, \mathbf{k})\} \quad (9)$$

$$T_m(g, \mathbf{k1} \cup \mathbf{k2}) = \min_{\mathbf{k1} \cap \mathbf{k2} = \emptyset} \{T(v, \mathbf{k1}) \oplus T(v, \mathbf{k2})\} \quad (10)$$

Here, $\min$ means to choose the tree with minimum weight from all the trees in the argument. Note that, $T(v, \mathbf{k})$ may not exist for some v and $\mathbf{k}$, which reflects that node v can not reach some of the keyword nodes in $\mathbf{k}$, then $T(v, \mathbf{k}) = \perp$ with weight ∞. $T_g(v, \mathbf{k})$ reflects the tree grow case, and $T_m(v, \mathbf{k})$ reflects the tree merge case. An algorithm called DPBF for Best-First Dynamic Programming is proposed in [10]. Consider a graph G_D with n nodes and m edges, DPBF finds the optimal steiner three containing all the keywords in $Q = \{k_1, \cdots, k_l\}$, in time $O(3^l n + 2^l((l + n) \log n + m))$ [10].

DPBF can be modified slightly to output k steiner trees in increasing weight order, denoted as DPBF-k, by terminating DPBF after finding k steiner trees that contain all the keywords.

3.1.3 Enumerating Q-SUBTREEs with Polynomial Delay

Although BANKS-I can find an l-approximation of the optimal Q-SUBTREE and DPBF can find the optimal Q-SUBTREE, the non-first results returned by these algorithms can not guarantee their quality (or approximation ratio), and the delay between consecutive results can be very large. In [12, 18], the authors show three algorithms to enumerate Q-SUBTREEs in increasing (or θ-approximate increasing) weight order with polynomial delay: (1) an enumeration algorithm enumerates Q-SUBTREEs in increasing *weight* order with polynomial delay under the *data complexity*, (2) an enumeration algorithm enumerates Q-SUBTREEs in $(\theta + 1)$-approximate *weight* order with polynomial delay under *data-and-query complexity*, (3) an enumeration algorithm enumerates Q-SUBTREEs in 2-approximate *height* order with polynomial delay under *data-and-query complexity*. The algorithms are adaption of the Lawler's procedure [20] to enumerate Q-SUBTREEs in rank order.

3.2 Distinct Root-Based Keyword Search

In this section, we show approaches to find Q-SUBTREEs using the distinct root semantics, where the weight of a tree is defined as the sum of the shortest distance from the root to each keyword node. As shown in the previous section, the problem of keyword search under the directed steiner tree is, in general, a hard problem. Using the distinct root semantics, there can be at most n Q-SUBTREEs for a keyword query, and in the worst case, all the Q-SUBTREEs can be found in time $O(l(n \log n + m))$. The approaches introduced in this section deal with very large graphs in general, and they propose search strategies or indexing schemes to reduce the search time for an online keyword query.

3.2.1 Bidirectional Search

BANKS-I algorithm can be directly applied to the distinct root semantics. It would explore an unnecessarily large number of nodes in the following two scenarios. First, the query contains a frequently occurring keyword. In BANKS-I, one iterator is associated with each keyword node. The algorithm would generate a large number of iterators if a keyword matches a large number of nodes. Second, an iterator reaches a node with large fan-in (incoming edges). An iterator may need to explore a large number of nodes if it hits a node with a very large fan-in. BANKS-II [17] is proposed to overcome the drawbacks of BANKS-I. The main idea of bidirectional search is to start forward searches from potential roots. The main differences of bidirectional search from BANKS-I are as follows. First, all the single source shortest path iterators from the BANKS-I algorithm are merged into a single iterator, called the *incoming iterator*. Second, an *outgoing iterator* runs concurrently, which follows forwarding edges starting from all the nodes explored by the *incoming iterator*. Third, *spreading activation* is proposed to prioritize the search, which chooses *incoming iterator* or *outgoing iterator* to be called next. It also chooses the next node to be visited in the *incoming iterator* or *outgoing iterator*.

3.2.2 Bi-level Indexing

BLINKS [13] is proposed as a bi-level index to speed up BANKS-II, as no index (except the keyword-node index) is used in BANKS-II. A naive index precomputes and indexes all the distances from the nodes to keywords, but this will incur very large index size, as the number of distinct keywords is in the order of the size of the data graph G_D. A bi-level index can be built by first partitioning graph, and then building intra-block index and block index. Two node-based partitioning methods are proposed to partition a graph into blocks, namely, BFS-Based Partitioning, and METIS-Based Partitioning.

In [13], in a node-based partitioning of a graph, a node separator is called a *portal* node (or portal). A block consists of all nodes in a partition as well as all portals incident to the partition. For a block, a portal can be either "in-portal", "out-portal", or both. A portal is called in-portal if it has at least one incoming edge from another block and at least one outgoing edge in this block. And a portal is called out-portal if it has at least one outgoing edge to another block and at least one incoming edge from this block.

For each block b, the intra-block index (IB-index) is built.

In *BLINKS* [13], a priority queue $\mathcal{Q}_i$ of cursors is created for each keyword term k_i to simulate Dijkstra's algorithm by utilizing the distance information stored in the IB-index. Initially, for each keyword k_i, all the blocks that contain it are found by the keyword-block list, and a cursor is created to scan each intra-block keyword-node list and put in queue $\mathcal{Q}_i$. When an in-portal u is visited, all the blocks that have u as their out-portal need to be expanded, because a shorter path may cross several blocks.

3.2.3 External Memory Data Graph

Dalvi et al. study keyword search on graphs where the graph G_D can not fit into main memory [9]. They build a much smaller supernode graph on top of G_D that can resident in main memory. The supernode graph is defined as follows:

- **SuperNode:** The graph G_D is partitioned into components by a clustering algorithm, and each cluster is represented by a node called the *supernode* in the top-level graph. Each supernode thus contains a subset of $V(G_D)$, and the contained nodes (nodes in G_D) are called *innernodes*.

- **SuperEdge:** The edges between the supernodes called superedges are constructed as follows: if there is at least one edge from an innernode of supernode s_1 to an innernode of supernode s_2, then there exists a superedge from s_1 to s_2.

A supernode graph is constructed that fits into main memory, where each supernode has a fixed number of innernodes and is stored on disk.

A *multi-granular graph* is used to exploit information presented in lower-level nodes (innernodes) that are cache-resident at the time a query is executed. A multi-granular graph is a hybrid graph that contains both supernodes and innernodes. A supernode is present either in *expanded* form, i.e., all its innernodes along with their adjacency lists are present in the cache, or in *unexpanded* form, i.e., its innernodes are not in the cache.

The innernodes and their adjacency lists are handled in the unit of supernodes, i.e. either all or none of the innernodes of a supernode are presented in the cache.

When searching the multi-granular graph, the answers generated may contain supernodes, called *supernode answer*. If an answer does not contain any supernodes, it is called *pure answer*. The final answer returned to users must be pure answer. The Iterative Expansion Search algorithm (IES) [9] is a multi-stage algorithm that is applicable to multi-granular graphs. Each iteration of IES can be broken up into two phases.

- **Explore phase:** Run an in-memory search algorithm on the current state of the multi-granular graph. The multi-granular graph is entirely in memory, whereas the supernode graph is stored in main memory, and details of expanded supernodes are stored in cache. When the search reaches an expanded supernode, it searches on the corresponding innernodes in cache.

- **Expand phase:** Expand the supernodes found in top-n ($n > k$) results of the previous phase and add them to input graph to produce an expanded multi-granular graph, by loading all the corresponding innernodes into cache.

The graph produced at the end of Expand phase of iteration i acts as the graph for iteration $i+1$. The algorithm stops when all top-k results are pure.

3.3 Subgraph-Based Keyword Search

The previous sections define the answer of a keyword query as Q-SUBTREE, which is a directed subtree. We show two subgraph-based notions of answer definition for a keyword query in the following, namely, r-radius steiner graph, and multi-center induced graph.

3.3.1 r-radius steiner graph

Li et al. in [21] define the result of an l-keyword query as an r-radius steiner subgraph. The graph is unweighted and undirected, and the length of a path is defined as the number of edges in it. The definition of r-radius steiner graph is based on *centric distance* and *radius*. The centric distance of v in G, denoted as $CD(v)$, is the maximum among the shortest distances between v and any node $u \in V(G)$, i.e. $CD(v) = \max_{u \in V(G)} dist(u, v)$. The radius of a graph G, denoted as $\mathcal{R}(G)$, is the minimum value among the centric distances of every node in G, i.e. $\mathcal{R}(G) = \min_{v \in V(G)} CD(v)$. G is called an r-radius graph if its radius is exactly r.

Given an r-radius graph G and a keyword query Q, node v in G is called a content node if it contains some of the input keywords. Node s is called steiner node if there exist two content nodes, u and v, and s in on the simple path between u and v. The subgraph of G composed of the steiner nodes and associated edges is called an r-radius steiner graph (SG). The radius of an r-radius steiner graph can be smaller than r.

The result of an l-keyword query, with a given radius, is a set of r-radius steiner graphs. The approaches to find r-radius steiner graphs are based on finding r-radius subgraphs using the adjacency matrix, $M = (m_{ij})_{n \times n}$, with respect to G_D, which is a $n \times n$ Boolean matrix. An element m_{ij} is 1, if and only if there is an edge between v_i and v_j, m_{ii} is 1 for all i. $M^r = M \times M \cdots \times M = (m_{ij})_{n \times n}$ is the r-th power of adjacency matrix M. An element m^r_{ij} is 1, if and only if the shortest path between v_i and v_j is less than or equal to r. $N^r_i = \{v_j | m^r_{ij} = 1\}$ is the set of nodes that have a path to v_i with distance no larger than r. G^r_i denotes the subgraph induced by the node set N^r_i. We use $G_i \trianglelefteq G_j$ to denote that G_i is a subgraph of G_j. The r-radius subgraph is defined based on G^r_i's. The following lemma is used to find all the r-radius subgraphs [21].

Lemma 1: [21] Given a graph G, with $\mathcal{R}(G) \geq r > 1$, $\forall i, 1 \leq i \leq |V(G)|$, G^r_i is an r-radius subgraph, if, $\forall v_k \in N^r_i$, $N^r_i \not\subseteq N^{r-1}_k$.

Note that, the above lemma is a sufficient condition for identifying r-radius subgraphs, but not a necessary condition. [21] only considers $n = |V(G)|$ subgraphs, each is uniquely determined by one node in G.

An r-radius subgraph G_i^r is maximal if and only if there is no other r-radius subgraph G_j^r that is a super graph of G_i^r, i.e. $G_i^r \trianglelefteq G_j^r$. [21] considers those maximal r-radius subgraphs G_i^r as the subgraphs that will generate r-radius steiner subgraphs. All these maximal r-radius subgraphs G_i^r are found, which can be pre-computed and indexed on the disk, because these maximal r-radius graph are query independent.

3.3.2 Multi-Center Induced Graph

In contrast to tree-based results that are single-center (root) induced trees, query answers can be multi-centered induced subgraphs of G_D. These are referred to as *communities* [26]. The nodes of a community $R(V, E)$, $V(R)$ is a union of three subsets, $V = V_c \cup V_l \cup V_p$, where V_l represents a set of keyword nodes (*knode*), V_c represents a set of center nodes (*cnode*) (for every *cnode* $v_c \in V_c$, there exists at least a single path such that $dist(v_c, v_l) \leq R_{max}$ for any $v_l \in V_l$, where R_{max} is introduced to control the size of a community), and V_p represents a set path nodes (*pnode*) that include all the nodes that appear on any path from a *cnode* $v_c \in V_c$ to a *knode* $v_l \in V_l$ with $dist(v_c, v_l) \leq R_{max}$. $E(R)$ is the set of edges induced by $V(R)$.

A community, R, is uniquely determined by the set of *knode*s, V_l, which is called the core of the community. The weight of a community R, $w(R)$ is defined as the minimum value among the total edge weights from a *cnode* to every *knode*; more precisely,

$$w(R) = \min_{v_c \in V_c} \sum_{v_l \in V_l} dist(v_c, v_l). \tag{11}$$

Algorithms are proposed in [26] to compute communities by adopting the Lawler's procedure [20].

4 Conclusion Remarks

In this article, we surveyed some main results on finding structural information in an *RDB* for an l-keyword query. The current work focus on identifying primitive structures as answers and efficiently computing all and/or top-k of such answers. One future work is how to compute more general structural information by making use of the primitive structures. More information can be found in [5, 7, 28].

Acknowledgment: We acknowledge the support of our research on keyword search by the grant of the Research Grants Council of the Hong Kong SAR, China, No. 419109.

References

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *Proc. 18th Int. Conf. on Data Engineering*, pages 5–16, 2002.

[2] S. Amer-Yahia, P. Case, T. Rölleke, J. Shanmugasundaram, and G. Weikum. Report on the db/ir panel at sigmod 2005. *SIGMOD Record*, 34(4):71–74, 2005.

[3] S. Amer-Yahia and J. Shanmugasundaram. Xml full-text search: Challenges and opportunities. In *Proc. 31st Int. Conf. on Very Large Data Bases*, page 1368, 2005.

[4] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *Proc. 18th Int. Conf. on Data Engineering*, pages 431–440, 2002.

[5] S. Chaudhuri and G. Das. Keyword querying and ranking in databases. *Proc. of the VLDB Endowment*, 2(2):1658–1659, 2009.

[6] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating db and ir technologies: What is the sound of one hand clapping? In *Proc. of CIDR'05*, 2005.

[7] Y. Chen, W. Wang, Z. Liu, and X. Lin. Keyword search on structured and semi-structured data. In *Proc. 2009 ACM SIGMOD Int. Conf. On Management of Data*, pages 1005–1010, 2009.

[8] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[9] B. B. Dalvi, M. Kshirsagar, and S. Sudarshan. Keyword search on external memory data graphs. *Proc. of the VLDB Endowment*, 1(1):1189–1204, 2008.

[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 *Proc. 23rd Int. Conf. on Data Engineering*, pages 836–845, 2007.

[11] S. E. Dreyfus and R. A. Wagner. The steiner problem in graphs. In *Networks*, 1972.

[12] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *Proc. 2008 ACM SIGMOD Int. Conf. On Management of Data*, pages 927–940, 2008.

[13] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *Proc. 2007 ACM SIGMOD Int. Conf. On Management of Data*, pages 305–316, 2007.

[14] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-Style keyword search over relational databases. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 850–861, 2003.

[15] V. Hristidis, H. Hwang, and Y. Papakonstantinou. Authority-based keyword search in databases. *ACM Trans. Database Syst.*, 33(1), 2008.

[16] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 670–681, 2002.

[17] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proc. 31st Int. Conf. on Very Large Data Bases*, pages 505–516, 2005.

[18] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *Proc. 25th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 173–182, 2006.

[19] B. Kimelfeld and Y. Sagiv. New algorithms for computing steiner trees for a fixed number of terminals. In *http://www.cs.huji.ac.il/ bennyk/papers/steiner06.pdf*, 2006.

[20] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7), 1972.

[21] 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 *Proc. 2008 ACM SIGMOD Int. Conf. On Management of Data*, pages 903–914, 2008.

[22] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *Proc. 2006 ACM SIGMOD Int. Conf. On Management of Data*, pages 563–574, 2006.

[23] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *Proc. 2007 ACM SIGMOD Int. Conf. On Management of Data*, pages 115–126, 2007.

[24] A. Markowetz, Y. Yang, and D. Papadias. Keyword search on relational data streams. In *Proc. 2007 ACM SIGMOD Int. Conf. On Management of Data*, pages 605–616, 2007.

[25] L. Qin, J. X. Yu, and L. Chang. Keyword search in databases: The power of rdbms. In *Proc. 2009 ACM SIGMOD Int. Conf. On Management of Data*, pages 681–694, 2009.

[26] L. Qin, J. X. Yu, L. Chang, and Y. Tao. Querying communities in relational databases. In *Proc. 25th Int. Conf. on Data Engineering*, pages 724–735, 2009.

[27] L. Qin, J. X. Yu, L. Chang, and Y. Tao. Scalable keyword search on large data streams. In *Proc. 25th Int. Conf. on Data Engineering*, pages 1199–1202, 2009.

[28] J. X. Yu, L. Qin, and L. Chang. *Keyword Search in Databases*. Morgan & Claypool, 2010.

36th International Conference on Very Large Data Bases (VLDB2010)

13 to 17 September 2010, Grand Copthorne Waterfront Hotel, Singapore,

Website: http://vldb2010.org, **Submission Site:** https://cmt.research.microsoft.com/VLDB2010

VLDB 2010 calls for outstanding research papers as well as proposals for demonstrations. Tutorial and Panel proposals on all topics that will be of particular interest for the community are welcome. VLDB 2010 also strongly encourages the submission of workshop proposals on challenging topics in areas related to the VLDB focus. VLDB 2010 is organized into four tracks.

Core Database Technology Track

The Core Database Technology Track will evaluate papers on technologies intended to be incorporated within the database system itself. The topics of interest to this track include (but are not limited to):

Active Databases
Benchmarking, Performance & Evaluation
Concurrency Control, Recovery and Transaction Management
Data Models and Languages
Database Administration and Manageability
Database Indexing and Search
Databases on Modern Hardware
Embedded and Mobile Databases
Engine-based Views, Replication, and Caching
Fuzzy, Probabilistic, and Approximate Data
Native Semi-Structured Data and XML
Parallel, Distributed, and Grid Databases
Private and Secure Databases
Query Processing and Optimization
Real-Time Databases
Reliable and Robust Databases
Scientific Databases
Spatial, Temporal & Multimedia Databases
Stream Databases

Infrastructure for Information Systems Track

The Information Infrastructure Track covers all aspects of data management not implemented within a conventional database engine. The topics covered by this track include (but are not limited to):

Content Delivery Networks
Data Design, Evolution and Migration
Data Extraction
Data Management in Computational Science
Data Mining
Data Quality and Semantics
Database Services and Applications
Heterogeneous and Federated DBMS (Interoperability)
Information Filtering and Dissemination
Information Integration and Retrieval
Meta-data Management
Middleware Platforms for Data Management
Mobile Data Management
Novel/Advanced Applications
On-Line Analytic Processing
P2P and Networked Data Management
Profile-based Data Management
Provenance Management
Scientific Databases
Sensor Networks
User Interfaces and Visualization
Web Replication and Caching
Web Services and Web Service Composition
XML Middleware Platforms
Social Systems and Recommendations

Industrial Applications and Experience Track

The Industrial, Applications, and Experience Track covers innovative commercial database implementations, novel applications of database technology, and experience in applying recent research advances to practical situations, in any of the following example areas (or, in other areas where data management is important):

Adapting DB Technology to Industrial Settings and Requirements
Application Areas (Government, Finance, Humanities, Telecommunications, Home and Personal Computing, ...)
Bio-Informatics/Life Sciences
Business Process Engineering and Execution Support
Data Management for Developing Countries
Digital Libraries/Document Management
Electronic Commerce
Engineering Information Systems
Enterprise Data Management
Enterprise Resource Planning
Environmental Management
Experiences in Using DB Technology
Geographic Information Systems
Industrial-Strength Systems based on DB Technology
Mobile Computing
Self-Managing Systems
System Design and Implementation using DB Technology

The Experiments and Analyses Track

Database management has been an active area of research for several decades. This special topic aims to meet needs for consolidation of a maturing research area by providing a prestigious forum for in-depth analytical or empirical studies and comparisons of existing techniques. The expected contribution of an Experiments and Analyses (E&A) paper is new, independent, comprehensive and reproducible evaluations and comparisons of existing data management techniques. Thus, the intended contribution of an E&A paper is not a new algorithm or technique but rather further insight into the state-of-the-art by means of careful, systematic, and scientific evaluation. Comparisons of algorithmic techniques must either use best-effort re-implementations based on the original papers, or use existing implementations from the original authors, if publicly available. Authors should discuss and validate substantial new results with authors of the original methods before submission.

IMPORTANT DATES

Research Papers
1 March 2010 (5:00pm GMT, 10:00am PDT) Abstracts submission
9 March 2010 (5:00pm GMT, 10:00am PDT) Paper submission
17 May - 20 May 2010 Author feedback period
12 June 2010 Notification
11 July 2010 Camera-ready paper due

PhD Workshop Papers
10 April 2010 (5:00pm GMT, 10:00am PDT) Paper submission
12 June 2010 Notification
11 July 2010 Camera-ready paper due

Demonstration Proposals
22 March 2010 (5:00pm GMT, 10:00am PDT) Proposal submission
12 June 2010 Notification
11 July 2010 Camera-ready paper due

Workshop Proposals
31 January 2010 (5:00pm GMT, 10:00am PDT) Proposal submission
31 March 2010 Notification

Tutorial proposals
2 April 2010 (5:00pm GMT, 10:00am PDT) Proposal submission
12 June 2010 Notification

Panel Proposals
2 April 2010 (5:00pm GMT, 10:00am PDT) Proposal submission
19 June 2010 Notification

Conference
13 and 17 September 2010 Workshops
14 to 16 September 2010 Main Conference

Call for Papers

The annual ICDE conference addresses research issues in designing, building, managing, and evaluating advanced data-intensive systems and applications. It is a leading forum for researchers, practitioners, developers, and users to explore cutting-edge ideas and to exchange techniques, tools, and experiences. We invite the submission of original research contributions and industrial papers, as well as proposals for workshops, panels, tutorials, and demonstrations.

ICDE 2011 will be held in Hannover, Germany. Hannover is the capital of the federal state of Lower Saxony (Niedersachsen), Germany. Hannover is known as a trade fair city (e.g. CeBIT) but also for their Royal Gardens of Herrenhausen and many other places of interest.

Submission Guidelines

All submissions must be prepared in the IEEE camera-ready format. Research and industrial paper submissions are limited to 12 pages. A paper submitted to ICDE 2011 cannot be under review for any other conference or journal during the entire time it is considered for ICDE 2011, and it must be substantially different from any previously published work (other than posters or short papers with a length of up to 4 pages when converted into the IEEE format).

Submissions are reviewed in a single-blind manner. Authors of research papers will have an opportunity to give concise feedback on preliminary reviews within a one-week time window.

Important Dates

Abstract due: July 16, 2010

Full paper submissions due: July 23, 2010

Notification of authors: Oct. 30, 2010

Final versions due: Nov. 30, 2010

Conference Chairs

General Chairs

Alfons Kemper (TU Munich, Germany)

Wolfgang Nejdl (Leibniz Universität Hannover)

Local Organization and Financial Chair

Thomas Risse (L3S Research Center, Germany)

Program Committee Chairs

Serge Abiteboul (INRIA, France)

Christoph Koch (Cornell University, USA)

Kian-Lee Tan (National University of Singapore)

Industrial Program Chairs

Surajit Chaudhuri (Microsoft, USA)

Volker Markl (TU Berlin, Germany)

Tutorial/Seminars Chairs

Nikos Mamoulis (University of Hong Kong, Hong Kong)

Pierre Senellart (Télécom ParisTech, France)

Workshop Chairs

Kevin S. Beyer (IBM, USA)

Sunil Prabhakar (Purdue University, USA)

Panel Chair

Patrick Valduriez (INRIA, France)

Poster Chairs

Georgia Koutrika (Stanford University, USA)

Heiko Schuldt (University Basel, Switzerland)

Demo Chairs

Kevin C. Chang (University of Illinois, USA)

Dean Jacobs (SAP, Germany)

Proceedings Chair

Klemens Böhm (University Karlsruhe, Germany)

Publicity Chair

Wolf-Tilo Balke (Universität Braunschweig, Germany)

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398