Computing Structural Statistics by Keywords in Databases

Lu Qin, Jeffrey Xu Yu, *Senior Member*, *IEEE*, and Lijun Chang

Abstract—Keyword search in *RDB*s has been extensively studied in recent years. The existing studies focused on finding all or top-k interconnected tuple-structures that contain keywords. In reality, the number of such interconnected tuple-structures for a keyword query can be large. It becomes very difficult for users to obtain any valuable information more than individual interconnected tuple-structures. Also, it becomes challenging to provide a similar mechanism like group-&-aggregate for those interconnected tuple-structures. In this paper, we study computing structural statistics keyword queries by extending the group-&-aggregate framework. We consider an *RDB* as a large directed graph where nodes represent tuples, and edges represent the links among tuples. Instead of using tuples as a member in a group, we consider rooted subgraphs. Such a rooted subgraph represents an interconnected tuple-structure among tuples and some of the tuples contain keywords. The dimensions of the rooted subgraphs are determined by dimensional keywords in a data driven fashion. Two rooted subgraphs are grouped into the same group if they are isomorphic based on the dimensions or in other words the dimensional keywords. The scores of the rooted subgraphs are computed by a user-given score function if the rooted subgraphs contain some of general keywords. Here, the general keywords are used to compute scores rather than determining dimensions. The aggregates are computed using an SQL aggregate function for every group based on the scores computed. We give our motivation using a real data set. We propose new approaches to compute structural statistics keyword queries, perform extensive performance studies using two large real data sets and a large synthetic data set, and confirm the effectiveness and efficiency of our approach.

Index Terms—Keyword search, relational database, structural statistics.

1 Introduction

Keyword search on relational databases (*RDB*s) has been extensively studied. It allows users to query *RDB*s using keywords. Most of the existing studies focused on finding interconnected structures among tuples via foreign key references in a relational database that contain the keywords [1], [12], [10], [19], [20], [22], [3], [14], [15], [6], [8], [9], [5], [16], [24].

Take the *DBLP* data set (http://www.informatik.uni-trier.de/~ley/db/) as an example to be stored in an *RDB* with four relations: Author(AID, Name, Affiliation), Paper(PID, Type, Title, CID), Conference(CID, Name, Year, Location), and Write(AID, PID). In the Author relation, an author is identified by AID, and is with a name and an affiliation. In the Paper relation, a paper is identified by PID, and has a type which can be either journal or conference. If a paper is a conference paper, its CID value refers to a conference in the Conference relation. A Conference is with a name, year, and location. The Write relation specifies the write relationships between authors and papers. In the most recent *DBLP* data set we use in our study, there are 750,000 authors, among them we obtained affiliation information for 40,000 authors. Consider a keyword query {"keyword," "search," "graph"}. The existing approaches may find one among many interconnected structures such that an author Jim writes a paper that contains "keyword search" in its title and his coauthor writes another paper that contains "graph" in its title. In reality, the number of such interconnected structures to be returned can be large. Thus, it becomes very difficult for users to identify any additional valuable information. In this paper, instead of finding interconnected structures among tuples, we study how to compute statistics on the interconnected tuple-structures using keywords.

Consider five keyword queries against the *DBLP* stored in *RDB*. (Q_1) Which conference is good for SQL query optimization? (Q_2) Which author is an expert on keyword search on graphs? (Q_3) Which author in which year has most papers about graph pattern mining? (Q_4) In which year, which conference is best for information retrieval on the web? (Q_5) In year 2007, which university has most papers about random walk on graphs? All these queries do not ask for what the individual interconnected structures look like. Instead, all these queries ask for some statistics. Take Q_1 as an example. Q_1 is to compute some statistics regarding conferences about SQL query optimization. In other words, the main question of Q_1 is on conferences, and the statistics to be collected regarding conferences are based on SQL query optimization. To the best of our knowledge, there does not exist any keyword search approach which can find valuable statistical information to answer the five queries.

In this paper, we study computing structural statistics for keyword queries by extending the group-&-aggregate computing framework. Recall that in an RDBMS tuples are grouped into the same group if they have the same attribute values in the user-specified dimensional attributes (or

• *The authors are with the The Chinese University of Hong Kong, William M. W. Mong Engineering Building, Shatin, Hong Kong. E-mail: {lqin, yu, ljchang}@se.cuhk.edu.hk,*

Manuscript received 30 May 2011; revised 3 Jan. 2012; accepted 1 Apr. 2012; published online 5 Apr. 2012.
Recommended for acceptance by S. Abiteboul, C. Koch, K.-L. Tan, and J. Pei. For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDESI-2011-05-0304.
Digital Object Identifier no. 10.1109/TKDE.2012.78.

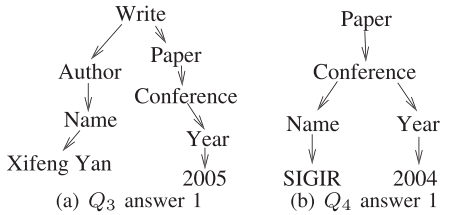


Fig. 1. The dimensional objects.

simply dimensions), and an SQL aggregate function is used to aggregate the numerical attribute values of the tuples in the same group. The four main factors in group-&-aggregate computing are tuples, dimensions, a numerical attribute to which an SQL aggregate function will be applied. In our study, we consider an *RDB* as a large directed graph where nodes represent tuples, and edges represent the links among tuples. There is an edge from a tuple t_i to another tuple t_j if a foreign key defined on t_i references to the primary key defined on t_j. We extend the group-&-aggregate computing framework as follows: 1) Instead of using tuples as a member in a group to be grouped, we consider rooted subgraphs as virtual tuples. Such a rooted subgraph represents an interconnected structure among tuples and some of the tuples contain keywords. 2) Regarding dimensions, it is worth noting that one of the main advantages of keyword queries is to find hidden interconnections among tuples in an *RDB* no matter where the keywords may appear or how the keywords appear. In other words, the dimensions in the conventional group-&-aggregate computing become data driven in keyword search contexts. We introduce new dimensions which are determined by dimensional-keywords (a subset of user-given keywords). Let t_γ be a virtual tuple rooted at tuple t_γ. We use a set of dimensional-keywords to determine a dimensional object called Dtree rooted at t_γ where its leaf nodes contain all the dimensional-keywords. Two virtual tuples rooted at t_γ and t'_γ are grouped into the same group if their dimensional objects are isomorphic. 3) Regarding the numerical attribute to which an aggregate function is applied, we extract a Gtree from a virtual tuple t_γ using a set of general-keywords, and we apply a score function (α) to score the extracted Gtree as a virtual document, and then we aggregate such scores in a group using an SQL aggregate function (β).

The five queries can be specified using keywords as follows: {"*conference*," "SQL," "query," "optimization"} for Q_1, {"*author*," "keyword," "search," "graph"} for Q_2, {"*author*" "*year*," "pattern," "mining," "graph"} for Q_3, {"*year*," "*conference*," "information," "retrieval," "web"} for Q_4, and {"*2007*," "*university*," "random," "walk," "graph"} for Q_5. Here, the underlined keywords are dimensional keywords, and the remaining keywords are general keywords. Let the score function (α) be the tree level ranking function used in *SPARK* [19], and the aggregate function be $\beta = sum$. We list three answers with highest scores for the five queries returned by our system below.

- Q_1: (SIGMOD, 340.1), (VLDB, 274.5), (ICDE, 268.0).
- Q_2: ("Benny Kimelfeld," 31.9), ("Yehoshua Sagiv," 23.7), ("Yannis Papakonstantinou," 18.8).
- Q_3: ("Xifeng Yan," 2005, 8.2), ("Wei Wang," 2005, 6.8), ("Jiawei Han," 2007, 6.7).

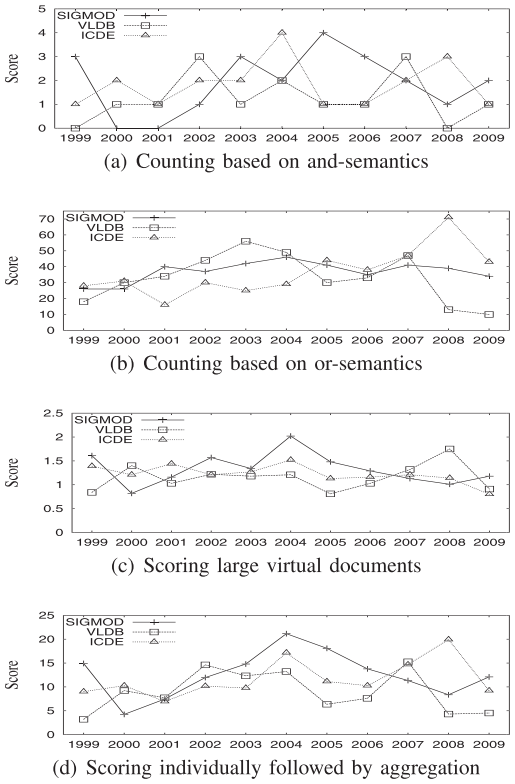


Fig. 2. "*conference*," "*year*," "data," "integration."

- Q_4: (2004, SIGIR, 66.9), (2008, CIKM, 57.1), (2007, SIGIR, 46.1).
- Q_5: (2007, "Carnegie Mellon University," 12.0), (2007, "University of California," 8.7), (2007, "University of Waterloo," 7.8).

The dimensional objects (Dtree) for the first answer of Q_3 is shown in Fig. 1a. It represents the structure of an author, Xifeng Yan, writes papers accepted in conferences in year 2005. Another example shown in Fig. 1b is the dimensional objects (Dtree) for the first answer of Q_4. It represents the structure of papers accepted in the SIGIR conference held in year 2004.

We further explain the meaning of α and β functions using another query: in which conference and which year, there are most papers about data integration (Q_6). This query can be formulated as a structural statistics keyword query {"*conference*," "*year*," "data," "integration"}. All the dimensional objects (Dtrees) contain both a conference name and a year, using the 2D keywords. Such a dimensional object may start from a paper tuple, t_p, which links to a conference tuple t_c. The conference tuple t_c contains "conference" in its Name attribute, and contains a year value because there is an attribute named Year in the Conference relation, which the "year" keyword matches. In this simple application, the two keywords "data" and "integration" may appear in the title of the paper, and may appear as a part of the conference name. The related virtual document (Gtree) for the same paper tuple t_p is an interconnected tuple-structure from t_p that may contain either "data," or "integration," or both. We plot the answers in Fig. 2 using different α score functions, and different β aggregate functions. All the four figures in Fig. 2 show scores for the three major database conferences, SIGMOD, VLDB, and ICDE from 1999-2009.

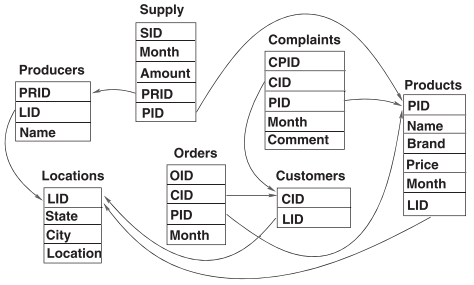


Fig. 3. A simple e-commerce database schema.

Figs. 2a and 2b show the scores using simple counts. Fig. 2a simply shows how many papers in a conference in a year that have both "data" and "integration" in the titles. It misses the papers that contain only one keyword. For example, in year 2001, there are three SIGMOD papers that contain only "integration" but not "data." Fig. 2b simply shows how many papers in a conference in a year that has either "data," or "integration," or both in the titles. It may overcount. For example, in year 2003, there are only three VLDB papers containing "integration," but 53 VLDB papers containing "data." Fig. 2c shows the TF-IDF score (α) by treating all the papers that contain either "data," or "integration," or both in their titles as a large virtual document for a conference in a year. Such approach is better than simple counts. But it may give some information which causes confusion. Consider a conference in two different years, y_1 and y_2. In year y_1, there are two papers, each of the two papers contains "data" and "integration" once. In year y_2, there are two papers, one contains "data" twice, and one contains "integration" twice. They have the same counts for the two keywords in y_1 and y_2. Obviously, the conference should have a higher score in y_1 than y_2. For example, in year 2008, there are three VLDB papers that contain "integration," but none of them contain both "data" and "integration." In year 2007, there are three VLDB papers that contain both "data" and "integration." But in Fig. 2c, VLDB 2007 is scored lower than VLDB 2008. In our approach, we consider each paper as an individual virtual document, and use a TF-IDF based α function to score, and then sum up all the scores for a conference in a year. The results are shown in Fig. 2d. With our approach, VLDB 2007 is scored higher than VLDB 2008.

The main contributions of this work are summarized below. First, we study a new structural statistics keyword query in an *RDB*. We extend the existing work on group-&-aggregate over attribute values in several ways based on keywords in a data-driven fashion. Second, we give a two-step approach to process a structural statistics keyword query using label trees. A label tree is obtained based on the schema information. We show how to avoid tree isomorphism testing, and how to share cost in processing a structural statistics keyword query, using the label-trees. Third, we performed extensive performance studies using two large real data sets and a large synthetic data set, and confirmed the effectiveness and efficiency of our approach. The early work is presented in [23].

The remainder of the paper is organized as follows: in Section 2, we discuss structural statistics keyword queries. In Section 3, we outline our two-step approach to compute structural statistics. We discuss the two steps in Sections 4 and 5, and discuss OLAP issues in Section 6. The related work is given in Section 7. We show our experimental studies in Section 8 and conclude our paper in Section 9.

CPID	CID	PID	Month	Comment
cp_1	c_1	p_1	February 2008	The monitor's quality is bad
cp_2	c_1	p_2	March 2008	It is not good for the computer
cp_3	c_2	p_2	March 2008	It is too small
cp_4	c_3	p_3	April 2008	I'm not satisfied for the color
cp_5	c_4	p_1	June 2008	The quality should be improved
cp_6	c_5	p_3	June 2008	I don't like the monitor
cp_7	c_3	p_4	May 2008	My monitor is broken
cp_8	c_2	p_5	May 2008	It is a little expensive

(a) Complaints

LID	State	City	Location
l_1	Illinois	Chicago	No.34, Road 3
l_2	Illinois	Chicago	No.53, Street 10
l_3	Nevada	Las Vegas	KM Tower, Road 9
l_4	California	Los Angeles	Eg Building, Street 33
l_5	California	Orange	No.121, Street 4
l_6	Washington	Tiger	No.113, Street 5

(b) Locations

CID	LID
c_1	l_1
c_2	l_2
c_3	l_3
c_4	l_4
c_5	l_4

(c) Customers

PID	Name	Brand	Price	Month	LID
p_1	Computer V7011	Orange	\$1000	Jan. 2008	l_5
p_2	Monitor 3500 Black	Tiger	\$300	Feb. 2008	l_6
p_3	Computer E1003	Orange	\$1300	Jan. 2008	l_6
p_4	Monitor K133 Large	Tiger	\$100	Mar. 2008	l_5
p_5	Monitor K140	Tiger	\$190	Mar. 2008	l_6

(d) Products

Fig. 4. Four relations in the e-commerce database.

2 Structural Statistics

Let $GS = \{R_1, R_2, \ldots\}$ be a relational database schema. Here, R_i in GS is a relation schema with a set of attributes. We call an attribute a text-attribute in the paper if the attribute is defined on either string (char/varchar) or full-text domain. Keyword search is allowed on any text-attributes. A relation schema may have a primary key and there are foreign key references defined in GS. We use $R_i \rightarrow R_j$ to denote that there is a foreign key defined on R_i referring to the primary key defined on R_j. A relation on relation schema R_i is an instance of the relation schema (a set of tuples) conforming to the relation schema, denoted $r(R_i)$. A relational database (*RDB*) is a collection of relations.

Example 2.1. An e-commerce database schema, GS, is shown in Fig. 3, which is modified based on TPC-H (http://www.tpc.org/tpch). It consists of seven relation schemas: Products, Customers, Orders, Locations, Producers, Complaints, and Supply. There are foreign key references denoted by $\rightarrow$. For example, there are two foreign key references from Complaints to two other relations Customers and Products, respectively. The text attributes include Name, Brand, Month, State, City, Location, and Comment. A part of the e-commerce database is shown in Fig. 4 including four relations, namely, Complaints, Locations, Customers, and Products.

We model an *RDB* over GS as a directed database graph $G_D(V, E)$. For easy discussion, we use two types of labeled nodes in the following discussions, $V = V_t \cup V_a$ for $V_t \cap V_a = \emptyset$. Here, V_t is the set of *tuple-nodes* for all tuples in *RDB*. A tuple-node, t_i, represents a tuple in $r(R')$ and is labeled with its relation name R'. V_a is the set of *attribute-nodes* for every distinctive pair of text-attribute and attribute value in *RDB*. An attribute-node is labeled $A_j : a_k$ where A_j

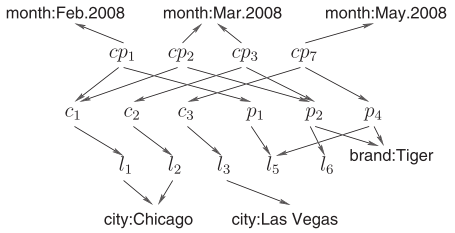


Fig. 5. A database subgraph for the e-commerce DB.

is a text-attribute name and a_k is an attribute value. Accordingly, $E = E_{tt} \cup E_{ta}$ consists of two types of edges. E_{tt} is the set of edges among tuple-nodes in G_D, and an edge $t_i \rightarrow t_j$ in E_{tt} indicates that there exists a foreign key reference from t_i to t_j in *RDB*. E_{ta} is the set of edges from tuple-nodes to attribute-nodes. There exists an edge from a tuple-node t_i to an attribute-node $A_j : a_k$ in E_{ta}, if the tuple t_i has the attribute value a_k in the attribute A_j in *RDB*. We will use $V(G)$ and $E(G)$ to denote the set of nodes and the set of edges of G, respectively.

A part of the database graph G_D for the *RDB* in Fig. 4 is shown in Fig. 5. Here, cp_i, c_i, p_i, and l_i are tuple-nodes whose labels are Complaints, Customers, Products, and Locations, respectively. We omit the tuple-node labels in Fig. 5. There are also attribute nodes. For example, p_2 links to an attribute-node labeled "brand:Tiger." We say an attribute-node v contains a keyword k_i if k_i is contained in either the attribute name or the attribute value in the node label. In Fig. 5, an attribute-node labeled "brand:Tiger" contains the keywords "brand" and "tiger." We also say a tuple-node u contains a keyword k_i if there is a path from tuple node u to an attribute node v that contains k_i.

Virtual tuple. We define a virtual tuple which is a tree representation of the maximum subgraph at a tuple-node t_γ in G_D. We denote such a virtual tuple as Vtuple, or explicitly $\mathrm{Vtuple}(t_\gamma)$ if it is rooted at a tuple-node t_γ. The maximum subgraph of t_γ is the induced subgraph of all nodes that are reachable from t_γ in G_D. The tree representation is done as follow. In $\mathrm{Vtuple}(t_\gamma)$, if a node u links to a node w which is an ancestor of u, the edge (u, w) will be deleted. Otherwise, if two nodes u and v link to a node w in G_D, we create an additional copy w' of w, and let u link to w and v link to w'. All leaf nodes in $\mathrm{Vtuple}(t_\gamma)$ must be attribute-nodes. A $\mathrm{Vtuple}(t_\gamma)$ includes all information a tuple-node t_γ can reach.

Dtree, Gtree, and DGtree. Given a set of keywords $\{k_1, k_2, \ldots\}$, a $\mathrm{Dtree}(t_\gamma)$ is a minimal subtree of $\mathrm{Vtuple}(t_\gamma)$ which contains all dimensional-keywords by connecting to the attribute-nodes that contain the given dimensional-keyword(s). By minimal, we mean that the Dtree with root t_γ does not contain all the dimensional-keywords, if any node is deleted from the Dtree. A $\mathrm{Gtree}(t_\gamma)$ is also a subtree of $\mathrm{Vtuple}(t_\gamma)$ by removing all the subtrees rooted at a tuple-node that do not have any attribute-node containing general-keywords. A $\mathrm{Gtree}(t_\gamma)$ matches a query if it contains at least one general-keyword. A $\mathrm{Gtree}(t_\gamma)$ may not contain all the general-keywords. Given a set of keywords, there exists one $\mathrm{Gtree}(t_\gamma)$, but many distinctive $\mathrm{Dtree}(t_\gamma)$, by definition. A $\mathrm{DGtree}(t_\gamma)$ consists of two parts, where the first part is $\mathrm{Dtree}(t_\gamma)$ and the second part is $\mathrm{Gtree}(t_\gamma)$.

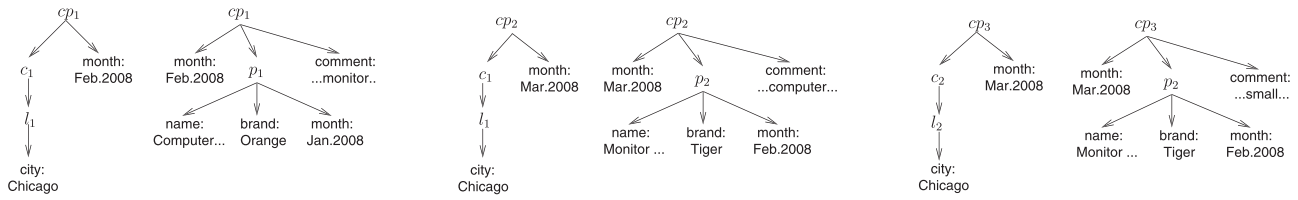
In this paper, we study a new structural statistics keyword query $Q = (Q_d, Q_g, \alpha, \beta)$ against an *RDB* over GS. It consists of two sets of keywords, namely Q_d and Q_g, for $Q_d \cap Q_g = \emptyset$, a score function α, and an aggregate function β. We call a keyword in Q_d and Q_g as a dimensional-keyword and a general-keyword, respectively. The two sets of keywords, Q_d and Q_g, together specify a set of trees $\mathcal{T}$ to be computed. A DGtree $T_i \in \mathcal{T}$ with root $t_\gamma = \mathrm{root}(T_i)$ consists of two subtrees, a Dtree rooted at t_γ for Q_d, and a Gtree rooted at the same t_γ for Q_g, denoted as $\mathrm{Dtree}(T_i)$ and $\mathrm{Gtree}(T_i)$, respectively. The set of trees, $\mathcal{T}$, are grouped into different groups. Let T_i and T_j be two trees in $\mathcal{T}$. T_i and T_j belong to the same group if $\mathrm{Dtree}(T_i)$ is isomorphic to $\mathrm{Dtree}(T_j)$. Here, the tree isomorphism is defined over the labeled Dtrees. As an example, month:Mar.2008 $\leftarrow cp_2 \rightarrow c_1 \rightarrow l_1 \rightarrow$ city:Chicago and month:Mar.2008 $\leftarrow cp_3 \rightarrow c_2 \rightarrow l_2 \rightarrow$ city:Chicago in Fig. 5 are isomorphic to each other because they have the same structure and the same labels, month:Mar.2008 $\leftarrow$ Complaints $\rightarrow$ Customers $\rightarrow$ Locations $\rightarrow$ city:Chicago. T_i and T_j may have different Gtrees, even though they are in the same group.

We allow a score function α to be any possible algebraic function based on TF-IDF, namely, $tf_w(T)$ and idf_w We explain it below. Let $\mathcal{T} = \{T_1, T_2, \ldots\}$ be a set of $\mathrm{DGtrees}$ in the same group. We consider every $\mathrm{Gtree}(T_i)$ for $T_i \in \mathcal{T}$ as a virtual document, by merging all attribute names and attribute values in the tree into a multiset. Then, $tf_w(T_i)$ is the number of times the keyword $w \in Q_g$ appears in the corresponding virtual document, and idf_w is calculated as follows:

$$idf_w = \frac{|\mathcal{T}|}{df_w(\mathcal{T}) + 1}, \tag{1}$$

where $|\mathcal{T}|$ is the number of Gtrees in $\mathcal{T}$ in the group, and $df_w(\mathcal{T})$ is the number of Gtrees that contain the keyword $w \in Q_g$ in the group. The tree level ranking function used in *SPARK* [19] is such an algebraic function based on $tf_w(T)$ and idf_w. The α function is to be applied to $\mathrm{Gtree}(T_i)$ for $T_i \in \mathcal{T}$ to give such $\mathrm{Gtree}(T_i)$ a score. Factors that can be involved in an α function can be, for example, to give a high score for a term if it is close to the root. For efficiency consideration, in this work, we require that factors in an α function for a tree must be computable from the factors of its subtrees. The aggregate function β aggregates the scores computed for $\mathrm{DGtrees}$ in the same group. An aggregate function can be any SQL aggregate functions (min, max, sum, avg, and $count$). The output for the group $\mathcal{T}$ is (T_A, ω), where T_A represents the Dtree for the group, and $\omega = \beta(\{\alpha(\mathrm{Gtree}(T_1)), \alpha(\mathrm{Gtree}(T_2)), \ldots\})$.

Consider $Q_d =$ {"city," "month"} and $Q_g =$ {"computer," "monitor"}. Fig. 6 shows three pairs of Dtrees and Gtrees rooted at cp_1, cp_2, and cp_3, respectively, based on the *RDB* in Fig. 4. We omit the tuple-node labels in Fig. 6. A DGtree consists of a Dtree and a Gtree rooted at the same node. For example, a DGtree with root cp_1 consists of a Dtree with root cp_1 and a Gtree with root cp_1 in Fig. 6a. The Dtrees with root cp_2 in Fig. 6b and cp_3 in Fig. 6c are isomorphic to each other, because their labeled Dtrees are isomorphic to each other. Note that month:Mar.2008 $\leftarrow cp_2 \rightarrow c_1 \rightarrow l_1 \rightarrow$ city:Chicago and month:Mar.2008 $\leftarrow cp_3 \rightarrow c_2 \rightarrow l_2 \rightarrow$ city:Chicago have the same structure with the same labels, month:Mar.2008 $\leftarrow$ Complaints $\rightarrow$ Customers $\rightarrow$ Locations $\rightarrow$ city:Chicago. The corresponding $\mathrm{DGtrees}$ rooted at cp_2 and cp_3 belong to



(a) A Dtree and a Gtree with root cp_1 (b) A Dtree and a Gtree with root cp_2 (c) A Dtree and a Gtree with root cp_3

Fig. 6. Dtrees and Gtrees.

the same group. The Dtree with root cp_1 in Fig. 6a is not isomorphic to the Dtree with root cp_2 in Fig. 6b.

We list the notations used in Fig. 7.

3 Solution Overview

Given a structural statistics keyword query $Q = (Q_d, Q_g, \alpha, \beta)$ over an *RDB*, a naive solution is to first compute all Dtrees using Q_d, which can be done using any existing algorithm in the literature. Let $\mathcal{T}_A = \{T_{a_1}, T_{a_2}, \ldots\}$ be the set of nonempty Dtrees computed. Second, for each $T_{a_i} \in \mathcal{T}_A$, it expands from $\text{root}(T_{a_i})$ to identify its Gtree using Q_g. Let $\mathcal{T}_O = \{T_{o_1}, T_{o_2}, \ldots\}$ be the set of corresponding Gtrees computed. Third, it computes $\alpha(T_{o_i})$ for each nonempty $T_{o_i} \in \mathcal{T}_O$. Fourth, it groups all Dtrees, T_{a_i} into groups, if the corresponding T_{o_i} is nonempty. Finally, it computes β for each group, and outputs the results. Such a naive solution is impractical for the following two main reasons. 1) The number of possible Dtrees and Gtrees can be very large. It is infeasible to compute. It is worth noting that all the existing solutions focus on finding top-k answers if they make use of ranking and allow some (not all) keywords to be contained in the answers. 2) It is costly to group Dtrees into groups even though tree isomorphism checking is polynomial.

In this paper, to compute a structural statistics keyword query, Q, we propose a two-step approach (Algorithm 1). In the first step, we generate a set of label-trees for dimensional keywords (LDs), denoted as $\mathcal{L} = \{\text{LD}_1, \text{LD}_2, \ldots\}$, such that every DGtree to be computed will conform to a unique LD. We will formally define LD trees shortly. In the second step, we compute the structural statistics keyword query Q using $\mathcal{L}$. In brief, for every LD_i, we compute all DGtrees, denoted as $\mathcal{T}_i = \{T_{i_1}, T_{i_2}, \ldots\}$ that conform to LD_i, group all the trees in $\mathcal{T}_i$ into groups based on $\text{Dtree}(T_{i_j})$ and compute $\alpha(\text{Gtree}(T_{i_j}))$ for every $T_{i_j} \in \mathcal{T}_i$, and then compute β for every group. The main idea behind label-trees is to avoid tree isomorphism checking and enumerating all possible DGtrees. In this section, we discuss the label-trees. The algorithms to generate all label-trees and to compute structural statistics keyword query using the label-trees will be discussed in the following sections.

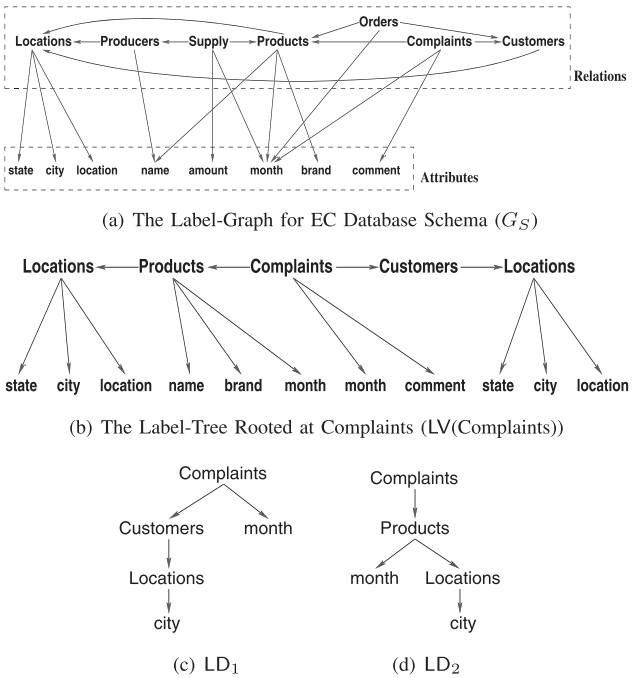
Notation	Meaning
$\text{Vtuple}(t)$	The virtual tuple (subtree) rooted at tuple node t
$\text{Dtree}(t)$	A subtree of Vtuple (t) containing all dimensional-keywords
$\text{Gtree}(t)$	A subtree of Vtuple (t) containing general-keywords
$\text{DGtree}(t)$	A tree consists of a Dtree (t) and a Gtree (t)
$G_S(V, E)$	The label-graph generated from the database schema graph
$\text{LV}(R_i)$	A labeled tree rooted at R_i generated from G_S for virtual tuples
$\text{LD}_j(R_i)$	A labeled tree generated from $\text{LV}(R_i)$ for dimensional keywords

Fig. 7. Notations.

Algorithm 1. Structural-Statistics(GS, Q, G_D)
1: $\mathcal{L} = \text{LD-Gen}(GS, Q)$;
2: **for every** $\text{LD}_i \in \mathcal{L}$ **do**
3: LD-Eval(Q, LD_i, G_D);

First, we define a label-graph, $G_S(V, E)$, for keyword search for a database schema GS. Here, V is a set of nodes such that $V = V_R \cup V_A$, where V_R is a set of nodes labeled with relation names, called relation nodes, and V_A is a set of nodes labeled with attribute names for those text-attributes only. E is a set of edges such that $E = E_{RR} \cup E_{RA}$. An edge $R_i \rightarrow R_j$ appears in E_{RR} if there is a foreign key reference from R_i to R_j. An edge $R_i \rightarrow A_j$ appears in E_{RA} if R_i has an attribute A_j. The label-graph G_S for the e-commerce database schema (Fig. 3) is shown in Fig. 8a.

Second, we define a rooted tree for every relation R_i in G_S, denoted as $\text{LV}(R_i)$, which is a labeled tree for all virtual tuples rooted at $t_\gamma \in r(R_i)$. $\text{LV}(R_i)$ is a connected tree representation of the maximum subgraph rooted at R_i in G_S. Here, the maximum subgraph of $\text{LV}(R_i)$ is the induced subgraph of all nodes that are reachable from R_i in G_S. The tree representation is done as follow. In $\text{LV}(R_i)$, if a node u



(a) The Label-Graph for EC Database Schema (G_S)

(b) The Label-Tree Rooted at Complaints (LV(Complaints))

(c) LD_1 (d) LD_2

Fig. 8. Label-graphs.

links to a node w which is an ancestor of u, the edge (u, w) will be deleted. Otherwise, if two nodes u and v link to a node w in G_S, we create an additional copy w' of w, and let u link to w and v link to w'. The reason of making a copy is explained below. Consider Fig. 3, the Location relation may be linked by two different relations, namely, Customers and Products, in the same LV(Complains). But the locations for customers are not necessary to be the same locations for products. All leaf nodes in an LV must be attribute names. The LV(Complaints) for G_S (Fig. 8a) is shown in Fig. 8b. An $\mathrm{LV}(R_i)$ is independent from any structural statistics keyword query.

Now consider a structural statistics keyword query $Q = (Q_d, Q_g, \alpha, \beta)$ against the data graph G_D using the label-graph G_S. We further give a specific LV for computing Dtrees. A label-tree for dimensional-keywords $\mathrm{LD}_i(R_j)$ is a subtree of $\mathrm{LV}(R_j)$ rooted at R_j that contains at most $|Q_d|$ attribute-nodes as leaf nodes. The attribute-nodes in $\mathrm{LD}_i(R_j)$ possibly contain all the dimensional-keywords in Q_d. When a leaf node of $\mathrm{LD}_i(R_j)$ contains multiple dimensional-keywords, for example, two appear in an attribute name, we create a virtual leaf node for each dimensional-keyword in $\mathrm{LD}_i(R_j)$, because each of them focuses on a different aspect. Given an LD tree, we use $\mathrm{att}(\mathrm{LD})$ to denote the set of attributes (or attribute-nodes) in the tree. Consider $Q = (Q_d, Q_g, \alpha, \beta)$, where $Q_d =$ {"city," "month"} and $Q_g =$ {"computer," "monitor"}. There are two LDs, LD_1 and LD_2 in Figs. 8c and 8d, which are subtrees of LV(Complaints) in Fig. 8b. In this example, both keywords "city" and "month" are contained in the corresponding attributes in LD_1 and LD_2. There may exist other LDs if they have attributes that possibly contain both keywords "city" and "month."

We observe that any $\mathrm{Dtree}(t_\gamma)$ conforms to one $\mathrm{LD}_k(R_i)$ if t_γ represents a tuple in the relation $r(R_i)$. By conformation, we mean that $\mathrm{Dtree}(t_\gamma)$ is isomorphic to $\mathrm{LD}_k(R_i)$, if we remove all attribute values from the $\mathrm{Dtree}(t_\gamma)$. Note that a Dtree is a Vtuple where a tuple node is labeled with a relation name, and an attribute-node is labeled with a pair of attribute name and attribute value. We consider a $\mathrm{Dtree}(t_\gamma)$ as an instance of $\mathrm{LD}_k(R_i)$. We also observe that two different trees $\mathrm{Dtree}(t_i)$ and $\mathrm{Dtree}(t_j)$ are isomorphic to each other, if they have the identical $\mathrm{LD}_k(R)$ and have the same corresponding attribute values in $\mathrm{att}(\mathrm{LD}_k(R))$. The former says that two are structurally identical, and the latter says that the attribute values are the same. As an example, $\mathrm{Dtree}(cp_2)$ (left in Fig. 6b) and $\mathrm{Dtree}(cp_3)$ (left in Fig. 6c) belong to the same group (isomorphic), because the two Dtrees conform to the same LD_1(Complaints) in Fig. 8, and have the same attribute values "city:Chicago" and "month:-Mar.2008." $\mathrm{Dtree}(cp_1)$ (left in Fig. 6a) is in a different group, because the attribute values do not match other attribute values. In addition, if two Dtrees, T and T' conform to two different LDs, they do not belong to the same group, because they have different structures.

Based on our observations, instead of checking tree isomorphism, we generate all possible LDs, and enumerate all Dtrees that conform to the same LD, and further group those Dtrees that conform to the same LD using attribute-value match, because we can ensure that they have the same tree structure already if they conform to the same LD.

4 Generate All LDs

Our solution is as follows: first, we precompute all LVs for G_S because the set of LVs is query independent. The algorithm to compute all LVs is shown in Algorithm 2. For each node R in G_S, we calculate $\mathrm{LV}(R)$ using a breadth-first search from R in G_S until all nodes that can reach from R are added into $\mathrm{LV}(R)$. For node u that is visited more than once in the breadth-first search, we create an extra copy in $\mathrm{LV}(R)$ if u is not visited from its descendant. Second, in order to efficiently generate all LDs for all possible dimensional-keywords, we construct an inverted index, called the dimensional inverted index (DII), using the names and values of the attributes in the *RDB*. The inverted index helps to find the attributes in a relation that a dimensional-keyword d_i matches. In detail, for each possible dimensional-keyword, w, in DII, there is a list of entries to describe the attributes in a relation the keyword w matches. We denote the list for w as $\mathrm{list}(w)$. Each entry $e \in \mathrm{list}(w)$ has three fields: $\mathrm{e} = (\mathrm{Type}, \mathrm{Rel}, \mathrm{Attr})$. Here, Type can be either Name or Value. When $\mathrm{Type} = \mathrm{Name}$, it means w is an attribute name. When $\mathrm{Type} = \mathrm{Value}$, it means w is an attribute value. Rel and Attr indicate the relation and the attribute that w matches. Consider the relational database shown in Fig. 4. For the dimensional-keyword "month," one entry in list("month") is (Name, Complaints, month). It suggests that "month" is an attribute name in the relation Complaints.

Algorithm 2. LV-Gen(G_S)
Input: The label-graph $G_S(V_R \cup V_A, E_{RR} \cup E_{RA})$.
Output: The set of all LVs.

```
1: S ← ∅;
2: for all R ∈ V_R do
3:     add R with links to all its text-attributes into LV(R);
4:     Q ← ∅;
5:     Q.enqueue(R);
6:     while Q ≠ ∅ do
7:         R_1 ← Q.dequeue();
8:         for all R_1 → R_2 ∈ E(G_S) do
9:             let R_2' be a copy of R_2;
10:            add an edge in LV(R) from R_1 to R_2';
11:            add links from R_2' to all its text-attributes;
12:            Q.enqueue(R_2');
13:    S ← S ∪ {LV(R)};
14: return S
```

We design a new algorithm to generate all LDs using $\mathrm{list}(d_i)$ for $d_i \in Q_d$, as shown in Algorithm 3. The algorithm generates all LDs for a structural statistics keyword query. First, it collects information if a keyword d_i matches the relation nodes in each LV (lines 2-6). Given the set of LVs $\mathcal{S}$, for each LV in $\mathcal{S}$, we use $\mathrm{LV}.list_i$ to maintain the set of candidate entries in $\mathrm{list}(d_i)$ that may contribute to generating LDs from LV. Second, in a for loop (lines 7-10), for each combination of nodes $e_1, \ldots, e_{|Q_d|}$ that contain keywords $d_1, \ldots, d_{|Q_d|}$, respectively, in a certain $\mathrm{LV} \in \mathcal{S}$, we generate a LD by constructing a minimum tree that contains nodes $e_1, \ldots, e_{|Q_d|}$ in LV. Given LV and $e_1, \ldots, e_{|Q_d|}$, the LD is unique and can be computed as follows: for each leaf node

of LV, we remove all the leaf nodes that do not belong to $\{e_1, \ldots, e_{|Q_d|}\}$. We do this iteratively until no leaf node is removed. The result is a minimum tree that contains nodes $e_1, \ldots, e_{|Q_d|}$ in LV.

Algorithm 3. LD-Gen(G_S, Q, $\mathcal{S}$, DII)
Input: The label-graph G_S, a query $Q = (Q_d, Q_g, \alpha, \beta)$, the dimensional inverted index DII and the set of LVs, $\mathcal{S}$.
Output: The set of all LDs.
1: $\mathcal{Q} \leftarrow \emptyset$;
2: **for each** keyword d_i in Q_d **do**
3: **for each** $e \in \text{list}(d_i)$ **do**
4: **for each** LV $\in \mathcal{S}$ **do**
5: **if** $e.Rel \in$ LV **then**
6: LV.$list_i \leftarrow$ LV.$list_i \cup \{e\}$;
7: **for each** LV $\in \mathcal{S}$ **do**
8: **if** LV.$list_i \neq \emptyset$ for any $1 \leq i \leq |Q_d|$ **then**
9: **for all** $(e_1, \ldots, e_{|Q_d|}) \in$ LV.$list_1 \times \cdots \times$ LV.$list_{|Q_d|}$ **do**
10: construct a minimal LD that only contains attribute-nodes $(e_1, \ldots, e_{|Q_d|})$;
11: $\mathcal{Q} \leftarrow \mathcal{Q} \bigcup \{\text{LD}\}$;
12: **return** $\mathcal{Q}$;

Theorem 4.1. *The* LDs *generated by Algorithm 3 are complete and without redundancy.*

Proof Sketch.

Complete. For any possible LD, we prove that LD can be generated by Algorithm 3. Suppose the root of LD is R, LD must be a subtree of LV(R). In lines 2-6, for each dimensional keyword d_i, we put all possible attributes (leaf nodes) that may contain d_i in LV(R) into a list LV(R).$list_i$. In lines 9-10, we enumerate all combinations of attributes (leaf nodes) in LV(R), such that all the dimensional keywords should be contained. Each combination corresponds to a unique subtree of LV(R) that satisfies the minimal condition. Since LD contains all the dimensional keywords and is minimal, LD must be one of them.

Redundancy free. First, we prove that any two LDs, LD_1 and LD_2, generated from different LV*s* are different. Notice that LD_1 and LD_2 are rooted trees. LD_1 and LD_2 should have different roots because different LV*s* have different roots, and LDs generated by the same LV(R) have the same root R. They must be different. Second, we prove that any two LDs, LD_1 and LD_2, generated from the same LV are different. In line 10, LD_1 and LD_2 are generated from different combinations of attributes. Since the attributes in each combination are unordered, it is possible that LD_1 and LD_2 have the same set of attributes. In this situation, LD_1 and LD_2 are still considered to be different since the attributes are matched to different dimensional keywords and thus have different meanings. □

Theorem 4.2. *The time complexity of Algorithm 3 is* $O(|\mathcal{Q}| + \sum_{i=1}^{m} |\text{list}(d_i)| \cdot |V_R(G_S)|)$, *where* $m = |Q_d|$, $|\mathcal{Q}|$ *is the number of valid* LDs *generated, and* $|V_R(G_S)|$*is the number of relation nodes in* G_S.

We omit the proof which is given in [23].

When the schema of the database is complex, the number of valid LDs can be very large. In such a situation, a large number of tree isomorphism tests are needed to generate all LDs using the naive approach. Theorem 4.2 shows that using the two-step approach, we can generate all LDs using linear time w.r.t. the number of nodes in the label-graph G_S and the number of LDs generated in the worst case. The two-step approach saves much computational cost in tree isomorphism checking.

5 Evaluate ALL LDs

In this section, for a structural statistics keyword query $Q = (Q_d, Q_g, \alpha, \beta)$, we first give a naive approach followed by a new two-phase approach to compute structural statistics for all the groups under a given LD. As shown in Algorithm 1, we will compute all groups for every LD.

5.1 The Naive Approach

A naive approach to compute all groups under LD is to 1) compute all the Dtrees (for Q_d) and 2) determine the Gtrees (for Q_g). For item 1, it can be done using an existing algorithm. For item 2, we can expand from a Dtree computed to determine the corresponding DGtree. The naive approach is shown in Algorithm 4. In line 1, we generate all Dtrees that conform to LD, which can be done using an existing algorithm. For every Dtree generated, we compute the DGtree by expanding from each node in the Dtree to include all nodes it can reach which contain any of the keywords in Q_g. Lines 6-11 is for group-&-aggregate. Since all the DGtrees have been computed, for each tree T_i', we compute $\alpha(T_i')$ (line 7) and identify its dimensional values that contain the corresponding keywords in Q_d (line 8). Then, we compute its group score using the β function (lines 9-10). For simplicity, we use $\gamma.score = \beta(\gamma.score, T_i'.score)$, if β is min, max, sum and $count$. If β is avg we can do the same using both sum and $count$ values.

Algorithm 4. LD-Eval-Naive(Q, LD, G_D)
Input: A query $Q = \{Q_d, Q_g, \alpha, \beta\}$, LD and a database graph $G_D(V, E)$.
Output: aggregates for all groups

1: $\mathcal{T} \leftarrow$ all Dtree*s* computed that conform to the LD;
2: $\mathcal{T}' \leftarrow \emptyset$; $\Gamma \leftarrow \emptyset$;
3: **for all** Dtree $T_i \in \mathcal{T}$ **do**
4: compute DGtree T_i' by expanding T_i in G_D;
5: $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{T_i'\}$;
6: **for all** $T_i' \in \mathcal{T}'$ **do**
7: $T_i'.score \leftarrow \alpha(T_i')$ by considering T_i' as a virtual document in $\mathcal{T}'$ w.r.t. Q_g.
8: let a_i be the attribute value in the attribute $A_i \in \text{att}(\text{LD})$ that contains d_i, for all $d_i \in Q_d$;
9: let γ be a group represented by $(a_1, a_2, \ldots, a_{|Q_d|})$;
10: $\gamma.score = \beta(\gamma.score, T_i'.score)$;
11: $\Gamma \leftarrow \Gamma \cup \{\gamma\}$ if $\gamma \notin \Gamma$;
12: **return** Γ;

There are mainly two drawbacks for the naive approach. It needs to compute all DGtrees before group-&-aggregate, which is costly. The same subtree will be computed several times in computing DGtrees and conducting group-&-aggregate.

5.2 A New Two-Phase Approach

We propose a new two-phase approach, namely, bottom-up followed by top-down, after pruning unnecessary nodes/edges from G_D that need to evaluate an $\mathrm{LD}(R)$. For a given $\mathrm{LD}(R)$ to be evaluated, let $\mathrm{LV}(R)$ be the labeled LV that generates $\mathrm{LD}(R)$. In the bottom-up phase, we collect statistics for TF-IDF, namely, $tf_w(T)$ and idf_w (refer to the discussions related to (1)) using general-keywords. The statistics collection is done in a bottom-up fashion, and will finish when it finally evaluates the relation R which is the root of $\mathrm{LD}(R)$. Let R' be the set of tuples in R that contain statistics for at least one general-keyword in Q_g. At the end of this phase, we have a set of $\{\mathrm{Gtree}(t_\gamma)\}$ for t_γ in R', and we compute all $\alpha(\mathrm{Gtree}(t_\gamma))$. In the top-down phase, we start from those tuples t_γ in R'. We retrieve all leaf attribute nodes that t_γ can reach using the depth first search from t_γ. If the leaf attribute nodes contain all the required dimensional-keywords in Q_d, then t_γ has a valid $\mathrm{Dtree}(t_\gamma)$. At the end of this top-down process, we compute β for all such t_γ in R' that has a valid $\mathrm{Dtree}(t_\gamma)$.

Below, we first discuss the pruning process using dimensional keywords before the two-phase approach. Given a dimensional keyword d_i, a node v in G_D can be pruned by d_i if there does not exist a Dtree that contains both keyword d_i and node v. We call v an unnecessary node if v can be pruned by any dimensional keyword in the query. For example, in the database graph shown in Fig. 5, for dimensional keyword "Chicago," the node l_3 is unnecessary because all nodes that can reach city:Chicago cannot reach l_3. Given an LD, we consider whether a keyword $d_i \in Q_d$ has enough pruning power using $Pow(d_i, \mathrm{LD})$ which is computed as follows: suppose $X(A, B)$ is the join selectivity in relation A that can join a tuple in relation B, i.e., the fraction of tuples of A that have a matching tuple in B. Suppose the attribute in relation R_i in LD contains d_i, and assume P_i is the path from the root of LD to R_i. We have

$$Pow(d_i, \mathrm{LD}) = \frac{1}{R_i.c(d_i) \cdot \prod_{(A,B)\in P_i} X(A, B)}. \quad (2)$$

Here $R_i.c(d_i)$ is the number of tuples in R_i that contain the keyword d_i in the specified attribute of R_i in LD. The basic idea behind (2) is as follows: for any join sequence $A \bowtie B \bowtie C$, we assume that for any tuples $a \in A$, $b \in B$, and $c \in C$, whether a can be joined with b and whether b can be joined with c are independent to each other. $R_i.c(d_i) \cdot \prod_{(A,B)\in P_i} X(A, B)$ is the expected number of root nodes to reach a node with keyword d_i. Intuitively, a keyword that ends up with a smaller expected number of the root nodes has higher pruning power. For any general-keyword $g_j \in Q_g$, suppose for any text-attribute A_i, $R(A_i)$ is the relation of A_i, and $P(A_i)$ is the path from the root of LD to the relation $R(A_i)$. We compute the pruning power of a general-keyword g_i, $Pow(g_i, \mathrm{LD})$ as follows:

$$Pow(g_i, \mathrm{LD}) = \frac{1}{\sum_{A_i \in \mathrm{LD}} R(A_i).c(g_i) \cdot \prod_{(A,B)\in P(A_i)} X(A, B)}. \quad (3)$$

Based on (2) and (3), we decide whether a dimensional-keyword $d_i \in Q_d$ has enough power to prune, or in other words, it is cost-effective to reduce G_D. We sort all dimensional-keywords $d_i \in Q_d$ in decreasing order based on (2). If the pruning power of d_i is larger than the largest pruning power of all general-keywords in Q_g ((3)), then we use d_i to reduce G_D by removing all the tuple-nodes that cannot reach any attribute nodes containing d_i.

The algorithm is shown in Algorithm 5. We assume that the structure of the data graph is held in memory. First, we prune unnecessary nodes from G_D using Q_d if they have enough pruning power (lines 5-7). Second, in the bottom-up phase, we compute trees in a sense to collect all needed information to compute α for every Gtree using Q_g. It is done from the leaf toward the root which is a tuple in $r(R)$ for the $\mathrm{LV}(R)$ that generates the given LD (lines 9-25). Finally, in a top-down phase, we aggregate for each group based on Q_d (lines 27-31).

Algorithm 5. LD-Eval(Q, LD, G_D)
Input: A query $Q = \{Q_d, Q_g, \alpha, \beta\}$, LD and a database graph $G_D(V, E)$.
Output: aggregates for all groups
1: $\Gamma \leftarrow \emptyset$;
2: let $\mathrm{LV}(R)$ be the LV that generates LD;
3: **for all** relation node $P \in \mathrm{LV}(R)$ **do**
4: $P.set \leftarrow \emptyset$;
5: **for all** $d_i \in Q_d$ sorted by decreasing order of $Pow(d_i, \mathrm{LD})$ **do**
6: **if** $Pow(d_i, \mathrm{LD}) > max_{g_i \in Q_g}(Pow(g_i, \mathrm{LD}))$ **then**
7: $G_D \leftarrow$ prune(G_D, d_i, LD);
8: *// The bottom-up phase*
9: **for all** relation node $P \in \mathrm{LV}(R)$ in the order from leaves to the root **do**
10: **for** $i = 1$ **to** $|Q_g|$ **do**
11: **for all** tuple-node $t_P \in P.contain(g_i)$ **do**
12: $t_P.cnt_i \leftarrow t_P.cnt_i + t_P.count(g_i)$;
13: $P.has_i \leftarrow P.has_i \cup \{t_P\}$;
14: $P.set \leftarrow P.set \cup \{t_P\}$;
15: **for all** child of P, C, in $\mathrm{LV}(R)$ **do**
16: **for all** node $t_C \in C.set$ **do**
17: **for all** node $t_P \in P$ such that $t_P \rightarrow t_C \in E(G_D)$ **do**
18: $P.set \leftarrow P.set \cup \{t_P\}$;
19: **for** $i = 1$ **to** $|Q_g|$ **do**
20: $t_P.cnt_i \leftarrow t_P.cnt_i + t_C.cnt_i$;
21: **if** $t_C.cnt_i > 0$ **then** $P.has_i \leftarrow P.has_i \cup \{t_P\}$;
22: **for all** tuple-node $t \in R.set$ **do**
23: **for** $i = 1$ **to** $|Q_g|$ **do**
24: $tf_{g_i}(t) \leftarrow t.cnt_i$; $df_{g_i}(\mathrm{LV}(R)) \leftarrow |R.has_i|$;
25: $t.score \leftarrow \alpha(t)$ using all $tf_{g_i}(t)$ and $df_{g_i}(\mathrm{LV}(R))$;
26: *// The top-down phase*
27: **for all** $t \in R.set$ **do**
28: let a_i be the attribute value in the attribute $A_i \in \mathrm{att}(\mathrm{LD})$ that contains d_i, for all $d_i \in Q_d$;
29: let γ be a group represented by $(a_1, a_2, \cdots, a_{|Q_d|})$;
30: $\gamma.score = \beta(\gamma.score, t.score)$;
31: $\Gamma \leftarrow \Gamma \cup \{\gamma\}$ if $\gamma \notin \Gamma$;
32: **return** Γ;

33: **Procedure** prune(G_D, d_i, LD)
34: let R' be the relation node in LD that links to an attribute for d_i;
35: let S be all R'-labeled tuple-nodes that link to an attribute-node containing d_i in G_D;
36: let S' be all nodes in G_D that can reach at least one

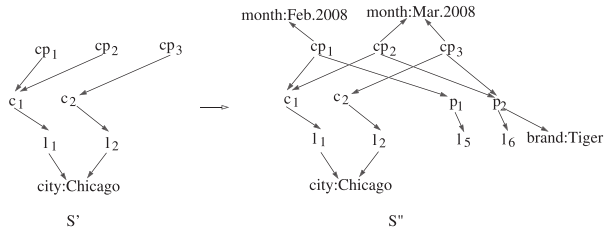


Fig. 9. Pruning using dimensional keyword "Chicago."

node in S;
37: let S'' be all nodes in G_D that at least one node in S' can reach;
38: **return** the reduced graph of S'' in G_D;

We use the following notations. P denotes a relation node in $\mathrm{LV}(R)$. $P.set$ denotes the set of P-labeled nodes in the Gtrees containing at least one keyword in Q_g, $P.has_i$ is a subset of $P.set$ including the nodes in the Gtrees that contain the keyword $g_i \in Q_g$. A tuple $t_P \in P.set$ is associated with a count, $t_P.cnt_i$, to record the number of times the keyword g_i appears in the Gtree rooted at t_P.

In Algorithm 5, we sort all dimensional-keywords $d_i \in Q_d$ in decreasing order based on (2) (line 5). If the pruning power is larger than the largest pruning power of all general-keywords in Q_g ((3)), then we can use it to reduce G_D (lines 6-7). The prune procedure is shown in lines 33-38, which returns a reduced subgraph of G_D. Fig. 9 shows an example to prune the database graph G_D shown in Fig. 5 using dimensional keyword "Chicago." S' represents the set of nodes in G_D that the node with label city:Chicago can reach. S'' represents the set of nodes in G_D that nodes in S' can reach. Any node $v \in V(G_D) - S''$ is useless (e.g., cp_7) because there is no root t that can reach both v and a node containing keyword "Chicago," and thus can be pruned.

The bottom-up phase are lines 9-25. We visit all relation nodes $P \in \mathrm{LV}(R)$ in a bottom-up fashion. For each P-labeled node, t_P, we maintain the counting information for general-keywords. Here, t_P contains g_i if g_i is contained in one of its text-attributes. This operation can be easily implemented using the techniques for full-text search engine in RDBMS by precomputing the full-text index on each text attribute of P. For t_P in $P.contain(g_i)$, $t_P.count(g_i)$ keeps the number of appearances of g_i in any text-attributes of t_P. Different from $t_P.count(g_i)$, $t_P.cnt_i$ keeps the number of appearances of g_i in the Gtree rooted at t_P. We add $t_P.count(g_i)$ into $t_P.cnt_i$ (line 12), and insert t_P into $P.set$ and $P.has_i$ (lines 13-14). Suppose C is a child

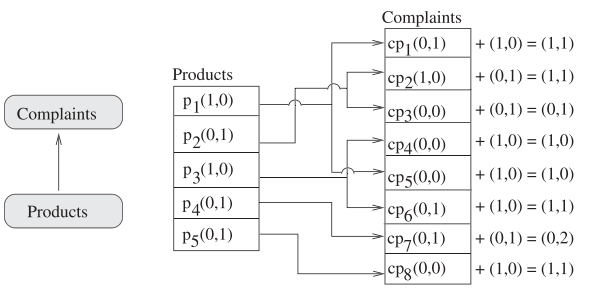


Fig. 10. The bottom-up phase.

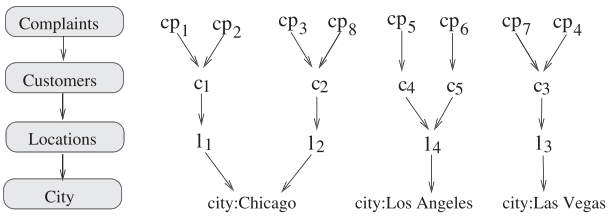


Fig. 11. The top-down phase.

of P, and all the counting information for the Gtrees rooted at nodes in C has been computed, because we process relation nodes in $\mathrm{LV}(R)$ in the order from leaf nodes to the root node. We update counting information for nodes in P using the information computed for nodes in C (lines 15-21). At the end of the bottom-up phase, the scores for all nodes in $R.set$ (the root of $\mathrm{LV}(R)$) can be computed using α (lines 22-25). Fig. 10 shows how to avoid computing all Gtrees when counting word frequencies for Gtrees rooted at Complaints, for the query with dimensional keywords $Q_g = \{\text{``}computer\text{''}, \text{``}monitor\text{''}\}$. The counter $p_i(j,k)$ denotes that in the Gtree rooted at p_i, "computer" appears j times and "monitor" appears k times. Such counters can be aggregated, in a bottom-up fashion, without traversal the whole tree from the root.

The top-down phase are lines 27-31. For each node t in $R.set$ computed in the bottom-up phase, we compute $\mathrm{Dtree}(t)$ that must have all the dimensional-keywords Q_d which must appear at specific attributes. We do it by searching along the path from t to the corresponding attribute nodes (line 28). Let γ be the group where t belongs to. We compute the aggregate function β using $t.score$ (line 30). To explain line 28, Fig. 11 shows an example on how to compute the "city" dimension for each virtual tuple rooted at the nodes in Complaints. Two paths may share subpaths. For each node in the root-to-leaf path, after computing its dimensional values, we add a pointer referring to its dimensional values directly, to avoid recomputing the dimension values for the same node appearing in other paths.

Cost saving in the bottom-up phase. We discuss cost sharing in the bottom-up phase using an example. In Fig. 12, the upper part shows how the naive algorithm (Algorithm 4) does. There are four Gtrees, rooted at cp_1, cp_2, cp_3, and cp_8, being computed individually without cost sharing. In our new two-phase algorithm, we compute the $tf_w(T)$ and idf_w

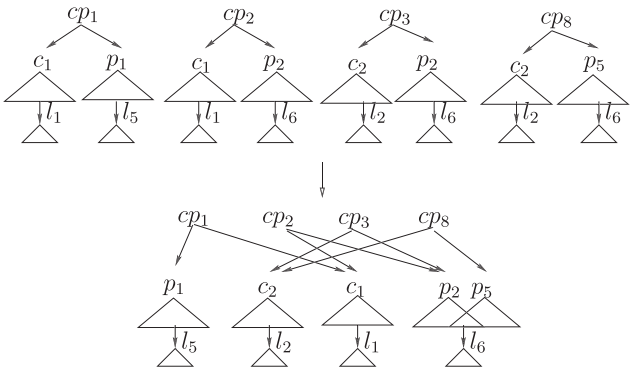


Fig. 12. Evaluate LD without generating all trees.

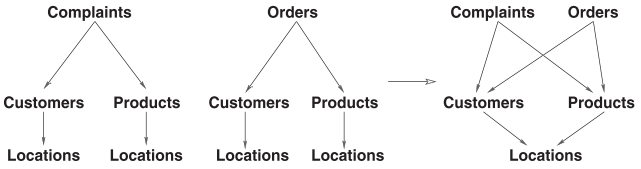


Fig. 13. Sharing computational cost cross LDs.

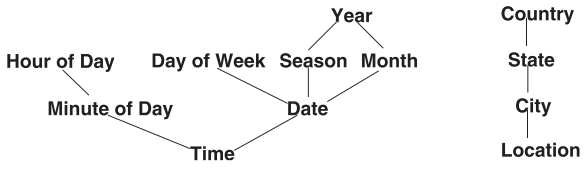


Fig. 14. Hierarchies for time and location attributes.

regarding $w \in Q_g$ from the leaf nodes to the roots. Suppose we have computed such $tf_w(T)$ and idf_w for the Locations-labeled nodes. The Locations-labeled node, l_6, is shared by both Gtrees rooted at p_2 and p_5, we pass the information computed on l_6 to p_2 and p_5 to achieve cost-sharing. In a similar way, the information on p_2 is further passed to cp_2 and cp_3, and thus the computational cost for computing the subtree rooted at p_2 is minimized.

Theorem 5.1. *Let $|P|$ be the number of tuples in P, N be the number of nodes in* $\mathrm{LV}(R)$, $M = max\{|P|$ *for any relation node P in $G_S\}$, and $m = |Q_d|$ and $n = |Q_g|$. The time complexity for the two-phase algorithm is $O((n+m) \cdot N \cdot M)$.*

We omit the proof which is given in [23].

Cost sharing cross LDs. In addition, for data sets with an acyclic schema graph, when there are many LDs computed, it is costly to evaluate each LD individually. LDs from $\mathrm{LV}(R)$ and $\mathrm{LV}(R')$ may share subtrees. As shown in Fig. 13, when evaluating the two LVs, LV(Complaints) and LV(Orders), they share LV(Customers) and LV(Products). Therefore, the information computed in the bottom-up phase for Customers, and Products can all be reused for computing the Gtrees for LV(Complaints) and LV(Orders). It can be done by adding the following before the start of the bottom-up phase: if $\mathrm{LV}(P)$ has been evaluated before, skip it and continue for the next node. Here, P is a relation node in $\mathrm{LD}(R)$ which represents a subtree rooted at P in $\mathrm{LD}(R)$.

6 OLAP

In this section, we discuss how to allow OLAP operations, such as roll up or drill down, to be conducted on the existing structural statistics keyword query result efficiently. In brief, for a structural statistics keyword query, after the initial result Γ has been computed, we allow users to incrementally change dimensions without recomputing the new Γ from scratch. We discuss four of the operations on dimensions, namely, roll up on a dimension, drill down on a dimension, insertion of a dimension, and deletion of a dimension.

Roll up/drill down on a dimension. In order to enable roll up/drill down operations, we assume there exist hierarchies on dimensional attributes. For example, Fig. 14 shows the hierarchies for the Time and Location attributes, respectively. Given a certain value v in a dimensional attribute, and a dimension dim in the corresponding dimension hierarchy, we use a function $f(v, dim)$ to obtain the value for the dimension dim of the value v. For example, f("2011-10-01 10:00:00," year) returns 2011 which is the value of the year dimension of "2011-10-01 10:00:00." For a roll up operation on a certain dimension, we only need to traverse all groups in the original result Γ, and recalculate the value for the dimension using the function $f()$. For each group in Γ, we put it into the corresponding larger group, followed by updating the score of the larger group using $\beta()$. The drill down operation is more complex, we need to reuse the information of elements in $R.set$, computed in the bottom-up phase, for the root relation R of the corresponding LD. For each element t in $R.set$, we do not need to compute the value of all its dimensions, because they have been computed in the top-down phase. We only need to compute its dimension value regarding the dimension to be drilled down. After that, we need to perform group-&-aggregate on every element in $R.set$. The algorithms for roll up/drill down operations are shown in Algorithm 6.

Algorithm 6. dim-operation(Q, LD, G_D, Γ, o)
Input: The original query $Q = \{Q_d, Q_g, \alpha, \beta\}$, LD! a database graph $G_D(V, E)$, the original result Γ, the dimension operation $o =$ (type,old-dim, new-dim).
Output: new result Γ'.
1: $\Gamma' \leftarrow \emptyset$;
2: let $\mathrm{LV}(R)$ be the LV that generates LD;
3: **case** o.type:
4: roll-up: $\Gamma' \leftarrow$ roll-up(o.old-dim,o.new-dim);
5: drill-down: $\Gamma' \leftarrow$ drill-down(o.old-dim,o.new-dim);
6: insertion: $\Gamma' \leftarrow$ insertion(o.new-dim);
7: deletion: $\Gamma' \leftarrow$ deletion(o.old-dim);
8: **return** Γ';
9: **Procedure** roll-up(old-dim dim_i, new-dim dim_i')
10: **for all** $\gamma = (a_1, a_2, \ldots, a_{|Q_d|}) \in \Gamma$ **do**
11: $a_i' \leftarrow f(a_i, dim_i')$;
12: let γ' be a group represented by $(a_1, \ldots, a_i', \ldots, a_{|Q_d|})$;
13: $\gamma'.score = \beta(\gamma'.score, \gamma.score)$;
14: $\Gamma' \leftarrow \Gamma' \cup \{\gamma'\}$ if $\gamma' \notin \Gamma'$;
15: **return** Γ';
16: **Procedure** drill-down(old-dim dim_i, new-dim dim_i')
17: **for all** $t \in R.set$ **do**
18: $a_i' \leftarrow f(t.a_i, dim_i')$;
19: let γ' be a group represented by $(t.a_1, \ldots, a_i', \ldots, t.a_{|Q_d|})$;
20: $\gamma'.score = \beta(\gamma'.score, t.score)$;
21: $\Gamma' \leftarrow \Gamma' \cup \{\gamma'\}$ if $\gamma' \notin \Gamma'$;
22: **return** Γ';
23: **Procedure** insertion(new-dim dim_i')
24: **for all** $t \in R.set$ **do**
25: let a_i' be the attribute value in the attribute $A_i \in \mathrm{att}(\mathrm{LD})$ that contains dim_i';
26: let γ' be a group represented by $(t.a_1, \ldots, t.a_i, a_i', t.a_{i+1}, \ldots, t.a_{|Q_d|})$;
27: $\gamma'.score = \beta(\gamma'.score, t.score)$;
28: $\Gamma' \leftarrow \Gamma' \cup \{\gamma'\}$ if $\gamma' \notin \Gamma'$;
29: **return** Γ';
30: **Procedure** deletion(old-dim dim_i)

31: **for all** $\gamma = (a_1, a_2, \ldots, a_{|Q_d|}) \in \Gamma$ **do**
32: let γ' be a group represented by $(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_{|Q_d|})$;
33: $\gamma'.score = \beta(\gamma'.score, \gamma.score)$;
34: $\Gamma' \leftarrow \Gamma' \cup \{\gamma'\}$ if $\gamma' \notin \Gamma'$;
35: **return** Γ';

Insertion/deletion of a dimension. The insertion of a new dimension can be considered as a drill down operation on a new dimension originally set to be the whole space. In details, we traverse elements in $R.set$ for the root relation R, and calculate their values for the new dimension in a top-down fashion. For each element t in $R.set$, we only need to perform group-&-aggregate using the scores of t calculated in the bottom-up phase. The deletion of a dimension is straightforward. We do not need to traverse elements in $R.set$, but only need to traverse the existing groups in Γ. We remove the dimension to be deleted for all groups in Γ and regroup them to the new larger groups before calculating the new aggregate values using $\beta()$. The algorithms for insertion/deletion operations are shown in Algorithm 6.

For other operations, such as slice and dice, each of them returns a subset of Γ, by specifying the values of some dimensions in Γ. We just need to remove those invalid groups from Γ by checking each of their dimensions in the slice/dice operation.

Example 6.1. Given dimensional keywords {"Chicago," "Month"}, suppose we perform a roll up operation from "Month" to "Year," for the LD shown in Fig. 8c. Using Algorithm 6, instead of processing the new query from scratch, we can reuse the result for the query with dimensional keywords {"Chicago," "Month"} in the group level. For example, for the group $\gamma' =$ ("Chicago," "2008"), two groups in the previous result that fall in the new group are $\gamma_1 =$ ("Chicago," "Feb. 2008") and $\gamma_2 =$ ("Chicago," "Mar. 2008"). We have $\gamma'.score = \beta(\gamma_1.score, \gamma_2.score)$. Suppose we perform a drill down operation from "Year" to "Month." We can reuse the result for all scores calculated in the bottom-up phase in the virtual tuple level. For example, for a group $\gamma' =$ ("Chicago," "Mar. 2008") to be computed, after the bottom-up phase, two virtual tuples t_1 and t_2 fall into the new group. We have $\gamma'.score = \beta(t_1.score, t_2.score)$. The cost for computing all virtual tuples and the scores and dimensions for each virtual tuple can be saved using Algorithm 6.

7 Related Work

Keyword-based aggregation on databases. Wu et al. [28] studied the keyword-driven OLAP (KDAP). The aggregation involved in KDAP is based on some predefined measures that are independent of keyword relevance. Tata et al. in [27] integrated the aggregate function and the numerical attribute to be aggregated into a keyword query. They focus on translating a user query into a set of proper interpretations. Bhide et al. in [4] studied keyword search over dynamic categorized information. In [4], categories have only one dimension and are all predefined based on the set of documents to be searched. Zhou et al. in [30] studied keyword-based aggregation using minimal group-bys. Tao and Yu in [26] proposed algorithms to find frequent cooccurring terms in relational keyword search.

Multidimensional search on text databases. Mothe et al. [21] proposed a user interface to provide users a global visualization of a large document collection. Inokuchi et al. [13] proposed a data representation and algebra operations to analyze large sets of textual documents with metadata. Simitsis et al. [25] proposed a Multidimensional Content eXploration (MCX) system, which is to effectively analyze and explore large amount of content by combining keyword search with OLAP-style aggregation. A Text-Cube model was proposed by Lin et al. [17] to study effective OLAP over text databases. Topic Cube was proposed by Zhang et al. [29] to combine OLAP with probabilistic topic modeling and enable OLAP on the dimension of general text data. It is worth noting that, for all of these works, the dimensions are systematically defined, and no keyword is used to specify the user-wanted dimensions.

Keyword search on relational databases. In the literature, for a keyword query on a relational database, it returns a set of interconnected structures in the *RDB* that contain the user given keywords. The techniques to answer keyword queries in *RDB*s are mainly in two categories: *CN*-based (schema-based) and graph-based (schema-free) approaches.

In the *CN*-based approaches [1], [12], [10], [19], [20], it processes a keyword query in two steps, namely, candidate network (*CN*) generation and *CN* evaluation using SQL on RDBMSs. The ranking issues are also discussed in [2], [11], [18].

Finding top-k interconnected structures has been extensively studied in the graph-based approaches in which an *RDB* is materialized as a weighted database graph $G_D(V, E)$. The representative works on finding top-k connected trees are [3], [14], [15], [6], [8]. In brief, finding the exact top-k connected-trees is an instance of the group Steiner tree problem [7], which is NP-hard. Top-k connected trees are hard to compute, the distinct root semantics are studied in [9], and [5], the r-radius Steiner graph is studied in [16], and the multicenter communities under the distinct core semantics is studied in [24].

It is worth noting that all the above works focus on finding individual keyword search results and no aggregation issues are involved.

8 Performance Studies

We conducted extensive performance studies. We implemented two algorithms, denoted Naive and New. Both follow the framework in Algorithm 1, and generate all LDs using Algorithm 3. For the Naive algorithm, we evaluate LDs using Algorithm 4. For the New algorithm, we evaluate LDs using Algorithm 5. Techniques for sharing cost across LDs are allowed in both Naive and New. Both the dimensional inverted index and the full-text index for general keywords are computed in memory, and maintained on disk. We only keep the offsets and the lengths of the inverted lists in memory. For each dimensional/general keyword, we retrieve the matched tuples from the disk in query processing, and such time is included in the total processing time of each test case in our experiments. All algorithms were implemented in Visual

TABLE 1
Keywords Used for the *DBLP* Data Set

Key Freq	Keywords
1	flexible resource pattern
2	communication optimal implementation
3	application knowledge dynamic programming method
4	parallel web performance
5	information data system

TABLE 2
Keywords Used for the *IMDB* Data Set

Key Freq	Keywords
1	radio dance place
2	black Kevin manager
3	part Smith driver party music
4	animation captain Martin
5	Jack Tom Paul

C++ 2008 and all tests were conducted on a 2.8 GHz CPU and 2 GB memory PC running Windows XP.

We use two large real data sets, *DBLP* (http://www.informatik.uni-trier.de/~ley/db/) and *IMDB* (http://www.imdb.com/interfaces), and a large synthetic data set *TPC-H* (http://www.tpc.org/tpch/) for testing. The time to construct the Dimensional Inverted Index for *DBLP*, *IMDB*, and *TPC-H* are 4.5, 12.4, and 16.9 minutes, respectively, and the time to construct the full-text index for general keywords for *DBLP*, *IMDB*, and *TPC-H* are 5.5, 15.3, and 21.3 minutes, respectively. We tested several queries with different keyword frequencies. We use the aggregation function sum as β, and the TF-IDF score function α introduced in [19] in the experiments.

For *DBLP*, the schema includes the following four relations: Papers(Paperid, Conference, Year, Title), Authors (Authorid, Authorname, Position, Affiliation, University, Interest), Write(Writeid, Authorid, Paperid), and Cite(Citeid, Paperid1, Paperid2). The primary key for each relation is underlined. The size of the raw data for *DBLP* is 643 MB. The numbers of tuples for the four relations are 2,045,150, 758,839, 4,033,272, and 112,435, respectively, and the total number of tuples in *DBLP* is 6,949,696.

The *IMDB* schema includes eight relations: Actors(Actorid, Gender, Actorname), Act(Actid, Actorid, Movieid, Charactor), Directors (Directorid, Directorname), Direct (Directid, Directorid, Movieid), Movies(Movieid, Genreid, Languageid, Locationid, Year, Name), Genres(Genreid, Genre), Languages(Languageid, Language), and Locations (Locationid, Location, Area, City, State, Country). The primary key for each relation is underlined. The size of the raw data for *IMDB* is 920 MB. The numbers of tuples in the eight relations are 1,774,431, 12,514,476, 180,702, 1,026,444, 1,501,623, 28, 315, and 63,382, respectively, and the total number of tuples in *IMDB* is 17,061,401.

For *TPC-H*, the schema includes the following eight relations: Part, Supplier, Partsupp, Customer, Nation, Lineitem, Region, Orders. There are totally 62 attributes in the eight relations. The size of the raw data for the *TPC-H* data set is 1,005 MB. The numbers of tuples for the eight relations are 200,000, 10,000, 800,000, 150,000, 25, 6,001,215, 5, and 1,500,000, respectively, and the total number of tuples in *TPC-H* is 8,661,245.

For the scalability testings, we report the processing time and memory consumption for each test case. The processing time includes the time for generating and evaluating all LDs for the query. The memory consumption is the memory used to process each query. For each data set, we select representative queries with different keyword frequencies as follows: after removing all the stop words, we set the maximum keyword frequency among all keywords as τ, and divide the keyword frequency range between 0 and τ into five partitions, namely, $\tau/5$, $2\tau/5$, $3\tau/5$, $4\tau/5$, and τ. For simplicity, we say a keyword has frequency p ($p \in \{1, 2, 3, 4, 5\}$), if and only if its frequency is between $(p-1)\cdot\tau/5$ and $p\cdot\tau/5$.

For all scalability testings, we vary three parameters, namely, the keyword frequency, the number of keywords, and the number of dimensions. The keyword frequency is the frequency for each general-keyword in the query. The number of keywords is the number of general-keywords used in a query, and the number of dimensions is the number of dimensional-keywords used in a query. Every parameter has a default value. For *DBLP*, the general-keywords selected with different keyword frequencies are shown in Table 1 and by default, the keyword frequency is 3. The number of keywords ranges from 1 to 5 with a default value 3 and the number of dimensions ranges from 1 to 4 with a default value 2. The set of dimensional-keywords are selected from {"conference," "year," "authorname," "20th"} and by default it is {"conference," "year"}. For *IMDB*, the keyword frequency ranges from 1 to 5 with a default value 3. The general-keywords selected with different keyword frequencies are shown in Table 2. The number of keywords ranges from 1 to 5 with a default value 3, and the number of dimensions ranges from 1 to 5 with a default value 3. When varying number of keywords k, we select the first k keywords from the five keywords shown in the third line of Table 2. The dimensional-keywords from all testings in *IMDB* are selected from {"year," "genre," "USA," "male," "city"} and by default it is {"year," "genre," "USA"}. When varying the number of dimensions d, we select the first d keywords from the 5D keywords. For *TPC-H*, the settings are all similar to *IMDB* and *DBLP*, and the default percent of data set is 100 percent. For *TPC-H*, we also vary the size of the data set by selecting a certain percent of virtual tuples from the original data set. We vary the percent from 20 to 100 percent.

Exp-1 (effectiveness testing). In addition to the six examples in Section 1, we analyze another two representative queries, Qa and Qb against *DBLP*, and another three representative queries, Qc, Qd, and Qe, against *IMDB*. In each query, the dimensional-keywords and general-keywords are separated by a semicolon.

(**Qa**) {"conference," "year," "relational," "database," "management"} asks "in which conference and year, there are most papers about relational database management" in *DBLP*. Under the LD consisting of edges (Papers $\rightarrow$ conference, Papers $\rightarrow$ year), the groups with highest scores are (SIGMOD, 2005, 41.8), (SIGMOD, 1982, 33.8), and (SIGMOD, 1979, 27.2). Under the LD consisting of edges (Write $\rightarrow$ Papers, Papers $\rightarrow$ conference, Papers $\rightarrow$ year), the groups with highest scores are (TODS, 1976, 199.8),

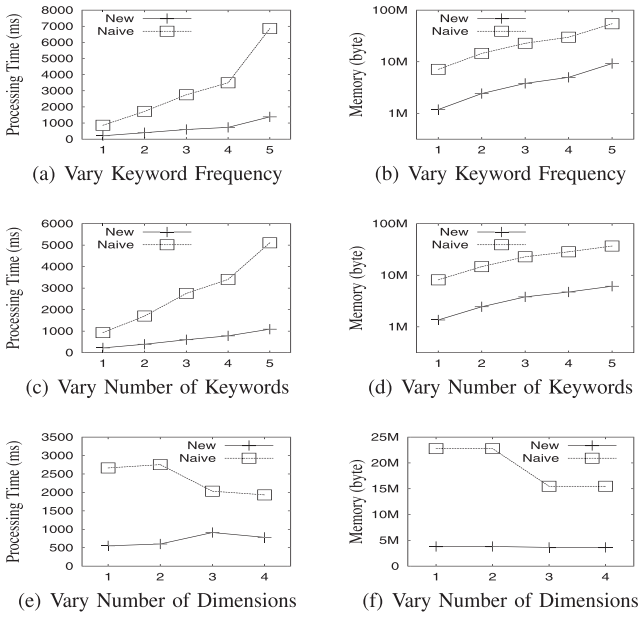


Fig. 15. Testing results for the *DBLP* Data Set.

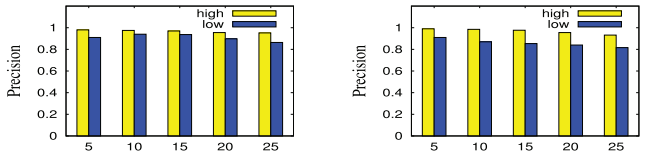


Fig. 16. Accuracy testing. (a) Vary top-k(*DBLP*). (b) Vary top-k(*IMDB*).

(SIGMOD, 2005, 96.7), and (VLDB, 2005, 95.4). After searching the *DBLP* website, we found that SIGMOD '05, SIGMOD '82, SIGMOD '79, TODS '76, and VLDB '05 all have many papers about "relational database management." Under the second LD, TODS '76 has the highest score because the paper "System R: Relational Approach to Database Management" in TODS '76 has as many as 14 authors.

(**Qb**) {"authorname," "2000"; "subgraph," "isomorphism"} asks "which author in the year 2000 has most papers about subgraph isomorphism" in *DBLP*. Under the LD consisting of edges (Write → Authors, Authors → authorname, Write → Papers, Papers → year), the groups with highest scores are (Ruth V. Spriggs, 2000, 50.0), (Shuichi Ichikawa, 2000, 33.3), and (Lerdtanaseangtham Udorn, 2000, 33.3). After searching the *DBLP* website, we found that all of the three authors have more than two papers about subgraph isomorphism in the year 2000.

(**Qc**) {"city," "USA"; "musical"} asks "for which city in US, there are most musical movies" in *IMDB*. Under the LD consisting of edges (Movies → Locations, Locations → city, Locations → country), the groups with highest scores are (New York City, US, 528.5), (Los Angeles, US, 492.0), (Culver City, US, 68.1), and (Las Vegas, US, 62.4). It indicates that most musical movies in US are from the New York City and Los Angeles. Under the LD consisting of edges (Act → Movies, Movies → Locations, Locations → city, Locations → country), the groups with highest scores are (New York City, US, 11691.3), (Los Angeles, US, 9998.2), (Culver City, US, 2390.0), and (Big Bear Lake, US, 1562.1). Here, the group (Big Bear Lake, US) has a higher rank because movies in Big Bear Lake have more actors than movies in Las Vegas.

(**Qd**) {"language," "gender," "genre"; "sing," "song"} asks "for which language, gender, and genre, there are most movies about sing and song" in *IMDB*. Under the LD consisting of edges (Act → Actors, Actors → gender, Act → Movies, Movies → Genres, Genres → genre, Movies → Languages, Languages → language), the groups with highest scores are (English, male, musical, 1421.1), (English, male, comedy, 1186.9).

(**Qe**) {"action," "actorname"; "spider"} asks "which actor acts most action movies about spider" in *IMDB*. Under the LD consisting edges (Act → Actors, Actors → actorname, Act → Movies, Movies → Genres, Genres → genre), the groups with highest scores are (action, Tobey Maguire, 79.9), (action, Bruce Campbell, 37.4), and (action, Kirsten Dunst, 37.4). The actor Tobey Maguire has the highest score because he is an actor in the movies from "Spider-Man 1" to "Spider-Man 5" and he himself acts the spider-man in the movies. The other persons are all the actors for the "Spider-Man" movies.

Exp-2 (accuracy testing). In our accuracy testing, for each data set, we select 40 queries. We divide the 40 queries into two groups according to the average keyword frequency for dimensional keywords in each query. Each group has 20 queries. We denote the two groups as low and high, respectively. For each query, we use the top-k precision, to denote the accuracy of each query. The top-k precision is calculated as the ratio of the number of relevant answers in the top-k results of a query. The top-k results of a query are the k groups with highest scores. Whether a result is relevant to a query is labeled manually by the researchers in our database group. Since the number of groups returned are usually small, we vary k from 5 to 25. The testing results are shown in Fig. 16. All precisions are calculated as the average precision of all the 20 queries in the corresponding query group.

Fig. 16a shows the accuracy testing for *DBLP*. When k increases, the precision for both low and high queries has a trend to decrease. This is because when k is large, results with low scores have high probabilities to be irrelevant. The precision for the low queries is smaller than the precision for the high queries, because in high queries, the textual information for each virtual tuple is rich, thus the statistics calculated are more accurate than those with little textual information. Fig. 16b shows the accuracy testing for *IMDB*. The results are similar to *DBLP*. All precisions are no less than 0.8 in both data sets.

Exp-3 (scalability testing for *DBLP*). Figs. 15a and 15b show that, when the keyword frequency increases in *DBLP*, the time/memory consumption for both Naive and New increases. New consumes six times less CPU and 10 times less memory than Naive. In Figs. 15c and 15d, when the number of general-keywords increases, New also performs much better than Naive. When increasing the number of dimensions, as shown in Fig. 15e and 15f, the time/memory consumption for New is consistent, but the time/memory consumption for Naive has a trend to decrease. This is because for this query, after adding a certain dimensional-keyword (e.g., authorname), some of the previously generated LDs do not contain any attribute that can match the new dimensional-keyword. As a result, the total number of LDs generated decreases,

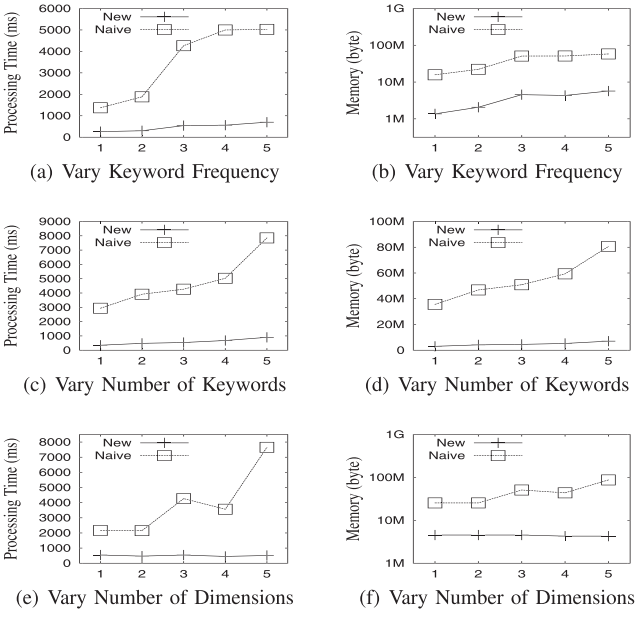


Fig. 17. Testing results for the *IMDB* Data Set.

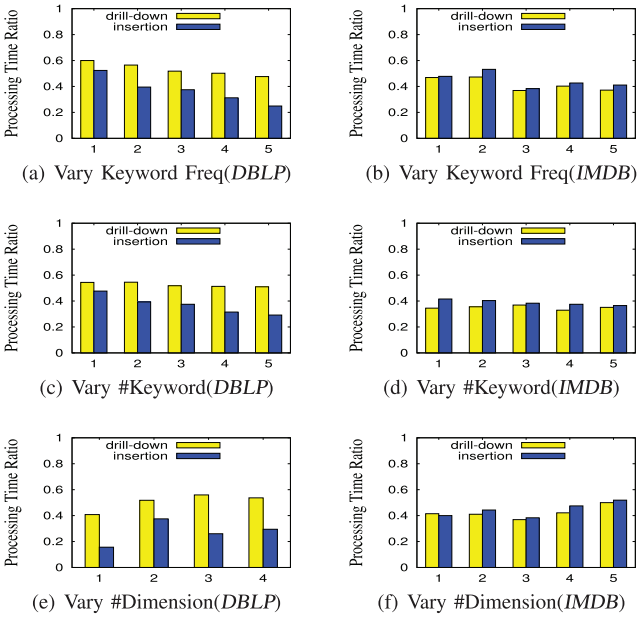


Fig. 19. Testing dimension operations.

which causes the time/memory consumption decreasing for Naive. New also performs much better than Naive in all cases.

Exp-4 (scalability testing for *IMDB*). The testing results for *IMDB* are shown in Fig. 17. The similar trends are observed. In Figs. 17e and 17f, we increase the number of dimensions by adding another dimensional-keyword each time. The processing time/memory consumption for Naive has a trend to increase. In this case, when the number of dimensions increases, more LDs will be generated, because some of the dimensional-keywords such as "USA" and "city" can be matched to many attributes in the relations. After adding the dimensional-keyword "male," the processing time/memory consumption decreases because the constraint "male" on the gender attribute can decrease the number of tuples matched in the final result, thus the time for aggregating and score computing decreases. The performance of New is also much better than Naive and is consistent when the number of dimensions increases, because much computational cost is shared when evaluating LD*s*.

Exp-5 (scalability testing for *TPC-H*). We conducted testing for *TPC-H*. The curves for *TPC-H* for both algorithms are all similar to the curves in *DBLP* and *IMDB*. The processing time and memory consumption when varying the database size from 20 to 100 percent are shown in Figs. 18a and 18b, respectively. The processing time for both Naive and New grows linearly w.r.t. the size of the database. This is because the time complexity for both algorithms are linear w.r.t. the database size. New runs much faster than Naive in all cases. The memory consumption for New is 10 times larger than Naive in all cases.

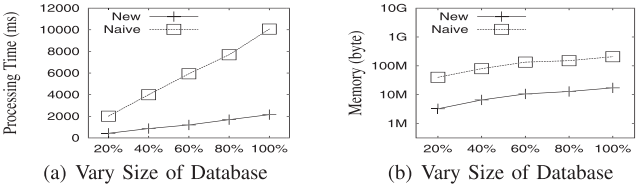


Fig. 18. Testing results for the *TPC-H* Data Set.

Exp-6 (testing OLAP operations). We test the four operations introduced in Section 6 by varying keyword frequencies, keyword numbers, and dimension numbers, in the two real data sets *DBLP* and *IMDB*. For each testing case, we record the processing time ratio which is computed using $t_{incremental}/t_{recompute}$, where $t_{incremental}$ is the processing time of the corresponding operation using Algorithm 6, and $t_{recompute}$ is the processing time of constructing a new query and processing the new query using Algorithm 5. For the general keywords, the default values and the variations are the same as before. In *DBLP*, we select the dimensional keywords from {"year," "conference," "authorname," "20th"}. For the roll up/drill down operations, we use "year" $\leftrightarrow$ "century," and for the insertion/deletion operations, we insert/delete a dimensional keyword "2000." In *IMDB*, we select the dimensional keywords from {"city," "year," "male," "USA," "gender"}. For the roll up/drill down operations, we use "city" $\leftrightarrow$ "country," and for the insertion/deletion operations, we insert/delete a dimensional keyword "genre."

The testing results are shown in Fig. 19. We show the processing time ratio for the drill down and dimension insertion operations. Figs. 19a and 19b show the efficiency when varying keyword frequency in the two data sets. The cost saving is large when the keyword frequency is large, this is because the cost in the bottom-up phase to process the general keywords is high for keywords with high frequency. In *DBLP*, the insertion operation performs much better than drill-down, because the dimension value "2000" we inserted has high-pruning power. Figs. 19c and 19d show the situation when varying the number of keywords in the two data sets. The performance does not vary much when increasing the number of general keywords. This is because the extra cost of adding a general keywords in our algorithm is not much. Figs. 19e and 19f show the results by varying the number of dimensions in the query. The average cost saving is above 0.5. When the number of dimensions is

small, the cost saving is significant. This is because when the number of dimensions is small, the cost on the top-down traversal is small, and thus we need small cost on computing the new dimensions.

For the roll up and dimension deletion operations, the processing time is less than 10 ms for every testing case. This is because for the roll up and dimension deletion operations, as shown in Algorithm 6, we do not need to traverse every virtual tuple in the data set. We only need to visit every group in the original result followed by assigning each group into new groups.

9 Conclusion

In this paper, we studied a new keyword query, $Q = (Q_d, Q_g, \alpha, \beta)$, to compute structural statistics for all groups of tuple-trees in an *RDB*. We represent the *RDB* as a data graph G_D. We show how to compute Dtrees, as m-dimensional objects, that must contain all the keywords in Q_d, for $m = |Q_d|$, over G_D. We also show how to compute Gtrees, that contain keywords in Q_g and are strongly related to an m-dimensional object. We discuss how to group m-dimensional objects into groups without tree isomorphism testing, using the label-trees (LDs) based on the database schema. We present a two-step approach, with a new algorithm to compute all label-trees for a structural statistics keyword query, and a new two-phase algorithm to compute group-&-aggregate using the label-trees computed. We discussed cost sharing, when computing a single label-tree, and cost sharing, when computing cross several label-trees. We analyze our algorithm and provide time complexity. We conducted extensive experimental studies using two large real data sets, *IMDB* and *DBLP* and a synthetic data set *TPC-H*, and confirmed the effectiveness and efficiency of our approach.

Acknowledgments

The work was supported by grant of the Research Grants Council of the Hong Kong SAR, China No. CUHK/419109.

References

[1] S. Agrawal, S. Chaudhuri, and G. Das, "DBXplorer: A System for Keyword-Based Search over Relational Databases," *Proc. 18th Int'l Conf. Data Eng. (ICDE '02),* 2002.

[2] A. Balmin, V. Hristidis, and Y. Papakonstantinou, "ObjectRank: Authority-Based Keyword Search in Databases," *Proc. 13th Int'l Conf. Very Large Data Bases (VLDB '04),* 2004.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword Searching and Browsing in Databases Using BANKS," *Proc. 18th Int'l Conf. Data Eng. (ICDE '02),* 2002.

[4] M. Bhide, V.T. Chakaravarthy, K. Ramamritham, and P. Roy, "Keyword Search over Dynamic Categorized Information," *Proc. IEEE Int'l Conf. Data Eng. (ICDE '09),* pp. 258-269, 2009.

[5] B.B. Dalvi, M. Kshirsagar, and S. Sudarshan, "Keyword Search on External Memory Data Graphs," *Proc. VLDB Endowment,* vol. 1, no. 1, pp. 1189-1204, 2008.

[6] B. Ding, J.X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin, "Finding Top-K Min-Cost Connected Trees in Databases," *Proc. Int'l Conf. Data Eng. (ICDE '07),* 2007.

[7] S.E. Dreyfus and R.A. Wagner, "The Steiner Problem in Graphs," *Networks,* 1972.

[8] K. Golenberg, B. Kimelfeld, and Y. Sagiv, "Keyword Proximity Search in Complex Data Graphs," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '08),* 2008.

[9] H. He, H. Wang, J. Yang, and P.S. Yu, "BLINKS: Ranked Keyword Searches on Graphs," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '07),* 2007.

[10] V. Hristidis, L. Gravano, and Y. Papakonstantinou, "Efficient IR-Style Keyword Search over Relational Databases," *Proc. 29th Int'l Conf. Very Large Data Bases (VLDB '03),* 2003.

[11] V. Hristidis, H. Hwang, and Y. Papakonstantinou, "Authority-Based Keyword Search in Databases," *ACM Trans. Database Systems,* vol. 33, no. 1, article 1, 2008.

[12] V. Hristidis and Y. Papakonstantinou, "DISCOVER: Keyword Search in Relational Databases," *Proc. 28th Int'l Conf. Very Large Data Bases (VLDB '02),* 2002.

[13] A. Inokuchi and K. Takeda, "A Method for Online Analytical Processing of Text Data," *Proc. 16th ACM Conf. Information and Knowledge Management (CIKM '07),* pp. 455-464, 2007.

[14] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, "Bidirectional Expansion for Keyword Search on Graph Databases," *Proc. 31st Int'l Conf. Very Large Data Bases (VLDB '05),* 2005.

[15] B. Kimelfeld and Y. Sagiv, "Finding and Approximating Top-K Answers in Keyword Proximity Search," *Proc. 25th ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems (PODS '06),* 2006.

[16] 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," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '08),* 2008.

[17] C.X. Lin, B. Ding, J. Han, F. Zhu, and B. Zhao, "Text Cube: Computing ir Measures for Multidimensional Text Database Analysis," *Proc. IEEE Eighth Int'l Conf. Data Mining (ICDM '08),* pp. 905-910, 2008.

[18] F. Liu, C.T. Yu, W. Meng, and A. Chowdhury, "Effective Keyword Search in Relational Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '06),* 2006.

[19] Y. Luo, X. Lin, W. Wang, and X. Zhou, "Spark: Top-K Keyword Query in Relational Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '07),* 2007.

[20] A. Markowetz, Y. Yang, and D. Papadias, "Keyword Search on Relational Data Streams," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '07),* 2007.

[21] J. Mothe, C. Chrisment, B. Dousset, and J. Alau, "Doccube: Multi-Dimensional Visualisation and Exploration of Large Document Sets," *J. Am. Soc. for Information Science and Technology,* vol. 54, no. 7, pp. 650-659, 2003.

[22] L. Qin, J.X. Yu, and L. Chang, "Keyword Search in Databases: The Power of Rdbms," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '09),* 2009.

[23] L. Qin, J.X. Yu, and L. Chang, "Computing Structural Statistics by Keyword in Databases," *Proc. IEEE 27th Int'l Conf. Data Eng. (ICDE '11),* 2011.

[24] L. Qin, J.X. Yu, L. Chang, and Y. Tao, "Querying Communities in Relational Databases," *Proc. IEEE Int'l Conf. Data Eng. (ICDE '09),* 2009.

[25] A. Simitsis, A. Baid, Y. Sismanis, and B. Reinwald, "Multi-dimensional Content Exploration," *Proc. VLDB Endowment,* vol. 1, no. 1, pp. 660-671, 2008.

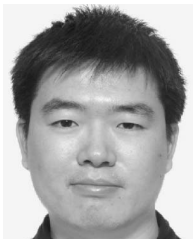
[26] Y. Tao and J.X. Yu, "Finding Frequent Co-Occurring Terms in Relational Keyword Search," *Proc. 12th Int'l Conf. Extending Database Technology: Advances in Database Technology (EDBT '09),* pp. 839-850, 2009.

[27] S. Tata and G.M. Lohman, "Sqak: Doing More with Keywords," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '08),* pp. 889-902, 2008.

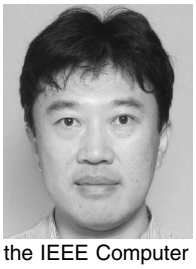
[28] P. Wu, Y. Sismanis, and B. Reinwald, "Towards Keyword-Driven Analytical Processing," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '07),* pp. 617-628, 2007.

[29] D. Zhang, C. Zhai, and J. Han, "Topic Cube: Topic Modeling for Olap on Multidimensional Text Databases," *Proc. Int'l Conf. Data Mining (SDM '09),* pp. 1123-1134, 2009.

[30] B. Zhou and J. Pei, "Answering Aggregate Keyword Queries on Relational Databases Using Minimal Group-Bys," *Proc. 12th Int'l Conf. Extending Database Technology: Advances in Database Technology (EDBT '09),* pp. 108-119, 2009.



Lu Qin received the BE degree in computer science and technology from the Renmin University of China in 2006, and the PhD degree in systems engineering and engineering management from the Chinese University of Hong Kong in 2010. He is currently a postdoctoral research fellow at the Chinese University of Hong Kong. His research interests include keyword search in relational databases, parallel keyword search in graphs, and structural keyword search in graphs.



Jeffrey Xu Yu received the BE, ME, and PhD degrees in computer science, from the University of Tsukuba, Japan, in 1985, 1987, and 1990, respectively. Currently, he is a professor in the Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong. His major research interests include graph mining, graph database, keyword search, and query processing and optimization. He is a senior member of the IEEE, a member of the IEEE Computer Society, and a member of ACM.

Lijun Chang received the BEng degree in computer science and technology from Renmin University of China in 2007, and the PhD degree in systems engineering and engineering management from the Chinese University of Hong Kong in 2011. He is currently a postdoctoral research fellow at the Chinese University of Hong Kong. His research interests include graph exploration, uncertain data management, and keyword search.