Interactive SQL Query Suggestion: Making Databases User-Friendly

Ju Fan [1], Guoliang Li [2], Lizhu Zhou [3]

Department of Computer Science and Technology, Tsinghua University
Tsinghua University, Beijing 100084, China
[1]fan-j07@mails.tsinghua.edu.cn
{[2]liguoliang,[3]dcszlz}@tsinghua.edu.cn

***Abstract*— SQL is a classical and powerful tool for querying relational databases. However, it is rather hard for inexperienced users to pose SQL queries, as they are required to be proficient in SQL syntax and have a thorough understanding of the underlying schema. To give users gratification, we propose SQLSUGG, an effective and user-friendly keyword-based method to help various users formulate SQL queries. SQLSUGG suggests SQL queries as users type in keywords, and can save users' typing efforts and help users avoid tedious SQL debugging. To achieve high suggestion effectiveness, we propose *queryable templates* to model the structures of SQL queries. We propose a template ranking model to suggest templates relevant to query keywords. We generate SQL queries from each suggested template based on the degree of matchings between keywords and attributes. For efficiency, we propose a progressive algorithm to compute top-*k* templates, and devise an efficient method to generate SQL queries from templates. We have implemented our methods on two real data sets, and the experimental results show that our method achieves high effectiveness and efficiency.**

I. INTRODUCTION

Structured query languages (e.g., SQL) are indispensable and powerful tools for many kinds of users, e.g., advanced searchers, database administrators, and SQL programmers. However, it is hard and tedious for inexperienced users to pose structured queries that satisfy their query intent, since the users are required to be proficient in writing the query languages and have a thorough understanding of the schema. On the other hand, they may encounter comprehension difficulties, formulation problems, and unclear error messages while using SQL. They have to refer to manuals and repeatedly try different SQL queries to obtain expected results [15], if lucky enough.

To address these problems, many assistant tools have been developed to help users formulate structured queries. For example, SQL Assistant[1] is a well-known system which can suggest table names, attribute names, preserved words in SQL syntax, etc. However, these tools still have the following limitations: 1) they cannot support the suggestion of data instances; 2) they require users to manually join multi-tables in an appropriate way. In other words, the SQL Assistant users should be skillful in writing SQL queries based on their information needs.

In order to reduce the burden of posing queries, the database community has started to introduce *keyword search* into relational databases [2,7,9,13]. This search paradigm allows users to pose keyword queries without having to understand the database schema and SQL syntax. Although keyword search may be acceptable for casual users, it is insufficient for the users who have to pose SQL queries, e.g., database administrators and SQL programmers, since keyword search cannot precisely capture users' query intent and may involve irrelevant results. Even worse, keyword search mixes all the answers together, and even if the answers have different structures, keyword search cannot group the answers.

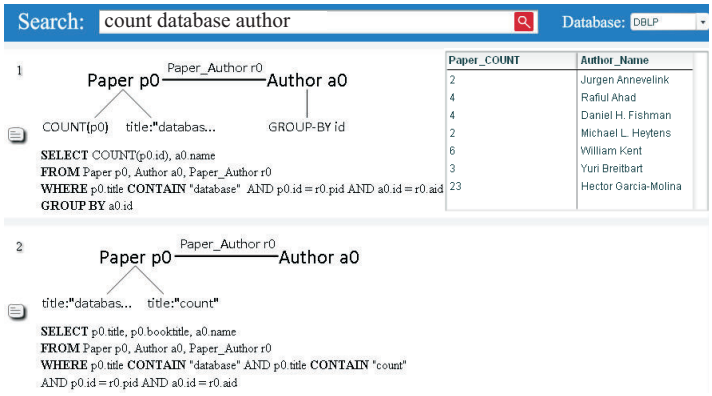


Fig. 1. An SQLSUGG-based system for publication search

To address these limitations, we propose a novel search paradigm, called interactive SQL query SUGGestion (SQLSUGG), which combines the convenience of keyword search and the power of SQL. In our approach, as users type in query keywords, we on-the-fly suggest the top-*k* most relevant SQL queries based on the keywords, and users can select SQL queries to retrieve the corresponding answers. For example, Figure 1 provides a screen shot of an SQLSUGG-based system for publication search. Consider a query "`count database author`", an SQLSUGG user will obtain a ranked list of SQL queries. The first SQL query is to find the number of papers with title containing "database" for each author. SQLSUGG can provide graphical representation and sample results to each suggested SQL query. The user can refine keywords as well as the suggested queries to obtain the desired results interactively.

An SQLSUGG-based method has the following advantages. Firstly, it helps users formulate (even complicated) structured queries based on limited keywords. Therefore, it can not only

[1]http://www.softtreetech.com/isql.htm

TABLE I

AN EXAMPLE DATABASE (JOIN CONDITIONS: PAPER.ID = WRITE.PID AND AUTHOR.ID = WRITE.AID).

(a) PAPER

id	title	booktitle	year
P_1	database ir	tois	2009
P_2	xml name count	tois	2009
P_3	evaluation	database theory	2008
P_4	database ir	database theory	2008
P_5	database ir xml	ir research	2008

(b) AUTHOR

id	name
A_6	lucy
A_7	john ir
A_8	tom
A_9	jim
A_{10}	gracy

(c) WRITE

id	pid	aid	id	pid	aid
W_{11}	P_2	A_6	W_{16}	P_3	A_6
W_{12}	P_1	A_7	W_{17}	P_4	A_7
W_{13}	P_5	A_7	W_{18}	P_4	A_8
W_{14}	P_2	A_9	W_{19}	P_5	A_9
W_{15}	P_3	A_{10}	W_{20}	P_5	A_{10}

reduce the burden of posing queries, but also boost SQL coding productivity significantly. Secondly, SQLSUGG helps users express their query intent more precisely than keyword search, especially for users who pose complex queries. Thirdly, as each SQL query represents a structure of the underlying data, our method inherently groups the answers and helps users browse the answers.

We study the research challenges that naturally arise in the proposed search paradigm. The first challenge is to infer users' query intent, including structures and aggregations, from limited keywords. We propose *queryable templates* ("templates" for short) to model the structures of promising SQL queries. We propose a probabilistic model to measure the relevance between a template and a keyword query for suggesting relevant templates. The second challenge is to generate SQL queries from templates. We generate SQL queries by matching keywords to attributes in templates, and rank the generated SQL queries based on the degree of matchings between keywords and attributes, and query abilities of matched attributes. The third challenge is the search efficiency. We devise a top-k algorithm to suggest templates relevant to keyword queries. We also propose a greedy approximation algorithm to generate SQL queries from templates. We have implemented our methods on two real data sets, and the experimental results show that our method achieves high search efficiency and result quality.

To summarize, we make the following contributions.

1) We propose a template-based framework, which suggests SQL queries as users type in keywords. We first generate templates by analyzing the query ability of the underlying schema and data. Then we on-the-fly suggest relevant templates and generate SQL queries from the suggested templates.
2) We develop effective ranking functions by considering the relevance between keywords and templates, the query abilities of entities and attributes, and the degree of matchings between keywords and attributes.
3) We devise a fast algorithm to progressively suggest top-k relevant templates, and a greedy algorithm to generate SQL queries efficiently.

This paper is organized as follows. The problem formulation and an overview of SQLSUGG are presented in Section II. We introduce our method of suggesting templates in Section III and discuss the method of generating SQL queries from templates in Section IV. Experimental results are provided in Section V and the related work is reviewed in Section VI. Finally, we conclude the paper in Section VII.

II. PROBLEM FORMULATION AND SQLSUGG OVERVIEW

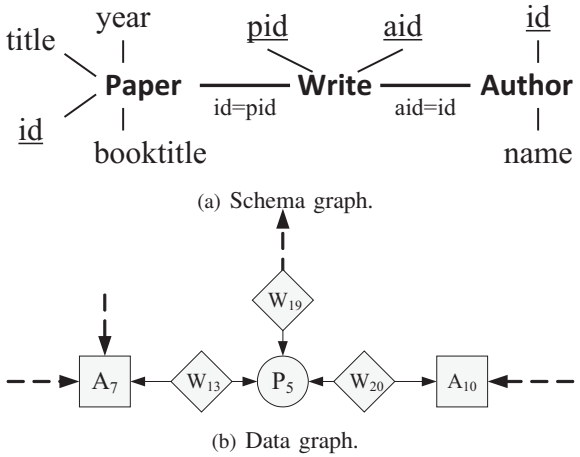
We formulate the problem in Section II-A, and give an overview of our method in Section II-B.

A. Problem Formulation

Data Model: Our work focuses on suggesting SQL queries for a relational database $\mathcal{D}$ with a set of relation tables, $R_1, R_2, \ldots, R_n$, and each table R_i has a set of attributes, $A_1^i, A_2^i, \ldots, A_m^i$. To represent the schema and underlying data of $\mathcal{D}$, we define the schema graph and the data graph respectively.

To capture the foreign key to primary key relationships in the database schema, we define the *schema graph* as an undirected graph $G_S = (V_S, E_S)$ with node set V_S and edge set E_S: 1) each node is either a relation node corresponding to a relation table, or an attribute node corresponding to an attribute; 2) an edge between a relation node and an attribute node represents the membership of the attribute to the relation; 3) an edge between two relation nodes represents the foreign key to primary key relationship between the two relation tables.

Similarly, we define the *data graph* to represent the data instances in the database. The data graph is a directed graph, $G_D = (V_D, E_D)$ with node set V_D and edge set E_D, where nodes in V_D are data instances (i.e., tuples). There exists an edge from node v_1 to node v_2 if their corresponding relation tables have a foreign key to primary key relationship, and the foreign key of v_1 equals to the primary key of v_2.



(a) Schema graph.

(b) Data graph.

Fig. 2. Examples of schema graph and data graph.

Table I provides an example database containing a set of publication records. The database has three relation tables, PAPER, AUTHOR, and WRITE, which are respectively abbreviated to P, A and W in the rest of the paper for simplicity.

Figure 2(a) shows the schema graph of the example database. In the graph, the relation PAPER has four attributes (i.e., id, title, booktitle, and year), and has an edge to another relation WRITE. Figure 2(b) shows the data graph. In this graph, an instance of PAPER (i.e., P_5) is referred by three instances of WRITE (i.e., W_{13}, W_{19}, and W_{20}).

Query Model: We focus on suggesting a ranked list of SQL queries from limited keyword queries. Note that the query keywords can be very flexible that they may refer to either data instances, the meta-data (e.g., names of relation tables or attributes), or aggregate functions (e.g., the function COUNT). Formally, Given a keyword query, $Q = \{k_1, k_2, \ldots, k_{|Q|}\}$ posed by a user, the answer of Q is a list of SQL queries, each of which contains all keywords in its clauses, e.g., the WHERE clause, the FROM clause, the SELECT clause, or the GROUP-BY clause, etc. Since there may be many SQL queries corresponding to a keyword query, we propose to rank SQL queries by their relevance to the keyword query. For example, consider the example database in Table I and a keyword query "`count database author`" . SQLSUGG can suggest two SQL queries as follows.

```
1)  SELECT      COUNT( P.id ), A.name
    FROM        P, W, A
    WHERE       P.title CONTAIN "database" AND
                P.id = W.pid AND A.id = W.aid
    GROUP BY    A.id

2)  SELECT      P.title, P.booktitle, A.name
    FROM        P, W, A
    WHERE       P.title CONTAIN "database" AND
                P.title CONTAIN "count" AND
                P.id = W.pid AND A.id = W.aid
```

where CONTAIN is a user-defined function (UDF) which can be implemented using an inverted index. Observed from the above SQL queries, the first one is to group the number of papers with titles containing "`database`" by authors, and the second one is to find a paper as well as its author such that the title contains the keywords, "`database`" and "`count`" .

Comparison with Existing Methods. Some approaches to keyword search on databases [2,7,9,13] suggest Candidate Networks (CN), each of which corresponds to a SPJ (Selection-Projection-Join) SQL query, from keyword queries. Compared with these approaches, SQLSUGG has the following advantages.

- SQLSUGG cannot only suggest SPJ queries, but also support aggregate functions, which are extensively used by SQL programmers.
- SQLSUGG can group the results by their underlying query structures, rather than mixing all results together.
- SQLSUGG ranks the suggested SQL queries by their relevance to keyword queries.
- SQLSUGG employs fast ranking algorithms to suggest SQL queries efficiently.

B. Overview of SQLSUGG

In this section, we present an overview of SQLSUGG and give an example to show how SQLSUGG works.

Overview: SQL query suggestion is rather challenging, because the keywords may refer to either data instances of different relation tables, relation or attribute names, or aggregate functions. Even worse, users may not only be interested in a single table, but also want to join multiple tables in various ways. Therefore, given a keyword query, there could be a very large amount of possibly-relevant SQL queries. To suggest most relevant SQL queries, we propose a two-step based method: we first suggest query structures, *queryable templates* ("templates" for short), and rank templates by their relevance to the keyword query. Then, we generate SQL queries from each suggested template and rank the queries by the degree of matchings between keywords and attributes in the template. Thus, the SQL queries are actually grouped by their corresponding templates. A user can firstly select a template, and then narrow to SQL queries in this template. This strategy would be more convenient for users to find the desired SQL queries, since all SQL queries are well organized according to their structures. In contrast, mixing SQL queries with different templates together may confuse users. On the other hand, the two-step framework has performance superiority.

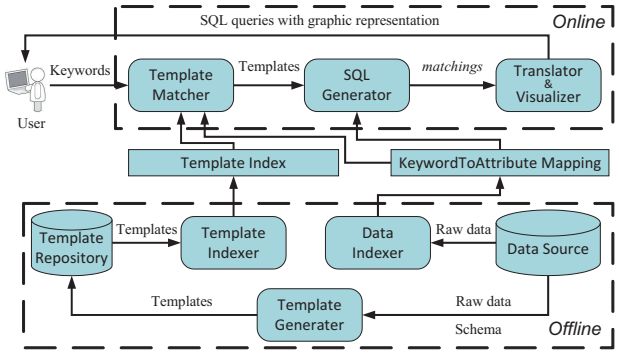


Fig. 3. Architecture of an SQLSUGG-based system.

Figure 3 shows the architecture of an SQLSUGG-based system, which is composed of the offline template indexing and the online SQL query suggestion. In the offline part, templates are generated by the Template Generator from the Data Source, and stored in a Template Repository. Since the Template Generator may generate many templates, especially for complex schemata, a Template Index is built for efficient online template suggestion by a Template Indexer. Moreover, since keywords may refer to data instances, meta-data, functions, etc., a Data Indexer preprocesses the Data Source to construct a KeywordToAttribute Mapping for mapping keywords to attributes. For online suggestion, given a keyword query, the Template Matcher suggests relevant templates that reflect user's query intent. Then, the SQL Generator takes the matched templates as input, and produces matchings between keywords and attributes in the suggested templates to generate SQL queries. Finally, the Translator & Visualizer presents SQL statements with graphical representation. After users obtain the suggested SQL queries, they can either modify the keywords or the suggested SQL statements to refine the query.

Running Example: We give a running example to illustrate

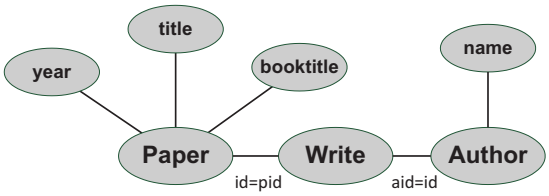


Fig. 4. An example template for finding a paper with its author

how an SQLSUGG-based system suggests SQL queries for a keyword query "`count database author`" . The system first suggests the relevant templates to infer the query structures and ranks them by their relevance. Figure 4 shows one of the suggested templates. We will explain template suggestion in Section III. Then, the system generates SQL queries from each template by matching keywords to attributes. For example, "`count`" and "`database`" can be matched to attribute P.`title`, and "`author`" can be matched to attribute A.`id` (If a keyword refers to the name of a relation, we map the keyword to its primary-key). We present the details of SQL generation from templates in Section IV. Finally, the system translates the matching between keywords and attributes into an SQL statement.

III. QUERYABLE TEMPLATE SUGGESTION

To suggest SQL queries from limited keyword queries, it is very important to infer query structures relevant to the keyword queries. We introduce the *queryable template* to model the query structure in Section III-A. To suggest relevant templates, we propose a template ranking model to measure the relevance between keywords and templates in Section III-B and devise a top-k ranking algorithm for efficient template suggestion in Section III-C.

A. Queryable Template

Query structure is essential to an SQL query, because it specifies which entities are involved in the query and how they are joined together. From the SQL syntax point of view, the query structure includes relation entities in the FROM clause and JOIN conditions in the WHERE clause. Some approaches have been proposed to model the query structure in the database community. A well-known approach is Candidate Network (CN) [8,9,18] for keyword search in relational databases. Another approach is Simple Query Network in the context of posing "aggregate queries" [19]. The approaches model query structures as trees of relation entities, and generate them from the schema graph. However, they are limited to measure the relevance between keywords and the structure, and to analyze the query ability of the structure.

In order to address this problem, we propose the *queryable template* to model the query structure. Conceptually, a template is a group of joined entities with their attributes, and can be taken as the skeleton of SQL queries. Formally, a template is defined as follows.

Definition 1 (Queryable Template): A template is an undirected graph, $G_T(V_T, E_T)$ with the node set V_T and the edge set E_T, where nodes in V_T are:

- *entity nodes:* relation tables, or
- *attribute nodes:* attributes of entities,

and edges in E_T are:

- *membership edges:* edges between an entity node and its attribute nodes, or
- *foreign-key edges:* edges between entity nodes with foreign keys and the entity nodes referred by them.

In particular, templates with only one entity node are called **atomic-templates**. □

For example, Figure 4 provides a template corresponding to the query structure for finding a paper with its author. This template has three entities, i.e., P, A and W, as well as their attributes. The entities are joined according to the foreign-key-to-primary-key relationships. For simplicity, we represent the template as P − W − A.

Template Generation. Given a database, there could be a huge amount of templates that capture various query structures. We design a method to generate them using the schema graph. The basic idea of the method is to *expand* templates to generate new templates. To this end, the algorithm firstly generates atomic-templates, and takes them as bases of template generation. Then, we use an *expansion rule* to generate new templates: A template can be expanded to a new template if it has an entity node that can be connected to a new entity node via a foreign-key edge. For example, consider an atomic template, P. It can be expanded to P−W according to the expansion rule. Since a template may be expanded from more than one template, we eliminate duplicated templates. Furthermore, we examine the relationship between relation tables. For example, since the two relation tables, P and W, have a *1-to-n* relationship, i.e., a instance of W has at most one instance of P. Hence, the template, P−W−P is invalid.

TABLE II
TEMPLATES FOR OUR EXAMPLE DATABASE ($\gamma = 5$).

Size	ID	Template
1	T_1	P
	T_2	A
	T_3	W
2	T_4	P − W
	T_5	A − W
3	T_6	P−W−A
	T_7	W−P−W
	T_8	W−A−W
4	T_9	P−(W,W,W)
	T_{10}	A−(W,W,W)
	T_{11}	W−P−W−A
	T_{12}	W−A−W−P
5	T_{13}	P−W−A−W−P
	T_{14}	A−W−P−W−A
	...	...

Apparently, the above-mentioned expansion rule may lead to a combinatorial explosion of templates. To address this problem, we employ a parameter γ to restrict the maximal size of templates (i.e., the number of entities in a template), which is also used in CN-based keyword-search approaches [8,9]. The motivation here is that templates with too many entities are meaningless, since the corresponding query structures are not of interests to users. Table II provides the templates

generated for our example database under $\gamma = 5$, where P−(W,W,W) represents that P is connected with three W entities. Further, even though we restrict the maximal size of templates, there still are many templates due to complex relationships between tables in schemas. Thus, we devise an efficient top-k template ranking algorithm to avoid exploring the entire space of searching templates (See Section III-C).

B. Template Ranking Model

It is very crucial to rank the templates, since there may be a huge number of templates relevant to the keyword query. Existing CN-based methods [8,9,18] rank candidate networks (which also capture query structures as templates do) by the number of entities involved in them, that is, they prefer the compact structures to the complicated ones. However, the compactness of query structures is limited to capture the relevance between keywords and structures. Consider a keyword query "`database gray`" on a publication data set. Although the keyword "`gray`" may occur in the title of a paper, it is conceptually more likely to refer to an author. Thus, the best template could be P−W−A rather than P, although the latter is more compact. Moreover, entities in a database do not have the same importance [20], and users may be more interested in templates with important entities. However, existing methods are limited to analyze the entity importance.

To address the limitations of existing methods, we propose a probabilistic model to rank templates for a given keyword query[2]. The basic idea of our model is to prefer the templates with important entities that are more relevant to the keyword query. Therefore, we take into account the following two factors: 1) the relevance between keywords and entities in a template, and 2) the importance of an entity, called *query ability*. Formally, we present the template ranking model as follows. Consider a keyword query, $Q = \{k_1, k_2, \ldots, k_{|Q|}\}$ and a template T which has entities, $\{R_1, R_2, \ldots, R_{|T|}\}$. The task of our model is to estimate a joint probability $P(Q, T)$ as relevance between the query Q and the template T.

To estimate the probability $P(Q, T)$, we first assume that the query keywords are conditionally independent to each other w.r.t. the template, and thus reduce the problem to estimate the relevance between each keyword and the template, i.e.,

$$P(Q,T) = P(T)P(Q \mid T) = P(T) \sum_{k} P(k \mid T), \qquad (1)$$

where $P(T)$ is the probability that users use T to express their query intent. It can be either determined by the database administrator or be learnt from the query logs. In our experiments, we assume that $P(T)$ follows a uniform distribution. Here, we focus on the estimation of the keyword relevance to the template, $P(k \mid T)$ by considering every entity in the template, and thus obtain

$$P(k \mid T) = \sum_{R \in T} P(k \mid R)P(R \mid T), \qquad (2)$$

where $P(k \mid R)$ is the keyword relevance to entity R, and $P(R \mid T)$ is the query ability of entity R in terms of T. Clearly, we use the two probabilities, $P(k \mid R)$ and $P(R \mid T)$ to represent the above-mentioned factors for measuring relevance between keywords and templates. Next, we introduce the estimation methods for the two probabilities.

Estimation of keyword relevance to entity, $P(k \mid R)$. We use the smoothing methods in language model to estimate $P(k \mid R)$. Many approaches in language model (see [21] for a good survey) have been proposed to estimate the relevance between a keyword and a document, i.e., $P(k \mid D)$. Borrowing the idea from them, we treat the entity R as a *virtual* document (For simplicity, we use R to represent both an entity and a virtual document in this section) and estimate the likelihood of sampling keyword k from the "document" using the Jelinek-Mercer smoothing technique, i.e.,

$$P(k \mid R) = (1-\lambda)\frac{\texttt{count}(k,R)}{\mid R \mid} + \lambda\frac{\texttt{count}(k,\mathcal{D})}{\mid \mathcal{D} \mid} \qquad (3)$$

where $\texttt{count}(k, R)$ is the frequency of k in R, $\mid R \mid$ is the number of keywords in R, $\texttt{count}(k, \mathcal{D})$ is the number of entities having k in the database $\mathcal{D}$, and $\mid \mathcal{D} \mid$ is the number of entities in $\mathcal{D}$. Thus, $P(k \mid R)$ is estimated by the relative frequency of k in R, interpolated with the (normalized) number of entities having k. In particular, the virtual document R does not only have words in tuples, but also contains words in meta-data (e.g., attribute names, entity names, etc.). We use the number of tuples to estimate $\texttt{count}(k, R)$, if k refers to the meta-data of R. For example, consider a keyword "`paper`". Since it is the name of entity PAPER, we incorporate it into the virtual document, and set its frequency as 5 (number of tuples in PAPER). Similarly, keywords referring attribute names, e.g., "`title`", "`year`", etc., are also taken into account, and their frequencies are estimated with number of tuples having the corresponding attributes.

Estimation of entity query ability, $P(R \mid T)$. Conceptually, $P(R \mid T)$ is the comparative importance of entity R in template T. To estimate $P(R \mid T)$, we propose to calculate the *PageRank* scores [3] for tuples over the data graph (Section II-A), and aggregate the scores of an entity as $P(R \mid T)$. PageRank, which is a classic algorithm to measure comparative importance of nodes in a graph structure, can be calculated as follows.

$$PR(n_i) = \frac{1-d}{N} + d \cdot \sum_{n_j \in I(n_i)} \frac{PR(n_j)}{O(n_j)}, \qquad (4)$$

where $I(n_i)$ is the set of nodes that link to n_i, $O(n_j)$ is the number of outgoing links on the node n_j, N is the number of nodes in the data graph, and d is a *damping factor*. After obtaining the converged PageRank score for each tuple, we average the scores of tuples of the same entity, and normalize the scores between 0 to 1. For example, the scores of tuples

[2]We have tried other ranking functions, e.g., template query abilities, template sizes, TF-IDF scores, etc. We have compared them and found that the ranking model in our paper is the best. However, we cannot include this comparison in the paper due to the space limitation.

in the relation PAPER are 0.05, 0.06, 0.08, 0.06 and 0.06. We average the scores to obtain the importance of PAPER, 0.065. Similarly, the importance of AUTHOR is 0.065, and that of WRITE is 0.035. Then, we normalize the scores and obtain that the query abilities of P, W and A are 0.39, 0.22 and 0.39 respectively.

Take a keyword query "`count paper john`" and the template T_6 in Table II as an example. According to Equation (3), given that $\lambda = 0.3$, we have $P(\texttt{paper} \mid \mathsf{P}) = 0.16$, and $P(\texttt{paper} \mid \mathsf{A}) = P(\texttt{paper} \mid \mathsf{W}) = 0$. On the other hand, $P(\mathsf{P} \mid T_6) = 0.39$ according to query ability of the entity. Hence, we obtain that $P(\texttt{paper} \mid T_6) = 0.06$. Similarly, $P(\texttt{john} \mid T_6) = 0.05$ and $P(\texttt{count} \mid T_6) = 0.04$. Then, we can combine the above-mentioned probabilities according to Equation (1) to the get ranking score of T_6. Next, we discuss how to deal with a template, say T', where "`count`" does not occur in either tuples or the meta-data. In template suggestion, T_6 may be more relevant if the scores of "`paper john`" to T_6 and T' are the same, since "`count`" may refer to either an aggregate function or data values in T_6. In SQL suggestion, we generate aggregation queries and non-aggregation queries in template T_6, and only aggregation queries in T' (See details in Section IV). Thus, users can select aggregation queries in template T', although "`count`" does not occur in the template.

C. Algorithm for Suggesting Top-k Templates

In this section, we study the problem of suggesting top-k templates for keyword queries. A straightforward way to address this problem is to calculate the ranking score for every template according to our template ranking model. Unfortunately, since the number of templates is exponential, this approach becomes impractical for real-world databases. Therefore, an efficient top-k ranking algorithm is rather necessary to avoid exploring all possible templates. To address this problem, we devise a threshold algorithm (TA) [6] to compute top-k templates efficiently.

The basic idea of our algorithm is to scan multiple lists that present different rankings of templates for an entity, and aggregate scores of multiple lists to obtain $P(Q,T)$ for each template. For early termination, the algorithm maintains an upper bound for the scores of all unscanned templates. If there are k scanned templates whose scores are larger than the upper bound, the top-k templates have been found. Formally, we can rewrite the ranking function in Equations (1) and (2) as follows.

$$P(Q,T) = \sum_{R \in \mathcal{D}} \alpha_R \cdot P(R \mid T), \tag{5}$$

where $\alpha_R = P(T) \cdot \sum_{k \in Q} P(k \mid R)$ (See Section III-B). If α_R is not equal to 0, we say that the corresponding entity R is associated with the keyword query Q. Equation (5) illustrates that the ranking score ($P(Q,T)$) is an aggregation of $P(R \mid T)$ for every entity R in the database $\mathcal{D}$ associated with the keyword query. The aggregation function is monotonous, that is, as $P(R \mid T)$ increases, the score $P(Q,T)$ increases accordingly. Thus, our TA-based algorithm scans multiple lists corresponding to probabilities of $P(R \mid T)$, which are sorted by $P(R \mid T)$ in descending order, and produces top-k templates with highest scores. Next, we introduce indexing schemes which facilitate fast access to $P(R \mid T)$, and explain our algorithm in details.

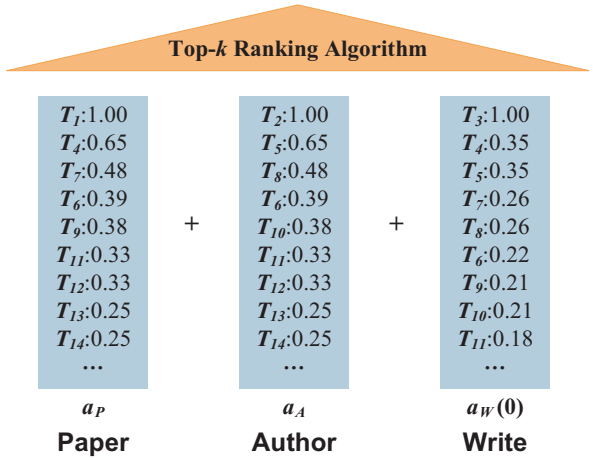


Fig. 5. Top-k ranking algorithm for the query, "`count database author`"

Indexing. We exploit two indexes for fast access to the probability, $P(R \mid T)$ of an entity R in a template T, leading to an efficient calculation of $P(Q,T)$ in Equation (5). We present the two indexes as follows.

- Inverted Index INV. The index maps an entity R to all templates containing it, i.e., $\mathsf{INV} : R \rightarrow T$. In our example, consider an entity P, the inverted index returns the templates, T_1, T_4, T_6, etc. in Table II. We sort the templates in the index by $P(R \mid T)$ in descending order.
- Forward Index FWD. The index maps a template to all entities contained in the template, i.e., $\mathsf{FWD} : T \rightarrow R$. In our example, consider a template T_6, the index returns P, A and W.

In addition, for calculating $\alpha_R = P(T) \cdot \sum_{k \in Q} P(k \mid R)$, we maintain $P(T)$ for each template and $P(k \mid R)$ for each keyword-entity pair. Therefore, with the above-mentioned indexes, we can efficiently rank templates based on the following threshold-based algorithm.

Threshold-based top-k algorithm. The algorithm uses the inverted index INV to construct multiple lists that present different rankings of templates in terms of various entities, and then scans the lists. In each list, when a new template becomes seen, we calculate its ranking score, $P(Q,T)$, by aggregating its partial scores in other lists according to the ranking function. Here, we exploit the forward index FWD to enable random access to the lists, which means that FWD is used to find partial scores, i.e., $P(R \mid T)$, in other lists for a specific template. An upper bound $\mathcal{B}$ is maintained for overall scores of unseen templates, and $\mathcal{B}$ is calculated by applying the ranking function to the last seen template in every list. In addition, we update $\mathcal{B}$ to represent the new upper bound of unseen templates. If the ranking score of a template is larger than or equal to $\mathcal{B}$, it means that its score is not smaller than any unseen templates. Hence, we can insert the template to a result set R, which uses a *heap* structure to maintain the

order of templates. If the size of R is equal to k, we can terminate the algorithm, since the top-k templates have been found. Furthermore, we guarantee that the suggested templates should have all entities associated with the keywords (such that $\alpha_R \neq 0$) in order to prefer the templates that have more entity information.

Figure 5 provides an example for the keyword query, "`count database author`" . Since the probabilities $P(k \mid \mathsf{W})$ for these three keywords are 0, we have that $\alpha_{\mathsf{W}} = 0$. Thus, our ranking algorithm takes the two lists corresponding to the entities, P and A respectively as input. At the first step, the algorithm scans the first template in every list and uses the seen templates, i.e., T_1 and T_2, to calculate the upper bound $\mathcal{B} = \alpha_{\mathsf{P}} + \alpha_{\mathsf{A}}$. Then the algorithm calculates the scores of the seen templates. Since neither T_1 nor T_2 covers all entities associated with the query keywords, i.e., P and A, their scores are 0. Next, the algorithm scans the following templates in every list and calculates $\mathcal{B}$. Consider the case of T_6. The upper bound of the corresponding step is $\mathcal{B} = 0.39 \cdot \alpha_{\mathsf{P}} + 0.39 \cdot \alpha_A$, which is equal to the score of T_6. Hence, we can insert T_6 into the result set and suggest the top-1 template for the query.

IV. SQL Generation From Templates

Since a template is only the *skeleton* of SQL queries, we further propose a method to generate SQL queries from the suggested templates in this section. We introduce an SQL generation model in Section IV-A and design a generation algorithm in Section IV-B.

A. SQL Generation Model

The essence of SQL generation from templates is to *match* query keywords to most-relevant attributes in the templates. This step is indispensable for SQL suggestion from keyword queries, because users often use keywords to refer to specific attributes rather than general templates. However, existing CN-based methods [8,9,18] are limited to match keywords to attributes. Instead, they use SQL queries corresponding to candidate networks to retrieve records from the underlying database and exploit different functions to rank the records. Unfortunately, they mix records with different keyword-to-attribute matchings together. For example, users may want to use the keyword "`database`" to refer to the title of a paper. However, the results returned by existing methods may have records with either title or booktitle containing "`database`" . Therefore, SQL generation from templates by matching keywords to attributes can produce more accurate result records, and help users express refined query intent.

Matching keywords to attributes is very difficult due to combinatorial explosion of mappings between keywords and attributes. Suppose a keyword can be mapped to n attributes on average. This leads to n^m possible keyword-to-attribute matchings for m keywords. Therefore, it is quite necessary to rank the matchings in order to generate SQL queries which are interested by most users. To address this challenge, we propose an SQL generation model by considering two factors: 1) the degree of a mapping between a keyword and an attribute, and 2) the query abilities of the mapped attributes, that is, we prefer an SQL query if query keywords are matched to the more related and more queryable attributes. Formally, we present the model as follows. Consider a keyword query $Q = \{k_1, k_2, \ldots, k_{|Q|}\}$ and a set of attributes $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ in a template T. Let $\mathcal{F} = \{M_1, M_2, \ldots, M_{|\mathcal{F}|}\}$ be a set of mappings, where each $M \in \mathcal{F}$ is a set containing a keyword k mapped to an attribute A. The objective of SQL generation model is to produce a matching between keywords and attributes, $\mathcal{M} \subseteq \mathcal{F}$, which covers all of Q, i.e., $\cup_{M \in \mathcal{M}} M = Q$. We measure the matching score S as follows.

$$S(\mathcal{M}) = \sum_{M \in \mathcal{M}} I(A) \cdot \rho_t(k, A), \tag{6}$$

where $k \in M$ and A are respectively the keyword and the attribute corresponding to the mapping M. $I(A)$ is the query ability of A, and $\rho_t(k, A)$ is the degree of mapping between k and A with a type t (We will explain t below).

Equation (6) shows that the model takes all possible mappings between keywords and attributes as input, and aims to find a matching, i.e., a set of mappings that covers all keywords with a best matching score. Next, we explain $I(A)$ and $\rho_t(k, A)$ in detail.

Attribute Query Ability, $I(A)$. We measure the query ability of attribute A by considering the query ability of its entity R (i.e., $P(R \mid T)$ in Section III-B) and the importance of A in the entity R (denoted as $P(A \mid R)$), i.e.,

$$I(A) = P(A \mid R) \cdot P(R \mid T) \tag{7}$$

While $P(R \mid T)$ has been investigated in Section III-B, we focus on $P(A \mid R)$ in this section. We employ the *entropy* [20] to estimate the probability. Let $V = \{v_1, v_2, \ldots, v_n\}$ denote distinct values of A and let f_i denote the relative frequency that A has each value v_i. The entropy of A, $E(A)$, is calculated as follows.

$$E(A) = -\sum_{i=1}^{n} f_i \cdot \log f_i \tag{8}$$

Then we normalize these entropies between 0 to 1, in order to incorporate it into our probabilistic model for estimating $P(A \mid R)$. Consider the attribute, `title`, in our example. Its entropy $E(\texttt{title}) = -(\frac{2}{5} \cdot \log \frac{2}{5} + 3 \cdot \frac{1}{5} \cdot \log \frac{1}{5}) = 1.33$. Similarly, $E(\texttt{booktitle}) = 1.05$, $E(\texttt{year}) = 0.67$, and $E(\texttt{id}) = 1.61$. Therefore, $P(\texttt{title} \mid \mathsf{Paper}) = 1.33/(1.33 + 1.05 + 0.67 + 1.61) = 0.29$.

Degree of A Mapping, $\rho_t(k, A)$. We suppose that mappings between keywords and attributes are of various types, each of which represents a specific usage of keywords. SQLSugg considers three types of mappings:

- *selection* (denoted as σ): keyword k refers to data instances of attribute A. We use the relative frequency that k refers to A to estimate $\rho_\sigma(k, A)$. For example, consider a keyword "`database`" and an attribute `title`. Since the keyword occurs 3 times in attribute `title`, and 2 times in attribute `booktitle`, its $\rho_\sigma(\texttt{database}, \texttt{title}) = 0.6$.

- *projection* (denoted as π): keyword k refers to the name of attribute A. To estimate $\rho_\pi(k, A)$, we set that $\rho_\pi = 1$ if k is the name of A, and $\rho_\pi = 0$ otherwise.
- *aggregation* (denoted as ϕ): keyword k refers to aggregate functions of attribute A. We maintain a list of keywords related to aggregate functions (e.g., "count", "maximal", etc.) and set $\rho_\phi = 1$ if k is contained in the list (otherwise, $\rho_\phi = 0$).

After finding a matching $\mathcal{M}$, we determine which attributes can be used to group the results (i.e., the GROUP BY statement in SQL syntax), if there are keywords mapped to aggregate functions. We assume that users identify the grouping attributes explicitly by keywords, and thus take keywords mapped to attributes with π type as candidates. Then we calculate the grouping abilities of the candidate attributes according to Equation (9). Finally, we take the attributes with highest grouping scores as grouping attributes.

$$\mathcal{G}(A) = 1 - \frac{\#\ distinct\ values}{\#\ values} \tag{9}$$

B. Best SQL Query Generation

In this section, we focus on the algorithm of generating the best (top-1) SQL query from a template. The key challenge here is how to find a matching $\mathcal{M} \subseteq \mathcal{F}$ that covers all keywords. As mentioned above, if a keyword can be mapped to n attributes on average, we have n^m mappings for m keywords, i.e., $| \mathcal{F} | = n^m$, leading to a combinatorial explosion. Even worse, the number of subsets of $\mathcal{F}$ covering all keywords may be exponential to $| \mathcal{F} |$. Our solution is to formulate this problem as a *weighted set-covering problem*, which is presented as follows.

Definition 2 (Weighted Set-Covering Problem (WSC)): An instance, $(X, \mathcal{F}, c)$, of the weighted set-covering problem consists of a finite set X, a family $\mathcal{F}$ of subsets of X, and a cost function $c : \mathcal{F} \to \mathbb{R}^+$. The problem is to find a cover $C \subseteq \mathcal{F}$ that has all of X and minimizes the cost, i.e.,

$$\begin{aligned} \mathsf{min} \quad & \sum_{S \in C} c(S) \\ \mathsf{st.} \quad & X = \cup_{S \in C} S \end{aligned}$$

Next, we give Lemma 1 to show the equivalence between our problem and WSC.

Lemma 1: Generating the best SQL query from a template is equivalent to a weighted set covering problem. □

Proof. We formally represent the problem of generating the best SQL query as the following optimization problem, $\mathsf{max}\ S(\mathcal{M}) = \sum_M I(A) \cdot \rho_t(k, A)$, $\mathsf{st.}\ Q = \cup_{M \in \mathcal{M}} M$. The elements in the two problems can be mapped in the following way: $\{X \leftrightarrow Q, C \leftrightarrow \mathcal{M}, c(M) \leftrightarrow 1 - I(A) \cdot \rho_t(k, A)\}$. In addition, their optimization objectives are equivalent, i.e., $\mathsf{min} \sum_M c(M) = \mathsf{max} \sum_M I(A) \cdot \rho_t(k, A)$. Thus we prove the lemma. □

Since the weighted set-covering problem has been proven to be NP-complete [16], we devise a greedy approximation algorithm [5] in Figure 6, and provide a logarithmic approximation ratio. Because the logarithmic function grows slowly, the algorithm can produce useful results. The algorithm takes the keyword set Q and a set of mappings $\mathcal{F}$ as input and a best matching $\mathcal{M}$ as output. We first initialize an empty matching $\mathcal{M}$ and choose mappings in $\mathcal{F}$ iteratively (lines 4-6). At each iteration, we choose an $M \in \mathcal{F}$ to minimize the following heuristic price function $price(M)$ (line 5).

Algorithm 1: Generate the best matching $\mathcal{M}$.

Input: A keyword set Q, a set of mappings $\mathcal{F}$
Output: The best matching $\mathcal{M}$

```
1 begin
2     Initialize M ← Φ ;
3     Define f(M) ≐ ∪_{M∈M} M ;
4     while f(M) ≠ Q do
5         Choose M ∈ F to minimize the following price:
              price(M) = (1 − I(A) · ρ_t(k, A)) / | M − f(M) |,
          where I(A) · ρ_t(k, A) is the mapping score ;
6         M ← M ∪ {M} ;
7     Return M ;
8 end
```

Fig. 6. An approximation algorithm for generating the best SQL query.

$$price(M) = \frac{1 - I(A) \cdot \rho_t(k, A)}{| M - f(\mathcal{M}) |}, \tag{10}$$

where $I(A) \cdot \rho_t(k, A)$ is the mapping score of M and $f(\mathcal{M}) \doteq \cup_{M \in \mathcal{M}} M$. We insert M into $\mathcal{M}$ (line 6). If the matching $\mathcal{M}$ covers all keywords in Q, we can terminate the loop and output $\mathcal{M}$ as the result. The greedy approximation algorithm is a polynomial-time $\rho(n)$-approximation algorithm, where $\rho(n) = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \leqslant \ln(n) + 1$ [5].

TABLE III
THE SET OF MAPPINGS $\mathcal{F}$ FOR THE EXAMPLE QUERY.

ID	Keyword k	Attribute A	Type t	Score, $I(A) \cdot \rho_t(k, A)$
...	...	...	...	...
M_1	database	P.`title`	σ	0.068
M_2	database	P.`booktitle`	σ	0.035
M_3	author	A.`id`	π	0.195
M_4	count	P.`id`	ϕ	0.133
M_5	count	P.`title`	ϕ	0.113
...	...	...	...	...

To construct the mapping set $\mathcal{F}$, we construct a keyword-to-attribute mapping index and maintain the corresponding score, $I(A) \cdot \rho_t(k, A)$ (see Section IV-A). Take the keyword query "`count database author`" and the template P – W – A as an example. Table III provides the keyword-to-attribute mapping index corresponding to the three keywords. Our greedy approximation algorithm can produce a best matching $\mathcal{M} = \{M1, M3, M4\}$. Then, since the matching has a M_4 which is of the ϕ type, we further select an attribute with the π type as the grouping attribute, i.e., M_3.

Finally, we can translate the matching mentioned above into the following SQL statement, which is used to find the number

of papers with title containing "database" and group the numbers by authors.

```
SELECT    COUNT( Paper.id ), Author.name
FROM      Paper, Write, Author
WHERE     Paper.title CONTAIN "database" AND
          Paper.id = Write.pid AND
          Author.id = Write.aid
GROUP BY  Author.id
```

Extension. We can extend the algorithm in Figure 6, and devise an approximation algorithm for generating top-k SQL queries according to our scoring function in Equation (6). The top-k algorithm also iteratively chooses mappings in $\mathcal{F}$ according to the price function, and examines whether a matching that covers all keywords has been found. However, different to the algorithm in Figure 6, when a matching is found, the top-k algorithm backtracks by removing a mapping M with lowest price, and try other mappings in $\mathcal{F}$ whose score is lower than $price(M)$. When k matchings have been found, the algorithm terminates.

V. Experiments

We have implemented our methods and conducted extensive experiments on two real data sets to evaluate the effectiveness and efficiency of our proposed methods. We employed a CN-based keyword-search algorithm DISCOVER-II as baseline. All the programs were implemented in JAVA and all the experiments were run on the Ubuntu machine with an Intel Core 2 Quad X5450 3.00GHz processor and 4 GB memory.

A. Experiment Setup

Data Sets: 1) DBLP[3]. It contains more than one million publication records. The schema and the number of rows in each table of DBLP are listed in Table IV. 2) DBLIFE [4]. It contains activity information for more than 500, 000 people in the database community. The schema and numbers of rows in each table of DBLIFE are listed in Table V.

Query Sets: We used 10 keyword queries for the each data set, as shown in Table VI and Table VII. The average lengths of the two query sets are 2.8 and 3.2 respectively.

Implementation of the existing methods: We used the source code of DISCOVER-II [8] as the baseline. DISCOVER-II generates candidate networks (i.e., CNs) from keywords, which is similar to our template generation, because both methods infer query structures from keywords. Thus, we examined the performance of template generation of SQLSUGG, compared with the CN generation of DISCOVER-II in Section V-B.1. In addition, DISCOVER-II retrieves all records containing the keywords ranked by a ranking function, rather than matching keywords to attributes. Thus, we compared the effectiveness of record retrieval of both methods to examine if users can find better records using SQLSUGG in Section V-B.3.

B. Evaluation of Effectiveness

To evaluate the effectiveness, we asked six experts in SQL syntax to judge whether the suggested SQL queries are relevant to corresponding keyword queries. We conducted some experiments to evaluate the effectiveness. In these experiments, the experts were presented with keyword queries in Table VI and Table VII, as well as SQL queries or the retrieved records. They labeled their relevance in a *blind-test* manner. We examined the performance of two important techniques of our approach, i.e. template suggestion in Section V-B.1 and SQL generation in Section V-B.2. The former examines if users satisfy the suggested query structures, and the latter examines if the generated SQL queries in each template are useful. In addition, we also examined the effectiveness of record retrieval of our method to investigate if users satisfy the records retrieved by the suggested SQL queries.

1) Effectiveness of Template Suggestion: We first investigated the effectiveness of template suggestion, and compared our method with DISCOVER-II. The experts labeled whether the templates (or candidate networks) suggested by SQLSUGG and DISCOVER-II were relevant to the corresponding keyword queries. We qualified the effectiveness through standard measures of precision and recall[4].

Figures 7(a) and 8(a) provide the experiment results on the

TABLE IV
THE SCHEMA OF THE DBLP DATA SET

Table	Attributes				# rows
Paper	ID	Title	BookTitle	Year	1109727
Author	ID	Name			653980
Paper-Author	AuthorID	PaperID			2766266

TABLE V
THE SCHEMA OF DBLIFE DATA SET

Table	Attributes					# rows
Person	ID	Name	Homepage	Title	...	68459
Org	ID	Name				163
Pubs	ID	Title	BookTitle	Year	...	108970
Topic	ID	Name				736
Conf	ID	Name				170
ServeConf	PID	CID				3591
GiveConfTalk	PID	CID				131
GiveTutorial	PID	CID				132
GiveOrgTalk	PID	OID				913
RelatedOrg	PID	OID				2436
RelatedTopic	PID	TID				114196
WritePub	PID	PUBID				328410
CoAuthor	PID1	PID2				56370
RelatedPeople	PID1	PID2				115436

TABLE VI
QUERIES FOR THE DBLP DATA SET.

IDs	Queries	IDs	Queries
Q_1	keyword search icde	Q_6	count paper icde
Q_2	database gray	Q_7	count paper jiawei
Q_3	li wang data	Q_8	count author mining
Q_4	paper icde 2005	Q_9	max year gray
Q_5	sequence itemset	Q_{10}	count paper author

TABLE VII
QUERIES FOR THE DBLIFE DATA SET.

IDs	Queries	IDs	Queries
Q_1	jim gray	Q_6	count person stanford
Q_2	database jim gray	Q_7	count conf sigmod jim gray
Q_3	person database	Q_8	max year jim gray
Q_4	topic jim gray	Q_9	count conf person
Q_5	jim gray alexander szalay	Q_{10}	count person topic

[3]http://dblp.uni-trier.de/xml/

[4]We use all relevant SQL queries from the two approaches as the whole relevant query set to compute the recall.

DBLP and DBLIFE data sets respectively. Our method outperforms DISCOVER-II significantly, especially on the DBLIFE data set. For example, the precision of our method is much better than that of DISCOVER-II at each rate of recall in Figure 8(a). The improvement of our method is due to the following reasons. Firstly, compared with DISCOVER-II, our method allows users to search the meta-data (i.e., names of relation tables, or attributes), while DISCOVER-II only supports full-text search. Thus, SQLSUGG can suggest more relevant templates than DISCOVER-II and improve the recall. For example, consider Q_9 in Table VII. Since DISCOVER-II only considers the full-text search, it can only suggest a template, PUBS. In contrast, SQLSUGG can suggest more templates, e.g., PERSON – SERVECONF – CONF, PERSON – GIVECONFTALK – CONF, PERSON – GIVETUTORIAL – CONF, etc., which are more relevant to Q_9 than PUBS.

In addition, even though we extend DISCOVER-II by considering the meta-data, SQLSUGG still has effectiveness advantages. Observed from Figure 7(a), for the DBLP data set with a very simple schema, both methods can provide nearly all relevant templates. However, our method achieves better ranking, as illustrated from precision scores at each recall. An important reason that SQLSUGG outperforms DISCOVER-II is that SQLSUGG exploits better models to measure the keyword-to-template relevance, while DISCOVER-II simply ranks CNs by their sizes.

2) Effectiveness of SQL Generation: Compared with DISCOVER-II, SQLSUGG matches query keywords to attributes, and can predict the conditions in the WHERE clause, projections and aggregations in the SELECT clause. In this section, we conducted experiments to evaluate the effectiveness of SQL generation as follows. For each suggested SQL query, the experts labeled 1, or 0, which indicates whether the matchings between keywords and attributes were precise. Then we exploited the average precision as the evaluation metric.

Figures 7(b) and 8(b) provide the experiment results on the two data sets. Our method achieves high precision for SQL generation. For example, the precision scores of all query keywords in the two data sets are above 80%. Next, we give an example to illustrate the effectiveness of our methods. Consider the query keyword Q_2 in Table VII for the DBLIFE data set. We list the top-3 templates along with the conditions of the top-1 SQL query in each template in Table VIII. From the table, we can see that the keywords "`Jim`" and "`Gray`" are matched to a person name, and "`database`" is matched to either person homepage, topic name, or publication title in the templates.

SQLSUGG can also match keywords to aggregate functions, and produce SQL queries with aggregate functions. We use the Q_{10} in Table VI to illustrate the effectiveness. We list the top-2 templates along with the top-1 SQL query in each template in table IX. From the table, we can see that SQLSUGG suggested two templates for this query: for the template P, it aggregated the relation table PAPER; for the template P – PA – A, SQLSUGG aggregated the relation table PAPER and used the table AUTHOR to group the number of papers.

TABLE VIII

SUGGESTION RESULTS FOR A KEYWORD QUERY, "`database Jim Gray`" ON THE DBLIFE DATA SET (WHERE P, T, R, AND W DENOTE PERSON, TOPIC, RELATEDTOPIC, AND WRITEPUB RESPECTIVELY).

Templates	Conditions in WHERE clause
P	P.name CONTAIN "Jim"
	P.name CONTAIN "Gray"
	P.homepage CONTAIN "database"
P – R – T	P.name CONTAIN "Jim"
	P.name CONTAIN "Gray"
	T.name CONTAIN "database"
P – W – PUBS	P.name CONTAIN "Jim"
	P.name CONTAIN "Gray"
	PUBS.title CONTAIN "database"

TABLE IX

SUGGESTION RESULTS FOR A KEYWORD QUERY, "`count paper author`" ON THE DBLP DATA SET (WHERE P, A, AND PA DENOTE PAPER, AUTHOR, AND PAPER-AUTHOR RESPECTIVELY.)

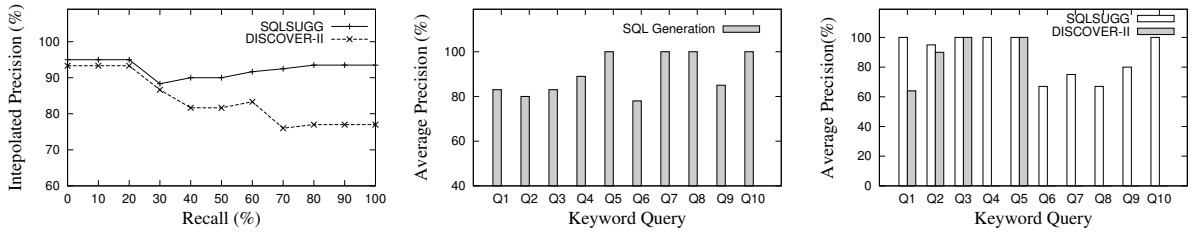
Templates	Aggregation	Grouping
P	P.id	-
P – PA – A	P.id	A.id

The high-effectiveness of SQL generation is due to our SQL generation model (Section IV-A). We exploit attribute importance and degree of mappings between keywords and attributes to measure the matching score. The experimental results show that the scoring scheme is effective.

3) Effectiveness of Record Retrieval: Since SQLSUGG can also help users retrieve desired records from the underlying database, we conducted an experiment to compare the effectiveness of record retrieval with DISCOVER-II. For DISCOVER-II, since it ranks records from different candidate networks (i.e., query structures) according to a ranking function [8], we took at most top-20 records as the result for each keyword query. For SQLSUGG, since it groups records by their corresponding SQL queries, we sent to the underlying database the best SQL query of each suggested template, each of which returned at most top-5 records. Then, we inserted the all records from the SQL queries into a ranked list, and took at most top-20 records as the result for the keyword query. We asked the experts to label 1 or 0 to each record, which indicates whether the record was relevant to the keyword query. We exploited the precision as a metric to evaluate the effectiveness[5].

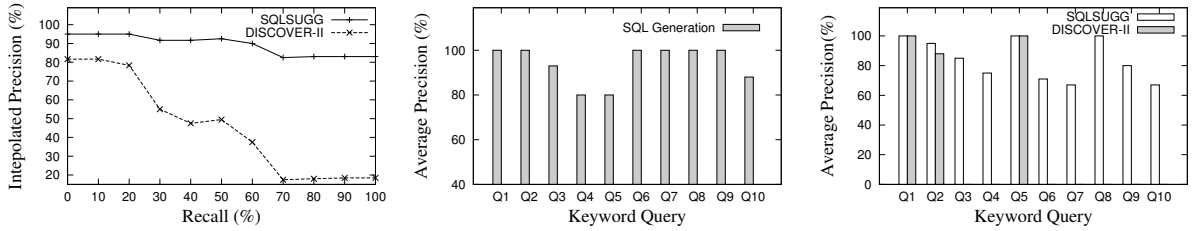
Figures 7(c) and 8(c) show the experimental results on the two data sets. From these figures, we can see that SQLSUGG performs significantly better when query keywords are referred to meta-data or aggregate functions, while DISCOVER-II cannot return relevant records because it cannot find tuples containing the keywords such as "`count`", "`paper`", etc. For example, in Figure 8(c), there are no relevant record returned by DISCOVER-II for the queries Q_4 and Q_6 - Q_{10}.

[5]As there is no standard benchmark, it is hard to find all relevant results and compute recalls. To address this issue, we retrieved top-20 tuples. In this case, the ratio of the recalls of the two systems and that of the precisions of the two systems are the same, i.e., precision and recall are equivalent. Thus we only compared the precision.



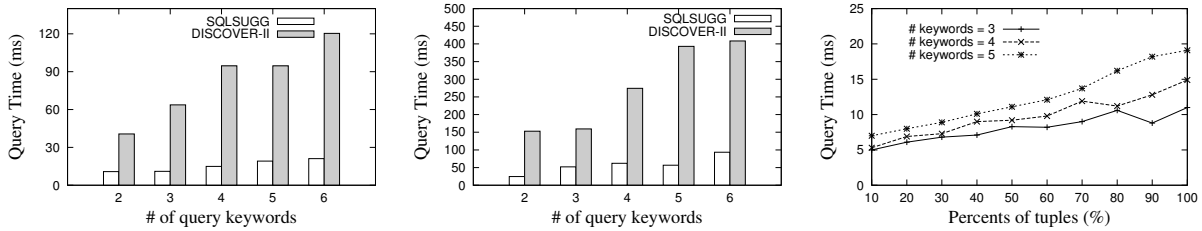
(a) Effectiveness of template suggestion. (b) Effectiveness of SQL generation. (c) Effectiveness of record retrieval.

Fig. 7. Effectiveness comparisons on the DBLP data set.



(a) Effectiveness of template suggestion. (b) Effectiveness of SQL generation. (c) Effectiveness of record retrieval.

Fig. 8. Effectiveness comparisons on the DBLIFE data set.



(a) Efficiency comparison (DBLP). (b) Efficiency comparison (DBLIFE). (c) Scalability (DBLP).

Fig. 9. Efficiency and scalability comparisons.

The improvement is due to the reason that SQLSUGG cannot only support full-text search, but also allow keywords to refer to meta-data and aggregate functions. The experimental result shows that SQLSUGG can help users formulate a broader class of SQL queries, and thus can obtain effective result records capturing more information needs. In addition, even for full-text search, SQLSUGG can also retrieve more precise records than DISCOVER-II. For example, in Figure 7(c), SQLSUGG can achieve higher precisions for Q_1, Q_2, Q_3 and Q_5. The reason is that SQLSUGG matches keywords to specific attributes, and thus can generate more accurate result tuples.

C. Evaluation of Efficiency and Scalability

We examined the efficiency of our SQL suggestion methods, and compared the query time[6] of SQLSUGG with that of DISCOVER-II. Figures 9(a) and 9(b) show the experiment results on the two data sets. The results show that our methods outperform DISCOVER-II significantly, especially in the DBLIFE data set. For example, consider the query time for a keyword query with length 6 in Figure 9(b). SQLSUGG outperforms DISCOVER-II by an order of magnitude. Moreover, on the two data set, the query time of SQLSUGG is very stable, and is always smaller than 100 ms. It indicates that SQLSUGG can suggest SQL queries in real time.

The improvement of our methods is mainly attributed to the top-k template ranking algorithm. DISCOVER-II exploits a keyword-to-attribute index to find tuple sets that may contain query keywords, and on-the-fly generates candidate networks. Since the amount of candidate networks could be large, especially for complex schemas, DISCOVER-II is inefficient to generate all candidate networks. For example, the query time of DISCOVER-II is hundreds of milliseconds on the DBLIFE data set which has 14 relation tables. In contrast, SQLSUGG focuses on suggesting top-k templates according to our ranking model. The experimental result shows that the algorithm is very efficient and can suggest template in real time.

Next, we examine the scalability of SQLSUGG on the DBLP data set. Initially, the entire database was empty. We inserted 10% of tuples in every relation table at each time, and ran SQLSUGG to evaluate the query time. Figure 9(c) shows the experiment results. SQLSUGG scales very well on the DBLP data set. The query time increases sub-linearly as the data set increases. For example, the query time is stable between 5 ms and 20 ms.

Summary. The experimental results show that SQLSUGG has the following advantages. Firstly, for advanced users who want to pose SQL queries, SQLSUGG can suggest heavily relevant templates, and can generate accurate SQL queries. Secondly, for casual users, SQLSUGG can retrieve records capturing more information needs, that is, SQLSUGG allows users to specify tables and attributes of the returned records, and can return aggregation tuples. Thirdly, SQLSUGG achieves higher efficiency than DISCOVER-II and scales very well due to our proposed ranking algorithms.

[6]All algorithms were tested using the same computer with the same system configuration and the same system/application workload.

VI. Related Work

The area most related to our work is keyword search over relational databases. The existing approaches of keyword search over relational databases can be broadly classified into two categories: those based on the Steiner tree [2,11,12,14], and others based on the candidate network (CN) [1,8,9,17,22]. The Steiner tree based methods first model the tuples in a relational database as a data-graph, where nodes are tuples and edges are the primary-foreign-key relationships. Steiner trees which contain all or some of input keywords are then identified to answer keyword queries. The candidate network based methods identify answers composed of relevant tuples by generating and extending a candidate network following the primary-foreign-key relationship. Compared with these methods, SQLSUGG provides a more powerful keyword-search paradigm that assists users to formulate more complex SQL queries, and thus it can help users express their query intent more precisely than keyword search. In addition, SQLSUGG improves the ranking model of generating query structures of CN-based methods by taking into account the keyword-to-template relevance and template query ability, and extends the methods by matching keywords to specific attributes.

Recently, some studies proposed keyword-based methods to help users pose aggregation SQL queries [19,23]. The basic idea of these methods is similar to ours, that is, to take keyword queries as input and SQL queries as output. Except that SQLSUGG can support a broader class of SQL queries rather than aggregation queries, we propose a more effective SQL query ranking mechanism, which is not considered by existing methods, and devise an efficient algorithm for supporting the ranking mechanism.

The concept, template, that captures query structure has also been investigated in form-based search [4,10]. The methods propose to summarize the schema and the underlying data to generate a good set of templates. However, their templates are *keyword-independent*. In contrast, SQLSUGG further considers the relevance between templates and keywords, and proposes to generate specific SQL queries from templates. Therefore, SQLSUGG can allow users to not only select templates using keywords, but also to generate most promising SQL queries from the general templates.

VII. Conclusion and Future Work

In this paper, we have proposed an effective and user-friendly keyword-based method, called SQLSUGG, to suggest SQL queries based on limited keyword queries. We proposed *queryable templates* to model the structures of SQL queries, and generated templates from the schema graph. We ranked the templates by their relevance to the keyword query, and devised a progressive algorithm to compute top-k templates efficiently. To generate SQL queries from templates, we proposed a generation model by considering the degree of matchings between keywords and attributes, and devised a greedy algorithm to compute the best matching between keywords and attributes. We have implemented our approach and examined it on two real data sets. The experimental results show that our approach achieves high effectiveness and efficiency.

We believe this study on SQL suggestion opens many new interesting and challenging problems that need further research investigation, such as how to estimate the number of returned records for each SQL, how to support non-string data types, how to allow users to express more information, and how to support personalized suggestion.

VIII. Acknowledgement

This work is partly supported by the National Natural Science Foundation of China (NSFC) under Grant No. 61003004, and the NSFC under Grant No. 60833003. We appreciate the helpful comments of the anonymous reviewers.

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] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[4] E. Chu, A. Baid, X. Chai, A. Doan, and J. F. Naughton. Combining keyword search and forms for ad hoc querying of databases. In *SIGMOD Conference*, pages 349–360, 2009.

[5] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233235, 1979.

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

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

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

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

[10] M. Jayapandian and H. V. Jagadish. Automated creation of a forms-based database query interface. *PVLDB*, 1(1):695–709, 2008.

[11] 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.

[12] B. Kimelfeld and Y. Sagiv. Efficiently enumerating results of keyword search over data graphs. *Inf. Syst.*, 33(4-5):335–359, 2008.

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

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

[15] H. Lu, H. C. Chan, and K. K. Wei. A survey on usage of sql. *SIGMOD Rec.*, 22(4):60–65, 1993.

[16] C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. *J. ACM*, 41(5):960–981, 1994.

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

[18] Y. Luo, W. Wang, and X. Lin. Spark: A keyword search engine on relational databases. In *ICDE*, pages 1552–1555, 2008.

[19] S. Tata and G. M. Lohman. Sqak: doing more with keywords. In *SIGMOD Conference*, pages 889–902, 2008.

[20] X. Yang, C. M. Procopiuc, and D. Srivastava. Summarizing relational databases. *PVLDB*, 2(1):634–645, 2009.

[21] C. Zhai. Statistical language models for information retrieval: A critical review. *Foundations and Trends in Information Retrieval*, 2(3):137–213, 2008.

[22] J. Zhang, Z. Peng, S. Wang, and H. Nie. Clascn: Candidate network selection for efficient top- keyword queries over databases. *J. Comput. Sci. Technol.*, 22(2):197–207, 2007.

[23] B. Zhou and J. Pei. Answering aggregate keyword queries on relational databases using minimal group-bys. In *EDBT*, pages 108–119, 2009.