**Noname manuscript No.**
(will be inserted by the editor)

# Aggregate Keyword Search on Large Relational Databases

**Bin Zhou · Jian Pei**

Received: November 17, 2009 / Revised: December 05, 2010 / Accepted: December 19, 2010

**Abstract** Keyword search has been recently extended to relational databases to retrieve information from text-rich attributes. However, all the existing methods focus on finding individual tuples matching a set of query keywords from one table or the join of multiple tables. In this paper, we motivate a novel problem of aggregate keyword search: finding minimal group-bys covering a set of query keywords well, which is useful in many applications. We develop two interesting approaches to tackle the problem. We further extend our methods to allow partial matches and matches using a keyword ontology. An extensive empirical evaluation using both real data sets and synthetic data sets is reported to verify the effectiveness of aggregate keyword search and the efficiency of our methods.

**Keywords** aggregate keyword search · data cube · group-by · relational database

A preliminary version of this paper appears as (Zhou and Pei 2009). The authors are grateful to the anonymous reviewers and the associate editor for their constructive comments that help to improve the quality of the paper. This research is supported in part by an NSERC Discovery Grant, an NSERC Discovery Accelerator Supplement Grant, and a British Columbia Natural Resources and Applied Sciences Endowment Fund. All opinions, findings, conclusions and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

Bin Zhou
School of Computing Science, Simon Fraser University, 8888 University Drive, Burnaby, B.C., Canada, V5A 1S6
E-mail: bzhou@cs.sfu.ca

Jian Pei
School of Computing Science, Simon Fraser University, 8888 University Drive, Burnaby, B.C., Canada, V5A 1S6
E-mail: jpei@cs.sfu.ca

| Month | State | City | Event | Description |
|---|---|---|---|---|
| December | Texas | Houston | Space Shuttle Experience | rocket, supersonic, jet |
| December | Texas | Dallas | Cowboy's Dream Run | motorcycle, culture, beer |
| December | Texas | Austin | SPAM Museum Party | classical American Hormel foods |
| November | Arizona | Phoenix | Cowboy Culture Show | rock music |

Table 1: A table of tourism events.

## 1 Introduction

Keyword search has been well accepted as one of the most popular ways to retrieve useful information from unstructured or semi-structured data, such as Web pages (Henzinger et al 2003), images (Gong and Liu 2009), XML documents (Liu and Chen 2007), and so on. Recently, keyword search has been applied successfully on relational databases where some text attributes are used to store text-rich information. As reviewed in Section 3, all of the existing methods address the following search problem: given a set of keywords, find a set of tuples that are most relevant (for example, find the top-$k$ most relevant tuples) to the set of keywords. Here, each tuple in the answer set may be from one table or from the join of multiple tables.

While searching individual tuples using keywords is useful, in some application scenarios, a user may be interested in an *aggregate group of tuples* jointly matching a set of query keywords.

*Example 1 (Motivation)* Table 1 shows a database of tourism event calendar. Such an event calendar is popular in many tourism web sites and travel agents' databases (or data warehouses). To keep our discussion simple, in the field of `description`, a set of keywords are extracted. In general, this field can store text description of events.

Scott, a customer planning his vacation, is interested in seeing space shuttles, riding motorcycle and experiencing American food. He can search the event calendar using the set of keywords {"space shuttle", "motorcycle", "American food"}. Unfortunately, the three keywords do not appear together in any single tuple, and thus the results returned by the existing keyword search methods may contain at most one keyword in a tuple.

However, Scott may find the aggregate group (December, Texas, $*$, $*$, $*$) interesting and useful, since he can have space shuttles, motorcycle, and American food all together if he visits Texas in December. The $*$ signs on attributes `city`, `event`, and `description` mean that he will have multiple events in multiple cities with different description.

To make his vacation planning effective, Scott may want to have the aggregate as specific as possible – it should cover a small area (for example, Texas instead of the whole United States) and a short period (for example, December instead of year 2009).

In summary, the task of keyword search for Scott is to find minimal aggregates in the event calendar database such that for each of such aggregates, all keywords are contained by the union of the tuples in the aggregate.

Different from the existing studies about keyword search on relational databases which find a tuple (or a set of tuples interconnected by primary-foreign key relationships) matching the requested keywords well, the aggregate keyword search investigated in this paper tries to identify a minimal context where the keywords in a query are covered. Other than the example of the tourism event query in Example 1, there are quite a few interesting applications of the aggregate keyword queries in practice.

- For movie retailer stores, their databases store all the information about thousands of movies, such as `title` of the movie, `director`, `leading actor`, `leading actress`, `writer`, and so on. The aggregate keyword queries can suggest to the customers some interesting factors of movies that customers may find interesting. For example, a customer favoring in science fiction movies may be suggested that the director `Steven Spielberg` has directed quite a few sci-fi movies. Later on, the customer may want to watch some more movies directed by Steven Spielberg.
- For online social networking sites, their databases store all the information about different user communities and groups, such as `member's location`, `member's interest`, `member's gender`, `member's age`, and so on. The aggregate keyword queries can recommend to those new users some important factors of those social communities that they may prefer to joining in. For example, a female university student may find that most people in her age likes `shopping`. Thus, she may want to first join those communities which focus on shopping information.

Similar situations arise in some related recommendation services, such as restaurant recommendation, hotel recommendation, friend finder suggestion, shopping goods promotion, and so on.

As analyzed in Section 3, aggregate keyword search cannot be achieved efficiently using the keyword search methods developed in the existing studies, since those methods do not consider aggregate group-bys in the search which are critical for aggregate keyword search. In this paper, we tackle the problem of aggregate keyword search systematically, and make the following contributions.

First, we identify and formulate the problem of aggregate keyword search, and demonstrate its applications. To the best of our knowledge, this is the first study on aggregate keyword search. Generally, it can be viewed as the integration of online analytical processing (OLAP) and keyword search, since conceptually we conduct keyword search in a data cube.

Second, to develop efficient methods for aggregate keyword search, we develop two promising approaches. The maximal-join approach uses the inverted lists of keywords to assemble the minimal group-bys covering all keywords in the query. Several effective heuristics are identified to speed up the search.

The keyword graph approach materializes the minimal aggregates for every pair of keywords in a keyword graph index. Then, the aggregate search using multiple keywords can be reduced to generalizing the aggregates in a clique of keywords.

Third, we extend the complete aggregate keyword search to general aggregate keyword search, where partial matches (for example, matching $m'$ of $m$ keywords in a query) are allowed, and a keyword is allowed to be matched by some other similar keywords according to a keyword ontology (that is, keyword "fruit" in a query can be matched by keyword "apple" in the data). To the best of our knowledge, we are the first to consider the relaxation of keyword matching using a keyword ontology on relational databases.

Last, we empirically evaluate our techniques using both real data sets and synthetic data sets. We conduct a comparison with a baseline algorithm which is an extension of iceberg cube computation algorithm. Our experimental results show that aggregate keyword search is practical and effective on large relational databases, and our techniques can achieve high efficiency.

The rest of the paper is organized as follows. In Section 2, we formulate the problem of aggregate keyword search on relational databases. We review the related work in Section 3. We develop the maximum join approach in Section 4, and the keyword graph approach in Section 5. In Section 6, the complete aggregate keyword search is extended and generalized for partial matching and matching based on keyword ontology. A systematic performance study is reported in Section 7. Section 8 concludes the paper.

## 2 Aggregate Keyword Queries

For the sake of simplicity, in this paper, we follow the conventional terminology in online analytic processing (OLAP) and data cubes (Gray et al 1996).

**Definition 1 (Aggregate cell)** Let $T = (A_1, \ldots, A_n)$ be a relational table. An **aggregate cell** (or a **cell** for short) on table $T$ is a tuple $c = (x_1, \ldots, x_n)$ where $x_i \in A_i$ or $x_i = *$ $(1 \leq i \leq n)$, and $*$ is a meta symbol meaning that the attribute is generalized. The **cover** of aggregate cell $c$ is the set of tuples in $T$ that have the same values as $c$ on those non-$*$ attributes, that is,

$$Cov(c) = \{(v_1, \ldots, v_n) \in T | v_i = x_i \text{ if } x_i \neq *, 1 \ \leq i \leq n\}$$

A **base cell** is an aggregate cell which takes a non-$*$ value on every attribute.

For two aggregate cells $c = (x_1, \ldots, x_n)$ and $c' = (y_1, \ldots, y_n)$, $c$ is an **ancestor** of $c'$, and $c'$ a **descendant** of $c$, denoted by $c \succ c'$, if $x_i = y_i$ for each $x_i \neq *$ $(1 \leq i \leq n)$, and there exists $i_0$ $(1 \leq i_0 \leq n)$ such that $x_{i_0} = *$ but $y_{i_0} \neq *$. We write $c \succeq c'$ if $c \succ c'$ or $c = c'$.

For example, in Table 1, the cover of aggregate cell (December, Texas, $*$, $*$, $*$) contains the three tuples about the events in Texas in December. Moreover, ($*$, Texas, $*$, $*$, $*$) $\succ$ (December, Texas, $*$, $*$, $*$).

Apparently, aggregate cells have the following property.

**Corollary 1 (Monotonicity)** *For aggregate cells $c$ and $c'$ such that $c \succ c'$, $Cov(c) \supseteq Cov(c')$.*

For example, in Table 1, $Cov(*, \text{Texas}, *, *, *) \supseteq Cov(\text{December}, \text{Texas}, *, *, *)$.

In this paper, we consider keyword search on a table which contains some text-rich attributes such as attributes of character strings or large object blocks of text. Formally, we define an aggregate keyword query as follows.

**Definition 2 (Aggregate keyword query)** Given a table $T$, an **aggregate keyword query** is a 3-tuple $Q = (\mathcal{D}, \mathcal{C}, W)$, where $\mathcal{D}$ is a subset of attributes in table $T$, $\mathcal{C}$ is a subset of text-rich attributes in $T$, and $W$ is a set of keywords. We call $\mathcal{D}$ the **aggregate space** and each attribute $A \in \mathcal{D}$ a **dimension**. We call $\mathcal{C}$ the set of **text attributes** of $Q$. $\mathcal{D}$ and $\mathcal{C}$ do not have to be exclusive to each other.

An aggregate cell $c$ on $T$ is an **answer** to the aggregate keyword query (or $c$ **matches** Q for short) if (1) $c$ takes value $*$ on all attributes not in $\mathcal{D}$, that is, $c[A] = *$ if $A \notin \mathcal{D}$; and (2) for every keyword $w \in W$, there exists a tuple $t \in Cov(c)$ and an attribute $A \in \mathcal{C}$ such that $w$ appears in the text of $t[A]$.

*Example 2 (Aggregate keyword query)* The aggregate keyword query in Example 1 can be written as $Q$ = ({`Month`, `State`, `City`, `Event`}, {`Event`, `Description`}, {"space shuttle", "motorcycle", "American food"}) according to Definition 2, where table $T$ is shown in Table 1.

Due to the monotonicity of aggregate cells in covers (Corollary 1), if $c$ is an answer to an aggregate keyword query, then every aggregate cell which is an ancestor of $c$ (that is, more general than $c$) is also an answer to the query. In order to eliminate the redundancy and also address the requirements from practice that specific search results are often preferred, we propose the notion of minimal answers.

**Definition 3 (Minimal answer)** An aggregate cell $c$ is a **minimal answer** to an aggregate keyword query $Q$ if $c$ is an answer to $Q$ and every descendant of $c$ is not an answer to $Q$.

**The problem of aggregate keyword search** is to find the complete set of minimal answers to a given aggregate keyword query $Q$.

In some situations, the minimal answers can be ranked according to a specific ranking function (e.g., TF-IDF based ranking function). Users may be only interested in top-$k$ answers. However, defining a meaningful ranking function in practice is a challenge. Moreover, in different relational databases, the ranking functions can be quite different, since the data stored in the databases may be quite different. In this paper, we focus on finding the complete set of minimal answers. The problem of top-$k$ answer retrieval is interesting for future research.

It is well known that all the aggregate cells on a table form a lattice. Thus, aggregate keyword search is to search the minimal answers in the aggregate cell lattice as illustrated in the following example.

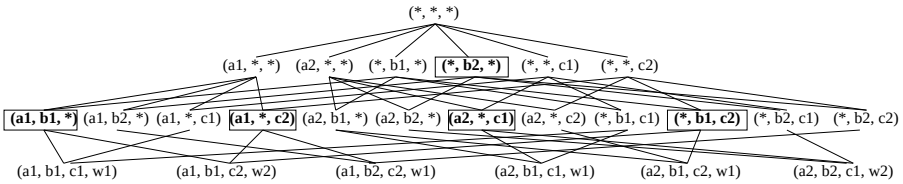



Fig. 1: The aggregate lattice on $ABC$.

| $A$ | $B$ | $C$ | $D$ |
|---|---|---|---|
| $a_1$ | $b_1$ | $c_1$ | $w_1, w_3$ |
| $a_1$ | $b_1$ | $c_2$ | $w_2, w_5$ |
| $a_1$ | $b_2$ | $c_2$ | $w_1$ |
| $a_2$ | $b_1$ | $c_1$ | $w_1, w_4$ |
| $a_2$ | $b_1$ | $c_2$ | $w_1, w_3$ |
| $a_2$ | $b_2$ | $c_1$ | $w_2, w_6$ |

Table 2: Table $T$ in Example 3.

*Example 3 (Lattice)* In table $T = (A, B, C, D)$ in Table 2, attribute $D$ contains a set of keywords $w_i$ $(i > 0)$. Consider query $Q = (ABC, D, \{w_1, w_2\})$.

Figure 1 shows the aggregate cells in aggregate space $ABC$ and the lattice. Aggregate cells $(a_1, b_1, *)$, $(a_1, *, c_2)$, $(*, b_1, c_2)$, $(*, b_2, *)$, and $(a_2, *, c_1)$ are the minimal answers to the query. They are highlighted in boxes in the figure.

## 3 Related Work

The aggregate keyword search problem is highly related to the previous studies on keyword search on relational databases and iceberg cube computation. In this section, we briefly review some representative studies and point out the differences between those studies and ours.

### 3.1 Keyword Search on Relational Databases

Recently, integration of information retrieval and database technology has attracted a lot of attention. A few critical challenges have been identified such as how to model the query answers in a semantic way and how to address the flexibility in scoring and ranking models. Chaudhuri et al (2005); Amer-Yahia et al (2005); Weikum (2007); Chen et al (2009); Chaudhuri and Das (2009); Park and goo Lee (2010) provided excellent insights into those issues.

As a concrete step to provide an integrated platform for text- and data-rich applications, keyword search on relational databases becomes an active topic in database research. Several interesting and effective solutions and prototype

systems have been developed (Agrawal et al 2002; Hristidis and Papakonstantinou 2002; Bhalotia et al 2002; Taha and Elmasri 2010).

DBXplorer (Agrawal et al 2002) is a keyword-based search system implemented using a commercial relational database and web server. DBXplorer returns all rows, either from individual tables or by joining multiple tables using foreign-keys, such that each row contains all keywords in a query. It uses a symbol table as the key data structure to look up the respective locations of query keywords in the database. DISCOVER (Hristidis and Papakonstantinou 2002) produces without redundancy all joining networks of tuples on primary and foreign keys, where a joining network represents a tuple that can be generated by joining some tuples in multiple tables. Each joining network collectively contains all keywords in a query. Both DBXplorer and DISCOVER exploit the schema of the underlying databases. Hristidis et al (2003) developed efficient methods which can handle queries with both AND and OR semantics and exploited ranking techniques to retrieve top-$k$ answers.

BANKS (Bhalotia et al 2002) models a database as a graph where tuples are nodes and application-oriented relationships are edges. Under such an extension, keyword search can be generalized on trees and graph data. BANKS searches for Steiner trees (Dreyfus and Wagner 1972) that contain all keywords in the query. Some effective heuristics are exploited to approximate the Steiner tree problem, and thus the algorithm can be applied to huge graphs of tuples. Furthermore, Kacholia et al (2005) introduced the bidirectional expansion techniques to improve the search efficiency on large graph databases. Because keyword search on graphs takes both vertex labels and graph structures into account, there are many possible strategies for ranking answers. Different ranking strategies reflect designers' respective concerns. Various effective IR-style ranking criteria and search methods are developed, such as (Liu et al 2006; Luo et al 2007; Ding et al 2007).

Most of the previous studies concentrate on finding minimal connected tuple trees from relational databases (Agrawal et al 2002; Hristidis and Papakonstantinou 2002; Hristidis et al 2003; Bhalotia et al 2002; Kacholia et al 2005; Liu et al 2006; Kimelfeld and Sagiv 2006; Luo et al 2007; Ding et al 2007). Recently, several other semantics of query answers have been proposed. Under the graph modeling of a relational database, BLINKS (He et al 2007) proposes to find distinct roots as answers to a keyword query. An $m$-keyword query finds a collection of tuples, that contain all the keywords, reachable from a root tuple within a user-given distance. BLINKS (He et al 2007) builds a bi-level index for fast keyword search on graphs. Recently, Qin et al (2009b) modeled a query answer as a community. A community contains several core vertices connecting all the keywords in the query. Qin et al (2009a) considered all the three previous semantics of query answers, and showed that the current commercial RDBMSs are powerful enough to support keyword queries in relational databases efficiently without any additional new indexing to be built and maintained. Li et al (2008) studied keyword search on large collections of heterogenous data. An $r$-radius Steiner graph semantic is proposed to model the query answers. Instead of returning sub-graphs that contain all the key-

words, ObjectRank (Balmin et al 2004) returns individual tuples as answers. It applies a modified PageRank algorithm to keyword search in a database for which there is a natural flow of authority between the objects. To calculate the global importance of an object, a random surfer has the same probability to start from any object of the base set (Tong et al 2008). ObjectRank returns objects having high authority with respect to all keywords.

The quality of approximation in top-$k$ keyword proximity search is studied in (Kimelfeld and Sagiv 2006). Yu et al (2007) used a keyword relationship matrix to evaluate keyword relationships in distributed databases. Most recently, Vu et al (2008) extended (Yu et al 2007) by summarizing each database using a keyword relationship graph, which can help to select top-$k$ most promising databases effectively in query processing.

The existing studies about keyword search on relational databases and our paper address different types of keyword queries on relational databases. First, previous studies focus on relational databases which consist of a set of relational tables. In our paper, an aggregate keyword query is conducted on a single large relational table. Second, previous studies look for *individual tuples* (or a set of tuples interconnected by primary-foreign key relationships) such that all the query keywords appear in at least one returned tuple in the database. In other words, the answers to be returned are tuples from the original database. However, in our paper, the answers to the aggregate keyword queries are aggregate cells (group-bys). All the query keywords must appear in at least one tuple in the corresponding group-bys. In other words, the answers to be returned are aggregated results from the original relational table. Third, in the previous studies, the tuples to be returned come from different relational tables which are connected by the primary-foreign key relationships. However, in our paper, the tuples are in the same table and those tuples in the same group-by have same values on those group-by attributes. As a result, the problem of aggregate keyword query is quite different from those existing studies about keyword search on relational databases.

The existing methods about keyword search on relational databases cannot be extended straightforwardly to tackle the aggregate keyword search problem. First, previous studies find tuples from different tables which are interconnected by primary-foreign key relationships, it is possible to extend some of the existing methods to compute those joining networks where tuples from the same table are joined (that is, self-join of a table). However, for a query of $m$ keywords, we need to conduct the self-joins of the original table for $m-1$ times, which is very time-consuming. Second, the number of joining networks generated by the self-joins can be much larger than the number of minimal answers due to the monotonicity of aggregate cells. Such extensions cannot compute the minimal answers to aggregate keyword queries efficiently. As a result, we need to develop specific query answering techniques to conduct the aggregate keyword queries efficiently and effectively.

### 3.2 Keyword-Driven Analytical Processing

Keyword-driven analytical processing (KDAP) (Wu et al 2007) probably is the work most relevant to our study. KDAP involves two phases. In the differentiate phase, for a set of keywords, a set of candidate subspaces are generated where each subspace corresponds to a possible join path between the dimensions and the facts in a data warehouse schema (for example, a star schema). In the explore phase, for each subspace, the aggregated values for some predefined measure are calculated and the top-$k$ interesting group-by attributes to partition the subspace are found.

For instance, as an example in (Wu et al 2007), to answer a query "Columbus LCD", the KDAP system may aggregate the sales about "LCD" and break down the results into sub-aggregates for "Projector Technology = LCD", "Department = Monitor, Product = Flat Panel (LCD)", etc. Only the tuples that link with "Columbus" will be considered. A user can then drill down to aggregates of finer granularity.

Both the KDAP method and our study consider aggregate cells in keyword matching. The critical difference is that the two approaches address two different application scenarios. In the KDAP method, the aggregates of the most general subspaces are enumerated, and the top-$k$ interesting group-by attributes are computed to help a user to drill down the results. In other words, KDAP serves the interactive exploration of data using keyword search.

In this study, the aggregate keyword search is modeled as a type of queries. Only the minimal aggregate cells matching a query are returned. Moreover, we focus on the efficiency of query answering. Please note that Wu et al (2007) did not report any experimental results on the efficiency of query answering in KDAP since it is not a major concern in that study.

### 3.3 Iceberg Cube Computation

As elaborated in Example 3, aggregate keyword search finds aggregate cells in a data cube lattice (that is, the aggregate cell lattice) in the aggregate space $\mathcal{D}$ in the query. Thus, aggregate keyword search is related to the problem of iceberg cube computation which has been studied extensively.

The concept of data cube is formulated in (Gray et al 1996). Fang et al (1998) proposed iceberg queries which find in a cube lattice the aggregate cells satisfying some given constraints (for example, aggregates whose `SUM` passing a given threshold).

Efficient algorithms for computing iceberg cubes with respect to various constraints have been developed. Particularly, the BUC algorithm (Beyer and Ramakrishnan 1999) exploits monotonic constraints like $\texttt{COUNT}(*) \geq v$ and conducts a bottom-up search (that is, from the most general aggregate cells to the most specific ones). Han et al (2001) tackled non-monotonic constraints like $\texttt{AVG}(*) \geq v$ by using some weaker but monotonic constraints in pruning. More efficient algorithms for iceberg cube computation are proposed in (Xin

et al 2003; Feng et al 2004). The problem of iceberg cube computation on distributed network environment is investigated in (Ng et al 2001).

A keyword query can be viewed as a special case of iceberg queries, where the constraint is that the tuples in an aggregate cell should jointly match all keywords in the query. However, this kind of constraints have not been explored in the literature of iceberg cube computation. The existing methods only consider the constraints composed by SQL aggregates like `SUM`, `AVG` and `COUNT`. In those constraints, every tuple in an aggregate cell contributes to the aggregate which will be computed and checked against the constraint. In aggregate keyword search, a keyword is expected to appear in only a small subset of tuples. Therefore, most tuples of an aggregate cell may not match any keyword in the query, and thus do not need to be considered in the search.

Due to the monotonicity in aggregate keyword search (Corollary 1), can we straightforwardly extend an existing iceberg cube computation method like BUC to tackle the aggregate keyword search problem? In aggregate keyword search, we are interested in the minimal aggregate cells matching all keywords in the query. However, all the existing iceberg cube computation methods more or less follow the BUC framework and search from the most general cell to the most specific cells in order to use monotonic constraints to prune the search space. The general-to-specific search strategy is inefficient for aggregate keyword search since it has to check many answers to the query until the minimal answers are computed.

## 4 The Maximum Join Approach

Inverted indexes of keywords (Harman et al 1992) are heavily used in keyword search and have been supported extensively in practical systems. It is natural to exploit inverted indexes of keywords to support aggregate keyword search.

### 4.1 A Simple Nested Loop Solution

For a keyword $w$ and a text-rich attribute $A$, let $IL_A(w)$ be the inverted list of tuples which contain $w$ in attribute $A$. That is, $IL_A(w) = \{t \in T | w \text{ appears in } t[A]\}$.

Consider a simple query $Q = (\mathcal{D}, C, \{w_1, w_2\})$ where there are only two keywords in the query and there is only one text-rich attribute $C$. How can we derive the minimal answers to the query from $IL_C(w_1)$ and $IL_C(w_2)$?

For a tuple $t_1 \in IL_C(w_1)$ and a tuple $t_2 \in IL_C(w_2)$, every aggregate cell $c$ that is a common ancestor of both $t_1$ and $t_2$ matches the query. We are interested in the minimal answers. Then, what is the most specific aggregate cell that is a common ancestor of both $t_1$ and $t_2$?

**Definition 4 (Maximum join)** For two tuples $t_x$ and $t_y$ in table $R$, the **maximum join** of $t_x$ and $t_y$ on attribute set $\mathcal{A} \subseteq R$ is a tuple $t = t_x \vee_{\mathcal{A}} t_y$

**Algorithm 1** The simple nested loop algorithm.

**Input:** query $Q = (\mathcal{D}, C, \{w_1, w_2\})$, $IL_C(w_1)$ and $IL_C(w_2)$;
**Output:** minimal aggregates matching $Q$;

```
    Step 1: generate possible minimal aggregates
 1: Ans = ∅; // Ans is the answer set
 2: for each tuple t1 ∈ IL_C(w1) do
 3:    for each tuple t2 ∈ IL_C(w2) do
 4:       Ans = Ans ∪ {t1 ∨_D t2};
 5:    end for
 6: end for
    Step 2: remove non-minimal aggregates from Ans
 7: for each tuple t ∈ Ans do
 8:    for each tuple t' ∈ Ans do
 9:       if t' ≺ t then
10:          Ans = Ans − {t'};
11:       else if t ≺ t' then
12:          Ans = Ans − {t};
13:          break;
14:       end if
15:    end for
16: end for
```

such that (1) for any attribute $A \in \mathcal{A}$, $t[A] = t_x[A]$ if $t_x[A] = t_y[A]$, otherwise $t[A] = *$; and (2) for any attribute $B \notin \mathcal{A}$, $t[B] = *$.

We call this operation maximum join since it keeps the common values between the two operant tuples on as many attributes as possible.

*Example 4 (Maximum join)* In Table 2, $(a_1, b_1, c_1, \{w_1, w_3\}) \vee_{ABC} (a_1, b_1, c_2, \{w_2, w_5\}) = (a_1, b_1, *, *)$.

As can be seen from Figure 1, the maximal-join of two tuples gives the least upper bound (supremum) of the two tuples in the lattice. In general, we have the following property of maximum joins.

**Corollary 2 (Properties)** *The maximum-join operation is associative. That is,* $(t_1 \vee_{\mathcal{A}} t_2) \vee_{\mathcal{A}} t_3 = t_1 \vee_{\mathcal{A}} (t_2 \vee_{\mathcal{A}} t_3)$. *Moreover,* $\vee_{i=1}^{l} t_i$ *is the supremum of tuples* $t_1, \ldots, t_l$ *in the aggregate lattice.*

Using the maximum join operation, we can conduct a nested loop to answer a simple query $Q = (\mathcal{D}, C, \{w_1, w_2\})$ as shown in Algorithm 1. The algorithm is in two steps. In the first step, maximum joins are applied on pairs of tuples from $IL_C(w_1)$ and $IL_C(w_2)$. The maximum joins are candidates for minimal answers. In the second step, we remove those aggregates that are not minimal.

The simple nested loop method can be easily extended to handle queries with more than two keywords and more than one text-rich attribute. Generally, for query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, \ldots, w_m\})$, we can derive the inverted list of keyword $w_i$ $(1 \leq i \leq m)$ on attribute set $\mathcal{C}$ as $IL_{\mathcal{C}}(w_i) = \cup_{C \in \mathcal{C}} IL_C(w_i)$. Moreover, the first step of Algorithm 1 can be extended so that $m$ nested loops are conducted to obtain the maximal joins of tuples $\vee_{i=1}^{m} t_i$ where $t_i \in IL_{\mathcal{C}}(w_i)$.

To answer query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, \ldots, w_m\})$, the nested loop algorithm has the time complexity $O(\prod_{i=1}^{m} |IL_{\mathcal{C}}(w_i)|^2)$. The first step takes time $O(\prod_{i=1}^{m} |IL_{\mathcal{C}}(w_i)|)$ and may generate up to $\prod_{i=1}^{m} |IL_{\mathcal{C}}(w_i)|$ aggregates in the answer set. To remove the non-minimal answers, the second step takes time $O(\prod_{i=1}^{m} |IL_{\mathcal{C}}(w_i)|^2)$. In a relational table $T = (A_1, \ldots, A_n)$ with $N$ rows, the total number of aggregate cells is $O(N \cdot 2^n)$ or $O(\Pi_{i=1}^{n}(|A_i| + 1))$, where $|A_i|$ is the number of distinct values in dimension $A_i$. Thus, the simple nested algorithm needs $O(N^2 \cdot 2^{n+1})$ time in the worst case.

Clearly, the nested loop method is inefficacious for large databases and queries with multiple keywords. In the rest of this section, we will develop several interesting techniques to speed up the search.

### 4.2 Pruning Exactly Matching Tuples

Hereafter, when the set of text-rich attributes $\mathcal{C}$ is clear from context, we write an inverted list $IL_{\mathcal{C}}(w)$ as $IL(w)$ for the sake of simplicity. Similarly, we write $t_1 \vee_{\mathcal{D}} t_2$ as $t_1 \vee t_2$ when $\mathcal{D}$ is clear from the context.

**Theorem 1 (Pruning exactly matching tuples)** *Consider query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, w_2\})$ and inverted lists $IL(w_1)$ and $IL(w_2)$. For tuples $t_1 \in IL(w_1)$ and $t_2 \in IL(w_2)$ such that $t_1[\mathcal{D}] = t_2[\mathcal{D}]$, $t_1 \vee t_2$ is a minimal answer. Moreover, except for $t_1 \vee t_2$, no other minimal answers can be an ancestor of either $t_1$ or $t_2$.*

**Proof:** The minimality of $t_1 \vee t_2$ holds since $t_1 \vee t_2$ does not take value $*$ on any attributes in $\mathcal{D}$. Except for $t_1 \vee t_2$, every ancestor of $t_1$ or $t_2$ must be an ancestor of $t_1 \vee t_2$ in $\mathcal{D}$, and thus cannot be a minimal answer. □

Using Theorem 1, once two tuples $t_1 \in IL(w_1)$ and $t_2 \in IL(w_2)$ are found such that $t_1[\mathcal{D}] = t_2[\mathcal{D}]$, $t_1 \vee t_2$ should be output as a minimal answer, and $t_1$ and $t_2$ should be ignored in the rest of the join.

### 4.3 Reducing Matching Candidates Using Answers

For an aggregate keyword query, we can use some answers found before, which may not even be minimal, to prune matching candidates.

**Theorem 2 (Reducing matching candidates)** *Let $t$ be an answer to $Q = (\mathcal{D}, \mathcal{C}, \{w_1, w_2\})$ and $t_1 \in IL(w_1)$. For any tuple $t_2 \in IL(w_2)$, if for every attribute $D \in \mathcal{D}$ where $t_1[D] \neq t[D]$, $t_2[D] \neq t_1[D]$, then $t_1 \vee t_2$ is not a minimal answer to the query.*

**Proof:** For every attribute $D \in \mathcal{D}$ where $t_1[D] \neq t[D]$, since $t_1[D] \neq t_2[D]$, $(t_1 \vee t_2)[D] = *$. On every other attribute $D' \in \mathcal{D}$ where $t_1[D'] = t[D']$, either $(t_1 \vee t_2)[D'] = t_1[D'] = t[D']$ or $(t_1 \vee t_2)[D'] = *$. Thus, $t_1 \vee t_2 \preceq t$. Consequently, $t_1 \vee t_2$ cannot be a minimal answer to $Q$. □

Using Theorem 2, for each tuple $t_1 \in IL(w_1)$, if there is an answer $t$ (not necessary a minimal one) to query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, w_2\})$ such that $t[D] = t_1[D]$ on some attribute $D \in \mathcal{D}$, we can use $t$ to reduce the tuples in $IL(w_2)$ that need to be joined with $t_1$ as elaborated in the following example.

*Example 5 (Reducing matching candidates)* Consider query $Q = (ABC, D, \{w_1, w_2\})$ on the table $T$ shown in Table 2. A maximum join between $(a_1, b_1, c_2)$ and $(a_1, b_2, c_2)$ generates an answer $(a_1, *, c_2)$ to the query. Although tuple $(a_1, b_1, c_2)$ contains $w_1$ on $D$ and tuple $(a_2, b_2, c_1)$ contains $w_2$ on $D$, as indicated by Theorem 2, joining $(a_1, b_1, c_2)$ and $(a_2, b_2, c_1)$ results in aggregate cell $(*, *, *)$, which is an ancestor of $(a_1, *, c_2)$ and cannot be a minimal answer.

For a tuple $t_1 \in IL(w_1)$ and an answer $t$ to a query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, w_2\})$, the tuples in $IL(w_2)$ surviving from the pruning using Theorem 2 can be found efficiently using inverted lists. In implementation, instead of maintaining an inverted list $IL(w)$ of keyword $w$, we maintain a set of inverted lists $IL_{A=a}(w)$ for every value $a$ in the domain of every attribute $A$, which links all tuples having value $a$ on attribute $A$ and containing keyword $w$ on the text-rich attributes. Clearly, $IL(w) = \cup_{a \in A} IL_{A=a}(w)$ where $A$ is an arbitrary attribute. Here, we assume that tuples do not take null value on any attribute. If some tuples take null values on some attributes, we can simply ignore those tuples on those attributes, since those tuples do not match any query keywords on those attributes.

Suppose $t$ is an answer to query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, w_2\})$ and $t_1$ contains keyword $w_1$ but not $w_2$. Then, $t_1$ needs to join with only the tuples in

$$Candiate(t_1) = \cup_{D \in \mathcal{D}: t_1[D] \neq t[D]} IL_{D=t_1[D]}(w_2).$$

Other tuples in $IL(w_2)$ should not be joined with $t_1$ according to Theorem 2.

An answer $t$ is called *overlapping* with a tuple $t_1$ if there exists at least one attribute $D \in \mathcal{D}$ such that $t[D] = t_1[D]$. Clearly, the more answers overlapping with $t_1$, the more candidates can be reduced.

Heuristically, the more specific $t_1 \vee t$ in Theorem 2, the more candidate tuples may be reduced for the maximum joins with $t_1$. Therefore, for each tuple $t_1 \in IL(w_1)$, we should try to find $t_2 \in IL(w_2)$ such that $t = t_1 \vee t_2$ contains as few $*$'s as possible. To implement the heuristic, for query $Q = (\mathcal{D}, \mathcal{C}, (w_1, w_2))$ and each tuple $t_1 \in IL(w_1)$ currently in the outer loop, we need to measure how well a tuple in $IL_{\mathcal{C}}(w_2)$ matches $t_1$ in $\mathcal{D}$. This can be achieved efficiently using bitmap operations as follows.

We initialize a counter of value 0 for every tuple in inverted list $IL_{\mathcal{C}}(w_2)$. For each attribute $D \in \mathcal{D}$, we compute $IL_{D=t_1[D]}(w_1) \cap IL_{\mathcal{D}}(w_2)$ using bitmaps. For each tuple in the intersection, the counter of the tuple is incremented by 1. After checking all attributes in $\mathcal{D}$, the tuples having the largest counter value match $t_1$ the best. Thus, we can sort tuples in $IL_{\mathcal{C}}(w_2)$ in the counter value descending order and conduct maximum joins with $t_1$ on them. In this order, the most specific matches can be found the earliest.

Comparing to the traditional solution which builds an inverted list $IL(w)$ for each keyword $w$, in our method, the number of inverted lists is larger. However, in terms of the space complexity, maintaining one inverted list per keyword and multiple lists per keyword only differs in $O(|T| \cdot |D|)$ space, where $|T|$ is the number of keywords, and $|D|$ is the number of dimensional attributes. Since $|D|$ is a constant, the space complexity is the same. Moreover, the inverted lists can be accessed in $O(1)$ time using hashing. Furthermore, as explained above, the number of inverted lists to be accessed only depends on the number of dimensions on which a tuple $t_1$ and an answer $t$ have different values. The heuristic used in our method can find the most specific answers as early as possible, which in turn is beneficial for the pruning strategy, since the number of dimensions that $t_1$ and $t$ having different values is small. As a result, maintaining the inverted lists for each dimension values is useful to improve the query performance.

### 4.4 Fast Minimal Answer Checking

In order to obtain minimal answers to an aggregate keyword query, we need to remove non-minimal answers from the answer set. The naïve method in Algorithm 1 adopts a nested loop. Each time when a new answer $t$ (not necessary a minimal answer) is found, we need to scan the answer set once to consider if $t$ should be inserted into the answer set. If $t$ is an ancestor of some answers in the answer set, $t$ cannot be a minimal answer, thus $t$ should not be inserted into the answer set. In the other case, if $t$ is a descendant of some answers in the answer set, those answers cannot be the minimal answers, thus need to be removed from the answer set. Without any index structures, it leads to $O(n^2)$ time complexity, where $n$ is the size of the answer set. Can we do better?

The answers in the answer set can be organized into inverted lists. For an arbitrary attribute $A$, $IL_{A=a}$ represents the inverted list for all answers $t$ such that $t[A] = a$. Using Definition 1, we have the following result.

**Corollary 3 (Ancestor and descendant)** *Suppose the answers to query $Q = (\mathcal{D}, \mathcal{C}, W)$ are organized into inverted lists, and $t$ is an answer. Let*

$$\begin{aligned} Ancestor(t) = &(\cap_{D \in \mathcal{D}: t[D] \neq *} (IL_{D=*} \cup IL_{D=t[D]})) \\ &\cap (\cap_{D \in \mathcal{D}: t[D] = *} IL_{D=*}) \end{aligned}$$

*and $Descendant(t) = \cap_{D \in \mathcal{D}: t[D] \neq *} IL_{D=t[D]}$. That is, if an answer $t_1 \in Ancestor(t)$, for those dimensions that $t$ has a $*$ value, $t_1$ also has a $*$ value; for those dimensions that $t$ has a non-$*$ value, $t_1$ must either have the same non-$*$ value or a $*$ value; if an answer $t_2 \in Descendant(t)$, for those dimensions that $t$ has a non-$*$ value, $t_2$ must have the same non-$*$ value.*

*Then, the answers in $Ancestor(t)$ are not minimal. Moreover, $t$ is not minimal if $Descendant(t) \neq \emptyset$.*

Both $Ancestor(t)$ and $Descendant(t)$ can be implemented efficiently using bitmaps. For each newly found answer $t$, we calculate $Ancestor(t)$ and

**Algorithm 2** The fast maximum-join algorithm.

**Input:** query $Q = (\mathcal{D}, C, \{w_1, \ldots, w_m\})$, $IL_C(w_1)$, $\ldots$, $IL_C(w_m)$;
**Output:** minimal aggregates matching $Q$;

```
1: Ans = ∅; // Ans is the answer set
2: CandList = {IL_C(w_1), ..., IL_C(w_m)};
3: initialize k = 1; /* m keywords need m − 1 rounds of joins */
4: while k < m do
5:    k = k + 1;
6:    pick two candidate inverted lists IL_C^1 and IL_C^2 with smallest sizes from CandList,
      and remove them from CandList;
7:    for each tuple t_1 ∈ IL_C^1 do
8:       use strategy in Section 4.3 to calculate the counter for tuples in IL_C^2;
9:       while IL_C^2 is not empty do
10:         let t_2 be the tuple in IL_C^2 with largest counter;
11:         if the counter of t_2 is equal to the dimension then
12:            use strategy in Section 4.4 to insert an answer t_1 into Ans; /* t_1 exactly
               matches t_2, as described in Section 4.2 */
13:            remove t_1 and t_2 from each inverted list;
14:            break;
15:         else
16:            maximum join t_1 and t_2 to obtain an answer;
17:            use strategy in Section 4.4 to insert the answer into Ans;
18:            use strategy in Section 4.3 to find candidate tuples in IL_C^2, and update IL_C^2;
19:         end if
20:      end while
21:   end for
22:   build an inverted list for answers in Ans and insert it into CandList;
23: end while
```

*Descendant*$(t)$. If *Descendant*$(t) \neq \emptyset$, $t$ is not inserted into the answer set. Otherwise, $t$ is inserted into the answer set, and all answers in *Ancestor*$(t)$ are removed from the answer set. In this way, we maintain a small answer set during the maximal join computation. When the computation is done, the answers in the answer set are guaranteed to be minimal.

### 4.5 Integrating the Speeding-Up Strategies

The strategies described in Sections 4.2, 4.3, and 4.4 can be integrated into a fast maximum-join approach as shown in Algorithm 2.

For a query containing $m$ keywords, we need $m - 1$ rounds of maximum joins. Heuristically, the larger the sizes of the two inverted lists in the maximum join, the more costly the join. Here, the size of an inverted list is the number of tuples in the list. Thus, in each round of maximum joins, we pick two inverted lists with the smallest sizes.

It is a challenging problem to develop a better heuristic to decide the orders of maximum joins for those keywords in the query. The general principle is to reduce the total number of maximum joins which are needed in the algorithm. However, there are a set of challenges which need to be addressed.

The first challenge is how to accurately estimate the number of maximum joins which are needed. In relational databases, we can accurately estimate the number of joins (e.g., equi-join, left outer join) based on the correlations of attribute values. However, in our case, the situation is much more complicated. We developed a set of pruning strategies to prune the number of maximum joins in the algorithm. Those pruning strategies may affect the estimation greatly. As a result, the estimation needs to take into account the pruning strategies, and is far from trivial.

The second challenge is how to estimate the number of intermediate minimal answers generated in each round of maximum joins. In our problem, the candidate answers generated by any two keywords will become the input to the next rounds of maximum joins. Thus, the size of the intermediate minimal answers plays an important role for the performance of the maximum-join algorithm. Consider two pairs of keywords $(w_1, w_2)$ and $(w_3, w_4)$. Suppose the number of maximum joins for $w_1$ and $w_2$ is smaller than that for $w_3$ and $w_4$, we cannot simply make a conclusion that $w_1$ and $w_2$ should be maximum-joined first, since it is possible that the number of the intermediate minimal answers generated by joining $w_1$ and $w_2$ is much larger than that by joining $w_3$ and $w_4$.

The current heuristic used in our algorithm is very simple, and it achieves relatively good performance. For a performance comparison, we compared our simple heuristic to a random selection method (that is, for a query containing $m$ keywords, we randomly pick two words and get the corresponding lists of tuples for the next round of maximum joins). In Figure 9, we show the performance of the maximum-join algorithm using the simple heuristic we developed in this paper, as well as that using the random method. The simple heuristic clearly outperforms the random method. We leave the problem of developing a better heuristic for deciding the orders of maximum-joins as an interesting future direction.

When we conduct the maximum joins between the tuples in two inverted lists, for each tuple $t_1 \in IL_C^1$, we first compute the counters of tuples in $IL_C^2$, according to the strategy in Section 4.3. Apparently, if the largest counter value is equal to the number of dimensions in the table, the two tuples are exactly matching tuples. According to the strategy in Section 4.2, the two tuples can be removed and a minimal answer is generated. We use the strategy in Section 4.4 to insert the answer into the answer set. If the largest counter value is less than the number of dimensions, we pick the tuple $t_2$ with the largest counter value and compute the maximum join of $t_1$ and $t_2$ as an answer. Again, we use the strategy in Section 4.4 to insert the answer into the answer set. The answer set should be updated accordingly, and non-minimal answers should be removed.

Based on the newly found answer, we can use the strategy in Section 4.3 to reduce the number of candidate tuples to be joined in $IL_C^2$. Once $IL_C^2$ becomes empty, we can continue to pick the next tuple in $IL_C^1$. At the end of the algorithm, the answer set contains exactly the minimal answers.

## 5 The Keyword Graph Approach

If the number of keywords in an aggregate keyword query is not small, or the database is large, the fast maximum join method may still be costly. In this section, we propose to materialize a keyword graph index for fast answering of aggregate keyword queries.

### 5.1 Keyword Graph Index and Query Answering

Since graphs are capable of modeling complicated relationships, several graph-based indices have been proposed to efficiently answer some queries. For example, Yu et al (2007) proposed a keyword relationship matrix to evaluate keyword relationships in distributed databases, which can be considered as a special case of a graph index. Moreover, Vu et al (2008) extended (Yu et al 2007) by summarizing each database using a keyword relationship graph, where nodes represent terms and edges describe relationships between them. The keyword relationship graph can help to select top-$k$ most promising databases effectively during the query processing. Recently, Daoud et al (2009) presented a personalized search approach that involves a graph-based representation of the user profile.

However, those graph indexes are not designed for aggregate keyword queries. Can we develop keyword graph indexes for effective and efficient aggregate keyword search?

Apparently, for an aggregate keyword query $Q = (\mathcal{D}, \mathcal{C}, W)$, $(*, *, \ldots, *)$ is an answer if for every keyword $w \in W$, $IL_{\mathcal{C}}(w) \neq \emptyset$. This can be checked easily. We call $(*, *, \ldots, *)$ a *trivial answer*. We build the following keyword graph index to find non-trivial answers to an aggregate keyword query.

**Definition 5 (Keyword graph index)** Given a table $T$, a **keyword graph index** is an undirected graph $G(T) = (V, E)$ such that (1) $V$ is the set of keywords in the text-rich attributes in $T$; and (2) $(u, v) \in E$ is an edge if there exists a non-trivial answer to query $Q_{u,v} = (\mathcal{T}, \mathcal{T}, \{w_1, w_2\})$, where $\mathcal{T}$ represents the complete set of attributes in $T$. Edge $(u, v)$ is associated with the set of minimal answers to query $Q_{u,v}$.

Obviously, the number of edges in the keyword graph index is $O(|V|^2)$, while each edge is associated with the complete set of minimal answers. In practice, the number of keywords in the text-rich attributes is limited, and many keyword pairs lead to trivial answers only. Thus, a keyword graph index can be maintained easily.

Keyword graph indices have the following property.

**Theorem 3 (Keyword graph)** *For an aggregate keyword query $Q = (\mathcal{D}, \mathcal{C}, W)$, there exists a non-trivial answer to $Q$ in table $T$ only if in the keyword graph index $G(T)$ on table $T$, there exists a clique on the set $W$ of vertices.*

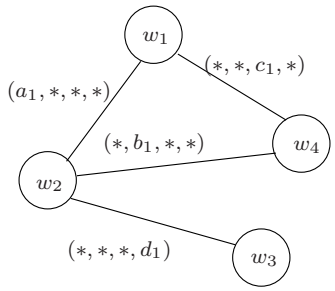



Fig. 2: A counter example showing that the condition in Theorem 3 is not sufficient.

**Proof:** Let $c$ be a non-trivial answer to $Q$. Then, for any $u, v \in W$, $c$ must be a non-trivial answer to query $Q_{u,v} = (\mathcal{D}, \mathcal{C}, \{u, v\})$. That is, $(u, v)$ is an edge in $G(T)$. □

Theorem 3 is a necessary condition. It is easy to see that the condition is not sufficient.

*Example 6 (A counter example)* Consider a keyword graph index in Figure 2. There are 4 distinct keywords. For a query $Q = \{w_1, w_2, w_4\}$, there exists a clique in the keyword graph index. However, by joining minimal answers on edges connecting those keywords, we can find that $Q$ does not have a nontrivial answer.

Once the minimal answers to aggregate keyword queries on keyword pairs are materialized in a keyword graph index, we can use Theorem 3 to answer queries efficiently. For query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, \ldots, w_m\})$, we can check whether there exists a clique on vertices $w_1, \ldots, w_m$. If not, then there is no non-trivial answer to the query. If there exists a clique, we try to construct minimal answers using the minimal answer sets associated with the edges in the clique.

If the query contains only two keywords (that is, $m = 2$), the minimal answers can be found directly from edge $(w_1, w_2)$ since the answers are materialized on the edge. If the query involves more than 2 keywords (that is, $m \geq 3$), the minimal answers can be computed by maximum joins on the sets of minimal answers associated with the edges in the clique. It is easy to show the following.

**Lemma 1 (Maximal join on answers)** *If $t$ is a minimal answer to query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, \ldots, w_m\})$, then there exist minimal answers $t_1$ and $t_2$ to queries $(\mathcal{D}, \mathcal{C}, \{w_1, w_2\})$ and $(\mathcal{D}, \mathcal{C}, \{w_2, \ldots, w_m\})$, respectively, such that $t = t_1 \vee_{\mathcal{D}} t_2$.*

Let $Answer(Q_1)$ and $Answer(Q_2)$ be the sets of minimal answers to queries $Q_1 = (\mathcal{D}, \mathcal{C}, \{w_1, w_2\})$ and $Q_2 = (\mathcal{D}, \mathcal{C}, \{w_2, w_3\})$, respectively. We call the process of applying maximal joins on $Answer(Q_1)$ and $Answer(Q_2)$ to compute

the minimal answers to query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, w_2, w_3\})$ the *join* of $Answer(Q_1)$ and $Answer(Q_2)$. The cost of the join is $O(|Answer(Q_1)| \cdot |Answer(Q_2)|)$.

To answer query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, \ldots, w_m\})$, using Lemma 1 repeatedly, we only need to check $m-1$ edges covering all keywords $w_1, \ldots, w_m$ in the clique. Each edge is associated with the set of minimal answers to a query on a pair of keywords. The *weight* of the edge is the size of the answer set. In order to reduce the total cost of the joins, heuristically, we can find a spanning tree connecting the $m$ keywords such that the product of the weights on the edges is minimized.

The traditional minimum spanning tree problem minimizes the sum of the edge weights. Several greedy algorithms, such as Prim's algorithm and Kruskal's algorithm (Cormen et al 2001), can find the optimal answers in polynomial time. The greedy selection idea can also be applied to our problem here. The greedy method works as follows: all keywords in the query are unmarked in the beginning. We sort the edges in the clique in the weight ascending order. The edge of the smallest weight is picked first and the keywords connected by the edge are marked. Iteratively, we pick a new edge of the smallest weight such that it connects a marked keyword and an unmarked one until all keywords in the query are marked.

### 5.2 Index Construction

A naïve method to construct a keyword graph is to compute maximum joins on the inverted lists of every keyword pairs. However, the naïve method is inefficient. If tuple $t_1$ contains keywords $w_1$ and $w_2$, and tuple $t_2$ contains $w_3$ and $w_4$, $t_1 \vee t_2$ may be computed up to 4 times since $t_1 \vee t_2$ is an answer to four pairs of keywords including $(w_1, w_3)$, $(w_1, w_4)$, $(w_2, w_3)$ and $(w_2, w_4)$.

As an efficient solution, we conduct a self-maximum join on the table to construct the keyword graph. For two tuples $t_1$ and $t_2$, we compute $t_1 \vee t_2$ only once, and add it to all edges of $(u, v)$ where $u$ and $v$ are contained in $t_1$ and $t_2$, but not both in either $t_1$ or $t_2$. By removing those non-minimal answers, we find all the minimal answers for every pair of keywords, and obtain the keyword graph.

Trivial answers are not stored in a keyword graph index. This constraint improves the efficiency of keyword graph construction. For a tuple $t$, the set of tuples that generate a non-trivial answer by a maximum join with $t$ is $\cup_D IL_{D=t[D]}$, where $IL_{D=t[D]}$ represents the inverted list for all tuples having value $t[D]$ on dimension $D$. Maximum joins should be applied to only those tuples and $t$. The keyword graph construction method is summarized in Algorithm 3.

### 5.3 Index Maintenance

A keyword graph index can be maintained easily against insertions, deletions and updates on the table.

**Algorithm 3** The keyword graph construction algorithm.

**Input:** A table $T$;
**Output:** A keyword graph $G(T) = (V, E)$;
1: initialize $V$ as the set of keywords in $T$;
2: **for** each tuple $t \in T$ **do**
3: initialize the candidate tuple set to $Cand = \emptyset$;
4: let $Cand = \cup_D IL_{D=t[D]}$;
5: let $W_t$ be the set of keywords contained in $t$;
6: **for** each tuple $t' \in Cand$ **do**
7: **if** $t = t'$ **then**
8: **for** each pair $w_1, w_2 \in W_t$ **do**
9: add $t$ to edge $(w_1, w_2)$, and remove non-minimal answers on edge $(w_1, w_2)$ (Section 4.4);
10: **end for**
11: **else**
12: let $W_{t'}$ be the set of keywords contained in $t'$;
13: $r = t \vee t'$;
14: **for** each pair $w_1 \in W_t - W_{t'}$ and $w_2 \in W_{t'} - W_t$ **do**
15: add $r$ to edge $(w_1, w_2)$;
16: remove non-minimal answers on edge $(w_1, w_2)$ (Section 4.4);
17: **end for**
18: **end if**
19: **end for**
20: **end for**

When a new tuple $t$ is inserted into the table, we only need to conduct the maximum join between $t$ and the tuples already in the table as well as $t$ itself. If $t$ contains some new keywords, we create the corresponding keyword vertices in the keyword graph. The maintenance procedure is the same as lines 3-19 in Algorithm 3.

When a tuple $t$ is deleted from the table, for a keyword only appearing in $t$, the vertex and the related edges in the keyword graph should be removed. If $t$ also contains some other keywords, we conduct maximum joins between $t$ and other tuples in the table. If the join result appears as a minimal answer on an edge $(u, v)$ where $u$ and $v$ are two keywords, we re-compute the minimal answers of $Q_{u,v} = (\mathcal{T}, \mathcal{T}, \{u, v\})$ by removing $t$ from $T$.

When a tuple $t$ is updated, it can be treated as one deletion (the original tuple is deleted) and one insertion (the new tuple is inserted). The keyword graph index can be updated accordingly.

## 6 Extensions and Generalization

The methods in Sections 4 and 5 look for *complete matches*, that is, all keywords are contained in an answer. However, complete matches may not exist for some queries. For example, in Table 1, query $Q = (\{$`Month, State, City, Event`$\}, \{$`Event, Descriptions`$\}, \{$"space shuttle", "motorcycle", "rock music"$\})$ cannot find a non-trivial answer.

In this section, we propose two solutions to handle queries which do not have a non-trivial answer or even an answer. Our first solution allows partial

matches (for example, matching $m'$ of $m$ keywords ($m' \leq m$)). In our second solution, a keyword is allowed to be matched by some other similar keywords according to a keyword ontology (for example, keyword "fruit" in a query can be matched by keyword "apple" in the data).

### 6.1 Partial Keyword Matching

Given a query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, \ldots, w_m\})$, **partial keyword matching** is to find all minimal, non-trivial answers that cover as many query keywords as possible.

For example, in Table 1, there is no non-trivial answer to query $Q$ = ({`Month, State, City, Event`}, {`Event, Descriptions`}, {"space shuttle", "motorcycle", "rock music"}). However, a minimal answer (December, Texas, $*$, $*$, $*$) partially matching $\frac{2}{3}$ of the keywords may still be interesting to the user.

For a query containing $m$ keywords, a brute-force solution is to consider all possible combinations of $m$ keywords, $m-1$ keywords, ..., until some non-trivial answers are found. For each combination of keywords, we need to conduct the maximum join to find all the minimal answers. Clearly, it is inefficient at all.

Here, we propose an approach using the keyword graph index. Theorem 3 provides a necessary condition that a complete match exists if the corresponding keyword vertices in the keyword graph form a clique. Given a query containing $m$ keywords $w_1, \ldots, w_m$, we can check the subgraph $G(Q)$ of the keyword graph which contains only the keyword vertices in the query. By checking $G(Q)$, we can identify if some non-trivial answers may exist for a subset of query keywords. Moreover, we can identify the maximum number of query keywords that can be matched by extracting the maximum clique from the corresponding query keyword graph.

Although the maximum clique problem is one of the first problems shown to be NP-complete (Garey and Johnson 1979), in practice, an aggregate keyword query often contains a small number of keywords (for example, less than 10). Thus, the query keyword subgraph $G(Q)$ is often small. It is possible to enumerate all the possible cliques in $G(Q)$.

To find partial matches to an aggregate keyword query, we start from those largest cliques. By joining the sets of minimal answers on the edges, the minimal answers can be found. If there is no non-trivial answer in the largest cliques, we need to consider the smaller cliques and the minimal answers. The algorithm stops until some non-trivial minimal answers are found.

Alternatively, a user may provide the minimum number of keywords that need to be covered in an answer. Our method can be easily extended to answer such a constraint-based partial keyword matching query – we only need to search answers on those cliques whose size passes the user's constraint.

In some other situations, a user can specify a subset of keywords that must be covered in an answer. Our method can also be easily extended to deal with

such cases. Those selected keywords are chosen as seed nodes in the keyword graph index. We only need to find larger cliques containing those seed nodes.

### 6.2 Hierarchical Keyword Matching

Keywords generally follow a hierarchical structure. A *keyword hierarchy* (or a *keyword ontology*) is a strict partial order $\prec$ on the set $L$ of all keywords. We assume that there exists a meta symbol $* \in L$ which is the most general category generalizing all keywords. For two keywords $w_1, w_2 \in L$, if $w_1 \prec w_2$, $w_1$ is *more general* than $w_2$. For example, `fruit` $\prec$ `apple`. We write $w_1 \preceq w_2$ if $w_1 \prec w_2$ or $w_1 = w_2$.

If an aggregate keyword query cannot find a non-trivial, complete matching answer, alternatively, we may loosen the query requirement to allow a keyword in the query to be matched by another keyword in data that is similar in the keyword hierarchy.

There are two ways of hierarchical keyword matching.

*Specialization* matches a keyword $w$ in a query with another keyword $w'$ which is a descendant of $w$ in the keyword hierarchy. For example, keyword "fruit" in a query may be matched by keyword "apple" in the hierarchy.

*Generalization* matches $w$ with keyword $w''$ which is an ancestor of $w$. For example, keyword "apple" in a query maybe matched by keyword "fruit" in the hierarchy.

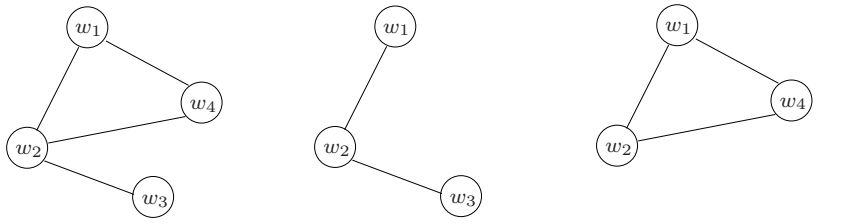
Limited by space, we only discuss specialization here, partly because in many applications users may be interested in more specific answers. However, our method can be easily extended to support generalization matching.

Using a keyword graph index, we can greedily loosen the query requirement to allow hierarchical keyword matching. For a query containing $m$ keywords $w_1, \ldots, w_m$ which does not have a non-trivial complete match answer, we can specialize a query keyword to a more specific one according to the hierarchical structure, and find the complete match answers for the new set of keywords. The search continues until some non-trivial minimal answers can be found.

When we choose a query keyword to specialize, we should consider two factors.

First, by specializing $w_1$ to $w_2$ such that $w_1 \prec w_2$, how many new edges can be added to the query subgraph if $w_1$ is replaced by $w_2$? Since keywords are specialized in the matching, the edges between $w_1$ and some other query keywords can be retained. We are interested in the new edges that can be added to the query subgraph by specializing the keyword which can help to find a clique and a non-trivial answer.

*Example 7 (Specialization)* Consider a keyword graph index in Figure 3(a). There are 4 distinct keywords. For a query $Q = \{w_1, w_2, w_3\}$, its corresponding keyword subgraph $G(Q)$ is shown in Figure 3(b). Clearly, keywords in $Q$ only have 2 edges connecting them, and they do not form a clique structure,




(a) A keyword graph index (b) A keyword subgraph $G(Q)$ (c) A keyword subgraph $G(Q)_{w_3 \to w_4}$

Fig. 3: An example of keyword specialization.

thus there are no exact answers to $Q$. However, we can replace keyword $w_3$ with another keyword $w_4$ if $w_4$ is a descendent of $w_3$. The keyword subgraph $G(Q)_{w_3 \to w_4}$ is shown in Figure 3(c). All the keywords in Figure 3(c) form a clique structure. By replacing keyword $w_3$ with keyword $w_4$, there are 3 edges connecting the keywords in the query.

Second, how many descendant keywords of $w_1$ are also descendants of $w_2$ in the keyword hierarchy? Formally, we consider the ratio $\frac{\alpha(w_2)}{\alpha(w_1)}$ where $\alpha(w)$ is the number of leaf keywords in the keyword hierarchy that are descendants of $w$. The larger the ratio, the less generality is lost in the specialization. By taking the above two factors into account, we can assign an "extendability" score for each keyword in the query.

**Definition 6 (Extendability)** Given a keyword $w_1$ in a query keyword subgraph $G(Q)$ and a keyword hierarchy, the **extendability score** of $w_1$ with respected to its descendant $w_2$ is

$$ext(w_1 \to w_2) = \frac{\alpha(w_2)}{\alpha(w_1)}(|E(G(Q)_{w_1 \to w_2})| - |E(G(Q))|),$$

where $|E(G(Q))|$ is the number of edges in graph $G(Q)$ and $|E(G(Q)_{w_1 \to w_2}|$ is the number of edges in graph $G(Q)$ by replacing $w_1$ with $w_2$. The **extendability** of $w_1$ is the maximum extendability value of $ext(w_1, desc(w_1))$ where $desc(w_1)$ is a descendant of $w_1$ in the keyword hierarchy.

We can calculate the extendability of each keyword in a query. The keyword $w$ with the highest extendability is selected, and the keyword $w$ is replaced by $w'$ such that $ext(w, w')$ is the maximum among all descendant of $w$. We call this a *specialization* of the query.

After a specialization of the query, we examine if a clique exists in the new query keyword subgraph. The more rounds of specialization, the more uncertainty introduced to the query. If a clique exists and the clique leads to a non-trivial answer, we terminate the specialization immediately and return the query answers; if not, we conduct another round of specialization greedily.

When the specialization of all keywords reaches the leaf keywords in the hierarchy, no more specialization can be conducted. In such an extreme case, the hierarchical keyword matching fails.

Too many rounds of specializations may not lead to a good answer loyal to the original query objective. A user may specify the maximum number of specializations allowed as a control parameter. Our method can be easily extended to accommodate such a constraint.

There are some other interesting directions to explore in the future study. For example, a keyword query typically is very short, which on average only contains two or three keywords[1]. As a result, keyword queries in practice are often ambiguous. To overcome this disadvantage, a useful tool is to develop some error tolerant keyword matching algorithms such that keywords in the query do not need to be exactly matched in the results. As another example, tuples in the relational databases may be semantically related. It is interesting to examine whether the semantic relationship among those tuples can be utilized to improve the performance of keyword search on relational databases.

## 7 Empirical Study

In this section, we report a systematic empirical study to evaluate our aggregate keyword search methods using both real datasets and synthetic datasets. All the experiments were conducted on a PC computer running the Microsoft Windows XP SP2 Professional Edition operating system, with a 3.0 GHz Pentium 4 CPU, 1.0 GB main memory, and a 160 GB hard disk. The programs were implemented in C/C++ and were compiled using Microsoft Visual Studio .Net 2005.

### 7.1 Results on Real Dataset IMDB

We first describe the data set we used in the experiments. Then, we report the experimental results.

#### *7.1.1 The IMDB Dataset*

The Internet Movie Database (IMDB) dataset (`http://www.imdb.com/interfaces/`) has been used extensively in the previous work on keyword search on relational databases (He et al 2007; Luo et al 2007; Ding et al 2007). We use this dataset to empirically evaluate our aggregate keyword search methods.

We downloaded the whole raw IMDB data. We preprocessed the dataset by removing duplicate records and missing values. We converted a subset of

[1] `http://www.keyworddiscovery.com/keyword-stats.html`

| Attribute | Description | Cardinality |
|---|---|---|
| Movie | movie title | 134,080 |
| Director | director of the movie | 62,443 |
| Actor | leading actor of the movie | 68,214 |
| Actress | leading actress of the movie | 72,908 |
| Country | producing country of the movie | 73 |
| Language | language of the movie | 45 |
| Year | producing year of the movie | 67 |
| Genre | genres of the movie | 24 |
| Keyword | keywords of the movie | 15,224 |
| Location | shooting locations of the movie | 1,049 |

Table 3: The IMDB database schema and its statistical information.

its raw text files into a large relational table. The schema of the table and the statistical information are shown in Table 3.

We used the first 7 attributes as the dimensions in the search space. Some attributes such as "actor" and "actress" may have more than one value for one specific movie. To use those attributes as dimensions, we picked the most frequent value if multiple values exist on such an attribute in a tuple. After the preprocessing, we obtained a relational table of $134,080$ tuples.

Among the 10 attributes in the table, we use "genre", "keyword" and "location" as the text attributes, and the remaining attributes as the dimensions. Table 3 also shows the total number of keywords for each text attribute and the cardinality of each dimension. On average each tuple contains 9.206 keywords in the text attributes.

In data representation, we adopted the popular packing technique (Beyer and Ramakrishnan 1999). A value on a dimension is mapped to an integer between 1 and the cardinality of the dimension. We also map keywords to integers.

### *7.1.2 Index Construction*

Both the simple nested loop approach in Algorithm 1 and the fast maximum join approach in Algorithm 2 need to maintain the inverted list index for each keyword in the table. The total number of keywords in the IMDB dataset is $16,297$. The average length of those inverted lists is 87.1, while the largest length is $13,442$. The size of the whole inverted list is 5.5 MB.

We used Algorithm 3 to construct the keyword graph index. The construction took 107 seconds. Among $16,297$ keywords, $305,412$ pairs of keywords (that is, 0.23%) have non-trivial answers. The average size of the minimal answer set on edges (that is, average number of minimal answers per edge) is 26.0. The size of the whole keyword graph index is 103.3 MB.

Both the inverted list index and the keyword graph index can be maintained on the disk. We can organize the indexes into chunks, while each chunk only contains a subset of the whole index. In the query answering, only those

| Query ID | Query Keywords in Text Attributes | | | # answers |
|---|---|---|---|---|
| | Genre | Keyword | Location | |
| $Q_1$ | Action | explosion | / | 1,404 |
| $Q_2$ | Comedy | / | New York | 740 |
| $Q_3$ | / | mafia-boss | Italy | 684 |
| $Q_4$ | Action | explosion, war | / | 2,109 |
| $Q_5$ | Comedy | family | New York | 1,026 |
| $Q_6$ | / | mafia-boss, revenge | Italy | 407 |
| $Q_7$ | Action | explosion, war | England | 724 |
| $Q_8$ | Comedy | family, christmas | New York | 308 |
| $Q_9$ | Crime | mafia-boss, revenge | Italy | 341 |
| $Q_{10}$ | Action | explosion, war, superhero | England | 215 |
| $Q_{11}$ | Comedy | family, christmas, revenge | New York | 0 |
| $Q_{12}$ | Crime | mafia-boss, revenge, friendship | Italy | 43 |

Table 4: The aggregate keyword queries.

related inverted lists, or the related keywords and the answer sets on the related edges need to be loaded into the main memory. Since the number of keywords in a query is often small, the I/O cost is low.

Recently, most of the modern computers have large main memories (typically several gigabytes). In general, the whole keyword graph index is small enough to be held in the main memory. As shown later in the performance comparisons, the query answering algorithm using the keyword graph index can achieve the smallest response time. In practice, which query answering algorithm to choose really depends on the requirements on the response time and the memory usage. If the response time is more crucial than the memory usage, the keyword graph algorithm is a better choice.

We will examine the index construction cost in detail using synthetic datasets.

#### *7.1.3 Aggregate Keyword Queries – Complete Matches*

We tested a large number of aggregate keyword queries, which include a wide variety of keywords and their combinations. We considered factors like the frequencies of keywords, the size of the potential minimal answers to be returned, the text attributes in which the keywords appear, etc.

We first focus our performance evaluation on a representative test set of 12 queries here. Our test set has 12 queries (denoted by $Q_1$ to $Q_{12}$) with query length ranging from 2 to 5. Among them, each length contains 3 different queries. Note that the query $Q_{i+3}$ ($1 \leq i \leq 9$) is obtained by adding one more keyword to the query $Q_i$. In this way, we can examine the effect when the number of query keywords increases. The queries are shown in Table 4.

We list in Table 4 the number of minimal answers for each query. When the number of query keywords increases from 2 to 3, the number of minimal answers may increase. The minimal answers to a two keyword query are often quite specific tuples. When the third keyword is added, for example, query $Q_4$,

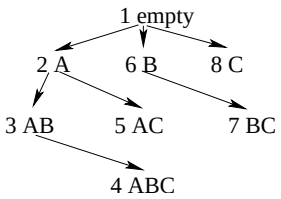



Fig. 4: The processing tree of the baseline algorithm.

the minimal answers have to be generalized. One specific answer can generate many ancestors by combining with other tuples. Query $Q_3$, however, is an exception, since most of the movies containing keywords "mafia-boss" and "Italy" also contain "revenge". Thus, many minimal answers to $Q_3$ are also minimal answers to $Q_6$.

When the number of query keywords increases further (for example, more than 3 keywords), the number of minimal answers decreases. When some new keywords are added into the query, the minimal answers are becoming much more general. Many combinations of tuples may generate the same minimal answers. The number of possible minimal answers decreases.

As discussed in Section 3.1, the existing studies about keyword search on relational databases and our paper address different types of keyword queries on relational databases. The existing methods cannot be extended straightforwardly to tackle the aggregate keyword search problem. Thus, we do not conduct the performance comparison with those existing keyword search algorithms on relational databases. Alternatively, to examine the efficiency of our methods, we compare our methods to the following baseline algorithm which is an extension of the conventional iceberg cube computation (Beyer and Ramakrishnan 1999). Consider Example 3 and the corresponding lattice in Figure 1, we compute the cells using the processing tree in Figure 4. The numbers in Figure 4 indicate the orders in which the baseline algorithm visits the group-bys.

The baseline algorithm first produces the empty group-by. Next, it partitions on dimension $A$. Since there are two distinct values on dimension $A$ (that is, $a_1$ and $a_2$), we produce two partitions $\langle a_1 \rangle$ and $\langle a_2 \rangle$. Then, the baseline algorithm recurses on partition $\langle a_1 \rangle$. The $\langle a_1 \rangle$ partition is aggregated and produces a cell $c_1 = (a_1, *, *)$ for the $A$ group-by. To examine whether $c_1$ is a valid answer, we need to examine whether all the query keywords $w_1$ and $w_2$ appear in some tuples in the cover $Cov(c_1)$. To efficiently achieve that goal, for each keyword $w$ in the database, we maintain an inverted list $IL(w)$. For each keyword $w$ in the query, we scan $Cov(c_1)$ once to examine if there exists at least one tuple in $IL(w)$. If it is not the case, we conclude that cell $c_1$ is not a valid answer, and the baseline algorithm stops the current loop, since there is no need to further examine the children in the processing tree. If all keywords in the query appear in $Cov(c_1)$, $c_1$ is a valid answer, and it is placed in the candidate answer set. We continue to partition the $\langle a_1 \rangle$ partition on dimen-

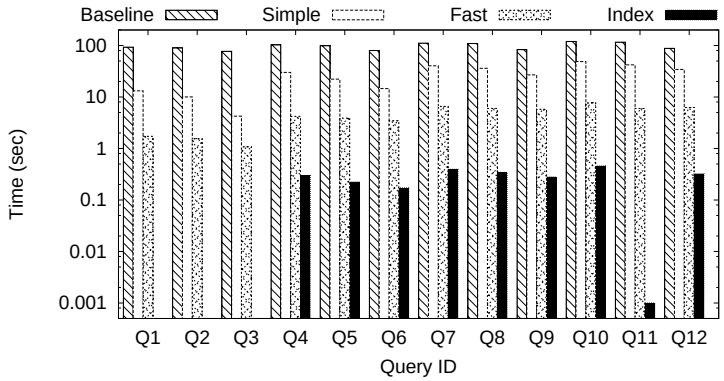



Fig. 5: Query time of queries with different sizes.

sion $B$. It recurses on the $\langle a_1, b_1 \rangle$ partition and generates a cell $c_2 = (a_1, b_1, *)$ for the $AB$ group-by. The baseline algorithm again examines whether $c_2$ is a valid answer. If $c_2$ is an valid answer due to $c_2$ being more specific than $c_1$, $c_1$ cannot be a minimum answer, thus $c_1$ is removed from the candidate answer set and $c_2$ is placed into the candidate answer set. Similarly, we process partitions $\langle a_1, b_1, c_1 \rangle$ and $\langle a_1, b_1, c_2 \rangle$. The baseline algorithm then recurses on the $\langle a_1, b_2 \rangle$ partition. When this is completed, it partitions the $\langle a_1 \rangle$ partition on dimension $C$ to produce the $\langle a_1, C \rangle$ aggregates.

After we obtain the candidate answer set, the baseline algorithm needs to scan the candidate answer set once and remove any non-minimum answers. Finally, the baseline algorithm can find the complete set of minimum answers for an aggregate keyword query.

To load the whole keyword graph index into the main memory, it took 9.731 seconds. On average, the loading time for one specific edge in the keyword graph index is 0.032 milliseconds, which is a very small number. In the experimental evaluation, we focus only on the efficiency of the query answering algorithms. Thus, for the response times with respect to different queries in the following analysis, we ignore the I/O costs and only consider the response time of the query answering algorithms.

Figure 5 compares the query answering time for the 12 queries using the baseline algorithm (denoted by *Baseline*), the simple nested loop algorithm (Algorithm 1, denoted by *Simple*), the fast maximum join algorithm (Algorithm 2, denoted by *Fast*), and the keyword graph index method (denoted by *Index*). The time axis is drawn in logarithmic scale.

The baseline algorithm performs the worst among the four methods. The major reason is that the baseline algorithm needs to compute the whole lattice. Moreover, the baseline algorithm needs to consider all the tuples in the database. In practice, given a keyword query, only a part of tuples are needed to be considered. The simple nested loop algorithm performs better than the baseline algorithm, but it is the worst among the three methods other than the baseline algorithm. The fast algorithm adopts several speed up strategies,

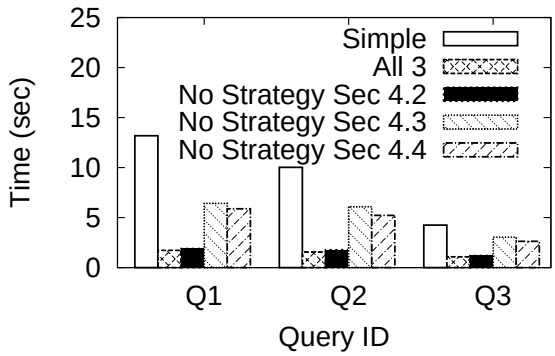



Fig. 6: Effectiveness of each pruning strategy in the maximum-join method.

thus its query time is about an order of magnitude shorter than the simple nested loop algorithm. The keyword graph index based algorithm is very fast in query answering. When the number of query keywords is 2, we can directly obtain the minimal answers from the edge labels, and thus the query answering time is ignorable. When the number of query keywords is at least 3, the query time is about 20 times shorter than the fast maximum join algorithm. One reason is that the keyword graph index method already calculates the minimum answers for each pair of keywords, thus, given $m$ query keywords, we only need to conduct maximum joins on $m - 1$ edges. Another reason is that only the minimal answers stored on the edges participate in the maximum joins, which are much smaller than the total number of tuples involved in the maximum join methods.

Generally, when the number of query keywords increases, the query time increases, since more query keywords lead to a larger number of maximum join operations. It is interesting to find that when the number of query keywords increases, the query time of the baseline algorithm does not increases greatly. The reason is that the size of queries is not the bottleneck in the baseline algorithm. The major time-consuming part of the baseline algorithm is to compute the whole lattice.

Using the keyword graph index, the query time for $Q_{11}$ is exceptionally small. The reason is that by examining the keyword graph index, the 5 query keywords in $Q_{11}$ do not form a clique in the graph index, thus we even do not need to conduct any maximum join operations. The results confirm that the query answering algorithm using keyword graph index is efficient and effective.

We also examine the effectiveness of the speed up strategies in Algorithm 2. Limited by space, we only show the basic cases $Q_1$, $Q_2$ and $Q_3$ each of which has 2 keywords. In Figure 6, for each query, we tested the query answering time of adopting all three speed up strategies, as well as leaving one strategy out.

All the speed up strategies contribute to the reduction of query time. However, their contributions are not the same. The strategy of reducing matching

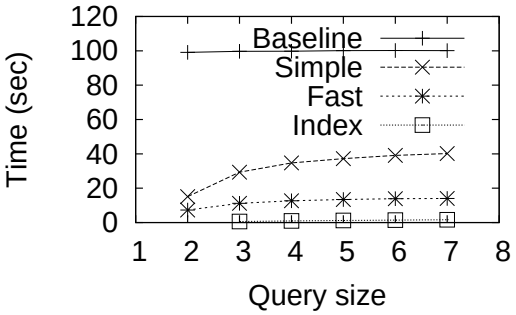



Fig. 7: Query time of queries with different sizes (random queries).

candidates (Section 4.3) contributes the best, since it can reduce the candidate tuples to be considered greatly. The fast minimal answer checking strategy (Section 4.4) removes non-minimal answers. Comparing to the nested-loop based superset checking, it is much more efficient. The effect of pruning exactly matching tuples (Section 4.2) is not as much as the other two, since the exact matching tuples are not frequently met.

In the above analysis, we focus on a set of 12 representative queries. To examine the performance of the query answering algorithms in general situations, we also conduct the performance evaluation on a large set of randomly generated queries.

We first examine the effect of the query size. The query size plays an important role in query answering. In practice (e.g., Web search), the number of keywords in a keyword query is relatively small (e.g, 2 or 3 words in Web search queries[2]). As explained in Section 4, the running time of our algorithms are highly related to two factors: (1) the rounds of maximum joins (the maximum join algorithm)/the number of edges to be examined in the graph index (the keyword graph algorithm); (2) the size of the intermediate minimal answers after each round of maximum joins. When the number of keywords in the query increases, the first factor obviously increases as well. However, the second factor does not increases always. At some stage, the number of intermediate minimal answers achieves the maximum value; then it decreases even more keywords appear in the query. This can be verified based on the results shown in Table 4. In general, in the IMDB dataset, the size of the minimal answers is largest when the number of keywords in the query is equal to 3.

In Figure 7, we plot the running time of the two algorithms with respect to different sizes of queries. For each fixed query size, the running time refers to the average time of 100 randomly generated queries. When the number of keywords is small (e.g., less than or equal to 3), the increase of the running time is large; however, when the number of keywords is large (e.g., larger than 4), the increase of the running time is small. The results in Figure 7 also verify

[2] `http://www.keyworddiscovery.com/keyword-stats.html`

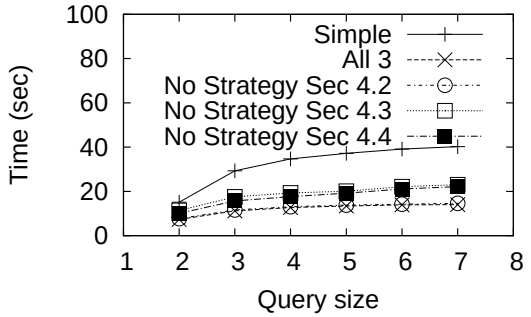



Fig. 8: Effectiveness of each pruning strategy in the maximum-join method (random queries).

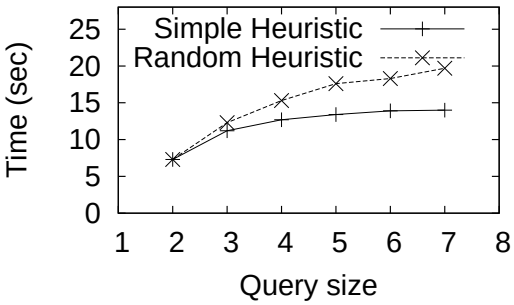



Fig. 9: Effectiveness of the heuristic for the orders of maximum joins (random queries).

that even if the size of the query is very large, our algorithms still can achieve reasonably good performance.

We next examine the effect of each pruning strategy in the maximum-join method using randomly generated queries. In Figure 8, we plot the running time of the queries with different sizes in terms of adopting all three speed up strategies, as well as leaving one strategy out. For each fixed query size, the running time refers to the average time of 100 randomly generated queries. The results are similar to those in Figure 6.

As we mentioned in Section 4.5, for a keyword query which contains $m$ different keywords, we need to conduct $m-1$ rounds of maximum joins. The orders of maximum joins is crucial to the performance of the query answering algorithms. In this paper, we adopt a simple heuristic, that is, in each round of maximum joins, we pick two inverted lists with the smallest sizes.

The heuristic used in our algorithm is very simple, and it achieves relatively good performance. For a performance comparison, we compared our simple heuristic to a random selection method (that is, for a query containing $m$

keywords, we randomly pick two words and get the corresponding lists of tuples for the next round of maximum joins). In Figure 9, we show the performance of the maximum-join algorithm using the **Simple Heuristic** we developed in this paper, as well as that using the **Random Method**. All the pruning strategies are adopted. The simple heuristic clearly outperforms the random method.

| Query size | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| # queries | 100 | 100 | 100 | 100 | 100 | 100 |
| # queries that the traditional keyword search methods cannot find a single matching tuple | 61 | 82 | 87 | 88 | 93 | 95 |
| # queries that the aggregate keyword search cannot find a non-trivial answer | 2 | 6 | 7 | 11 | 13 | 13 |

Table 5: The effectiveness of the aggregate keyword queries.

We examine the effectiveness of the proposed aggregate keyword search using randomly generated queries. We vary the number of keywords in the query from 2 to 7; and for each fixed query size, we randomly generate 100 different queries. For each query, we examine whether there exists a single tuple matching all the keywords in the query. If there does, the traditional keyword search algorithms on relational databases can retrieve some results; if not, the traditional keyword search algorithms cannot find any results. Table 5 shows the results. When the number of keywords in a query is not very small (e.g., more than 3), the majority of queries cannot find any single tuple matching all the keywords in the query. However, the aggregate keyword search is still able to find some useful results even if the query keywords do not appear in a tuple. In the worst case, the aggregate keyword search may just return a trivial answer (that is, an answer with $*$ on all the dimensions). We count the number of queries when the aggregate keyword search only returns a trivial answer, and the results are shown in Table 5. In general, only a small number of queries cannot find a non-trivial answer using the aggregate keyword search. The results clearly indicate that the aggregate keyword search is quite useful in practice.

### *7.1.4 Partial Keyword Matching Queries*

We first examine the effectiveness of the partial keyword matching. We use the same set of randomly generated queries in Section 7.1.3. As discussed in Section 7.1.3, some aggregate keyword queries may return a trivial answer that is not informative to users. For those queries which cannot find a non-trivial answer using the aggregate keyword search, we adopt the partial keyword matching technique. We count the maximal number of keywords in the query that can be matched using the partial keyword matching. If most of the keywords in the query can be matched, the answers returned by the partial keyword matching may still be interesting to users. Table 6 shows the

| Query size | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| # queries | 100 | 100 | 100 | 100 | 100 | 100 |
| # queries that the aggregate keyword search cannot find a non-trivial answer | 2 | 6 | 7 | 11 | 13 | 13 |
| Percentage of queries that the aggregate keyword search cannot find a non-trivial answer | 2% | 6% | 7% | 11% | 13% | 13% |
| Average # matched keywords using partial keyword matching | 1 | 2 | 2.86 | 3.73 | 4.23 | 4.85 |

Table 6: The effectiveness of the partial keyword matching.

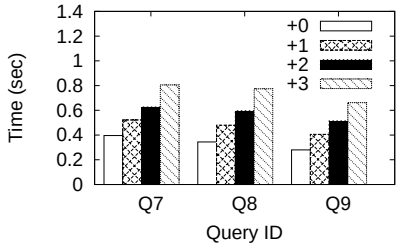



Fig. 10: Query time for partial keyword matching queries.

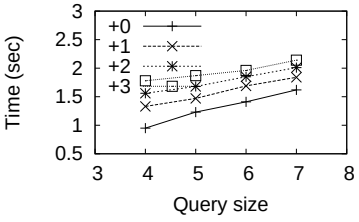



Fig. 11: Query time for partial keyword matching queries (random queries).

results. In general, the majority of keywords in the query are matched using the partial keyword matching. The results indicate that the partial keyword matching is useful to find some good answers even if the aggregate keyword search cannot find a non-trivial answer.

We conducted the experiments to evaluate our partial keyword matching queries using the keyword graph index. To simulate the partial keyword matching scenario, we used queries $Q_7$, $Q_8$ and $Q_9$ in Table 4 as the base queries and add some irrelevant keywords into each query. Specifically, for each query, we manually added 1, 2 and 3 irrelevant keywords into the query to obtain the extended queries, and make sure that any of those irrelevant keywords do not have non-trivial complete match answers together with the keywords in the base query.

Our method returns the answers to the base queries as the partially match answers to the extended queries. The experimental results confirm that our partial matching method is effective. Moreover, Figure 10 shows the runtime of partial matching query answering. When the number of irrelevant query keywords increases, the query time increases, because more cliques need to be considered in the query answering.

We also examine the performance of the partial matching using randomly generated queries. Figure 11 shows the results of the running time with respect to different sizes of queries. For each randomly generated query, we randomly added 1, 2 and 3 irrelevant keywords into the query to obtain the extended queries. The results are similar to those in Figure 10.

### *7.1.5 Hierarchical Keyword Matching Queries*

| Query size | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| # queries | 100 | 100 | 100 | 100 | 100 | 100 |
| # queries that the aggregate keyword search cannot find a non-trivial answer | 2 | 6 | 7 | 11 | 13 | 13 |
| Percentage of queries that the aggregate keyword search cannot find a non-trivial answer | 2% | 6% | 7% | 11% | 13% | 13% |
| Average # rounds of specialization | 2 | 2.5 | 2.71 | 3.09 | 3.46 | 3.62 |

Table 7: The effectiveness of the hierarchical keyword matching.

We first examine the effectiveness of the hierarchical keyword matching. We use the same set of randomly generated queries in Section 7.1.3. As discussed in Section 7.1.3, some aggregate keyword queries may return a trivial answer that is not informative to users. For those queries which cannot find a non-trivial answer using the aggregate keyword search, we adopt the hierarchical keyword matching technique. We count the number of rounds that the specification needs to be conducted. Too many rounds of specializations may not lead to a good answer loyal to the original query objective. Table 7 shows the results. In general, only a few rounds of specification are needed for the hierarchical keyword matching technique to find a non-trivial answer. The results indicate that the hierarchical keyword matching is useful to find some good answers even if the aggregate keyword search cannot find a non-trivial answer.

We also conducted experiments to evaluate our hierarchical keyword matching methods using the keyword graph index. A critical issue is to build a keyword hierarchy for the keywords in the data set. Since the number of keywords is large, we adopted the WordNet taxonomy (Fellbaum 1998) as the hierarchy. WordNet is a semantic lexicon for the English language. It groups English words into sets of synonyms. We queried the WordNet taxonomy database for each keyword, and built the hierarchical relations among them according to the WordNet synonyms.

To simulate the hierarchical keyword matching scenario, we used queries $Q_7$, $Q_8$ and $Q_9$ in Table 4 as the base queries, and replaced some keywords in those queries by their ancestors in the hierarchy. Specifically, for each query, we manually replaced 1, 2 or 3 keywords with some ancestor keywords to obtain the extended queries, and make sure that the keywords in each extended query do not form a clique in the keyword graph index.

When only 1 keyword is replaced with an ancestor keyword, among the 3 queries, all the returned results match the answers to the base queries. When the number of keywords to be replaced increases, however, the results become weaker. In the case of 2 keywords are replaced, only the extended queries based on $Q_7$ and $Q_8$ return the expected answers. In the case of 3 keywords are replaced, only the extended query based on $Q_7$ returns the expected answer. The reason is, when more keywords are replaced, when we select a keyword to

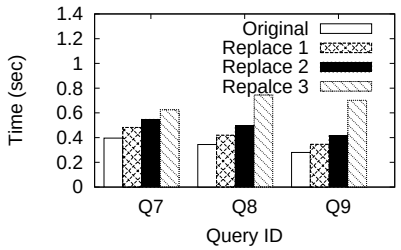



Fig. 12: Query time for hierarchical keyword matching queries.

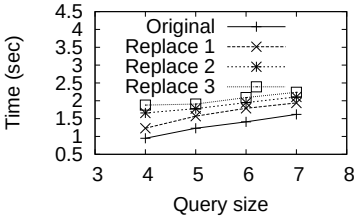



Fig. 13: Query time for hierarchical keyword matching queries (random queries).

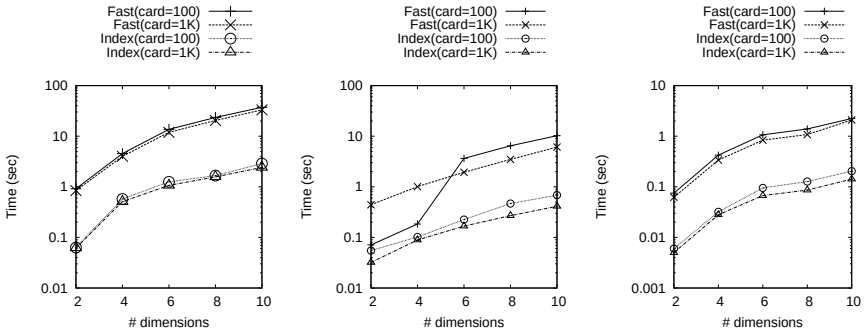
specialize, the number of choices increases, thus the probability to return the expected answers becomes smaller. However, our hierarchical keyword matching algorithm can still find meaningful answers. For example, $Q_4$ is a 3-keyword query {"Animation", "Animal", "USA"}. There is no exact answer in the table for this query. We can find an approximate keyword matching {"Animation", "lions", "Walt Disney World"}, and one answer with the movie title "The Lion King" will be returned.

Figure 12 shows the query answering time in hierarchical matching. Similar to the partial matching method, when the number of query keywords to be replaced increases, the query time increases. This is because more calculation for extendability score is needed. Generally, the hierarchical keyword matching queries can be answered efficiently using the keyword graph index.

We also examine the performance of the hierarchical matching using randomly generated queries. Figure 13 shows the results of the running time with respect to different sizes of queries. For each randomly generated query, we randomly replaced 1, 2 or 3 keywords with some ancestor keywords to obtain the extended queries. The results are similar to those in Figure 10.

## 7.2 Results on Synthetic Datasets

To test the efficiency and the scalability of our aggregate keyword search methods, we generated various synthetic data sets. In those data sets, we randomly generated 1 million tuples for each data set. We varied the number of dimensions from 2 to 10. We tested the data sets of the cardinalities 100 and $1,000$ in each dimension. Since the number of text attributes does not affect the keyword search performance, for simplicity, we only generated 1 text attribute. Each keyword appears only once in one tuple. We fixed the number of keywords in the text attribute for each tuple to 10, and varied the total number of keywords in the data set from $1,000$ to $100,000$. The keywords are distributed uniformly except for the experiments in Section 7.2.2. Thus, on average the number of tuples in the data set that contain one specific keyword varied from




Number of keywords is $1K$. Number of keywords is $10K$. Number of keywords is $100K$.

Fig. 14: Query time of the two methods on different synthetic data sets.

10,000 to 100. We also use the packing technique (Beyer and Ramakrishnan 1999) to represent the data sets.

### *7.2.1 Efficiency and Scalability*

To study the efficiency and the scalability of our aggregate keyword search methods, we randomly picked 10 different keyword queries, each of which contains 3 different keywords. Figure 14 shows the query answering time.

The keyword graph index method is an order of magnitude faster than the fast maximum join algorithm. The simple nested loop method is an order of magnitude slower than the fast maximum join method. To make the figures readable, we omit them here.

The number of dimensions, the cardinality of the dimensions and the total number of keywords affect the query answering time greatly. In general, when the number of dimensions increases, the query answering time increases. First, the maximum join cost is proportional to the dimensionality. Moreover, the increase of runtime is not linear. As the dimensionality increases, more minimal answers may be found, thus more time is needed. When the cardinality increases, the query answering time decreases. The more diverse the dimensions, the more effective of the pruning powers in the maximum join operations. The total number of keywords in the text attribute highly affects the query time. The more keywords in the table, on average less tuples contain a keyword.

We generated the keyword graph index using Algorithm 3. The keyword graph generation is sensitive to the number of tuples in the table. We conducted the experiments on 10 dimensional data with cardinality of 1,000 on each dimension, set the total number of keywords to 10,000, and varied the number of tuples from 0.25 million to 1.25 million. The results on runtime are shown in Figure 15. The runtime increases almost linearly as the number of tuples increases, since the number of maximum join operations and the number of answers generated both increase as the number of tuples increases.

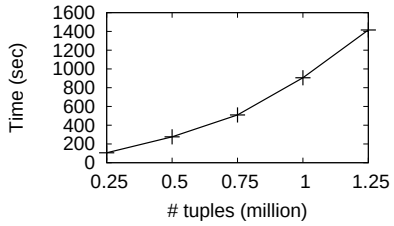



Fig. 15: Running time for the keyword graph index generation.

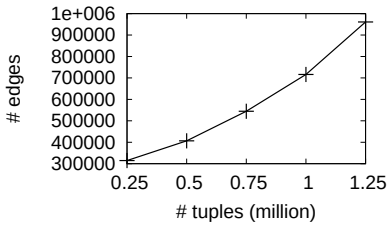



Fig. 16: The number of edges in the keyword graph index.

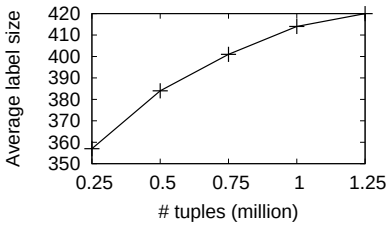



Fig. 17: The average length of edge labels in the keyword graph index.

Figures 16 and Figure 17 examine the size of the keyword graph index with respect to the number of tuples, where the settings are the same as Figure 15. To measure the index size, we used the number of edges in the graph and the average number of minimal answers on each edge. Generally, when the number of tuples increases, the number of edges increases because the probability that two keywords have a non-trivial answer increases. Meanwhile, the average number of minimal answers on each edge also increases because more tuples may contain both keywords.

### *7.2.2 Skewness*

The aggregate keyword search methods are sensitive to skewness in data. In all of the previous experiments, the data was generated uniformly. We ran an experiment on the synthetic data set with 1 million tuples, 10 dimensions with cardinality $1,000$, and a total number of $10,000$ keywords. We varied the skewness simultaneously in all dimensions. We used the Zipf distribution to generate the skewed data. Zipf uses a parameter $\alpha$ to determine the skewness. When $\alpha = 0$, the data is uniform, and as $\alpha$ increases, the skewness increases rapidly: at $\alpha = 3$, the most frequent value occurs in about 83% of the tuples. We randomly picked 10 different keyword queries with query size 3. The average query answering time is shown in Figure 18. The performance of the two methods becomes as the skewness on dimensions increases. However, the keyword graph index method still performs well.

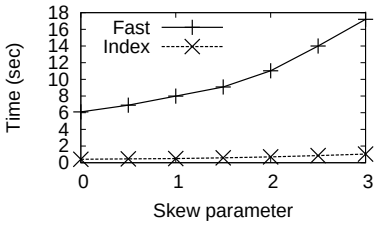



Fig. 18: Skew on dimensional attributes.

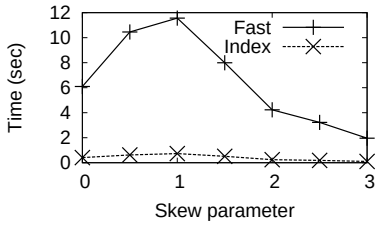



Fig. 19: Skew on text attribute.

Skewness may occur on text attributes, too. We ran another experiment on the data set with 1 million tuples, 10 dimensions with cardinality of $1,000$, and a total number of $10,000$ keywords. We made the skewness happen in the text attribute only. We also used a Zipf distribution. We randomly picked 10 different keyword queries with query size 2, in which one keyword has a high frequency and the other does not. The average query answering time is shown in Figure 19. When the parameter $\alpha$ is small (for example, 0.5), the query answering time increases when the data becomes skewed. This is because the tuples containing the frequent keyword in a query increases dramatically. However, when $\alpha$ is further larger, the query answering time decreases because the number of tuples containing the infrequent keyword in a query decreases dramatically. The query answering time is dominated by the infrequent keyword.

In summary, our experimental results on both real data and synthetic data clearly show that aggregate keyword queries on large relational databases are highly feasible. Our methods are efficient and scalable in most of the cases. Particularly, the keyword graph index approach is effective.

## 8 Conclusions

In this paper, we identified a novel type of aggregate keyword queries on relational databases. We showed that such queries are useful in some applications. We developed the maximum join approach and the keyword graph index approach. Moreover, we extend the keyword graph index approach to address partial matching and hierarchical matching. We reported a systematic performance study using real data and synthetic data to verify the effectiveness and the efficiency of our methods.

The techniques developed in this paper are useful in some other applications. For example, some techniques in this paper may be useful in KDAP (Wu et al 2007). As future work, we plan to explore extensions of aggregate keyword queries and our methods in those applications. Moreover, in some applications, a user may want to rank the minimal answers in some meaningful way such as finding the top-$k$ minimal answers. It is interesting to extend our method to address such a requirement.

## References


Agrawal S, Chaudhuri S, Das G (2002) DBXplorer: A system for keyword-based search over relational databases. In: Proceedings of the 18th International Conference on Data Engineering (ICDE'02), IEEE Computer Society, Washington, DC, USA, pp 5–16

Amer-Yahia S, Case P, Rölleke T, Shanmugasundaram J, Weikum G (2005) Report on the DB/IR panel at sigmod 2005. ACM, New York, NY, USA, vol 34, pp 71–74

Balmin A, Hristidis V, Papakonstantinou Y (2004) Objectrank: authority-based keyword search in databases. In: Proceedings of the Thirtieth international conference on Very large data bases (VLDB'04), VLDB Endowment, pp 564–575

Beyer K, Ramakrishnan R (1999) Bottom-up computation of sparse and iceberg cube. In: Proceedings of the 1999 ACM SIGMOD international conference on Management of data (SIGMOD'99), ACM, New York, NY, USA, pp 359–370

Bhalotia G, Hulgeri A, Nakhe C, Chakrabarti S, Sudarshan S (2002) Keyword searching and browsing in databases using banks. In: Proceedings of the 18th International Conference on Data Engineering (ICDE'02), IEEE Computer Society, pp 431–440

Chaudhuri S, Das G (2009) Keyword querying and ranking in databases. PVLDB 2(2):1658–1659

Chaudhuri S, Ramakrishnan R, Weikum G (2005) Integrating DB and IR technologies: What is the sound of one hand clapping? In: Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR'05), pp 1–12

Chen Y, Wang W, Liu Z, Lin X (2009) Keyword search on structured and semi-structured data. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD'09), ACM, pp 1005–1010

Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to Algorithms. McGraw-Hill Higher Education

Daoud M, Lechani LT, Boughanem M (2009) Towards a graph-based user profile modeling for a session-based personalized search. Knowledge and Information Systems 21(3):365–398

Ding B, Yu JX, Wang S, Qin L, Zhang X, Lin X (2007) Finding top-k min-cost connected trees in databases. In: Proceedings of the 23rd IEEE International Conference on Data Engineering (ICDE'07), IEEE Computer Society, Washington, DC, USA, pp 836–845

Dreyfus SE, Wagner RA (1972) The steiner problem in graphs. Networks 1:195–?07

Fang M, Shivakumar N, Garcia-Molina H, Motwani R, Ullman JD (1998) Computing iceberg queries efficiently. In: Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98), New York, NY, pp 299–310

Fellbaum C (ed) (1998) WordNet: an electronic lexical database. MIT Press

Feng Y, Agrawal D, Abbadi AE, Metwally A (2004) Range Cube: Efficient cube computation by exploiting data correlation. In: Proc. 2004 Int. Conf. Data Engineering (ICDE'04), Boston, MA, pp 658–669

Garey MR, Johnson DS (1979) Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA

Gong Z, Liu Q (2009) Improving keyword based web image search with visual feature distribution and term expansion. Knowledge and Information Systems 21(1):113–132

Gray J, Bosworth A, Layman A, Pirahesh H (1996) Data cube: A relational operator generalizing group-by, cross-tab and sub-totals. In: Proc. 1996 Int. Conf. Data Engineering (ICDE'96), New Orleans, Louisiana, pp 152–159

Han J, Pei J, Dong G, Wang K (2001) Efficient computation of iceberg cubes with complex measures. In: Proc. 2001 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'01), Santa Barbara, CA, pp 1–12

Harman D, Baeza-Yates R, Fox E, Lee W (1992) Inverted files. In: Information retrieval: data structures and algorithms, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, pp 28–43

He H, Wang H, Yang J, Yu PS (2007) Blinks: ranked keyword searches on graphs. In: Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD'07), ACM, New York, NY, USA, pp 305–316

Henzinger M, Motwani R, Silverstein C (2003) Challenges in web search engines. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03), pp 1573–1579

Hristidis V, Papakonstantinou Y (2002) Discover: Keyword search in relational databases. In: Proceedings of the 28st international conference on Very large data bases (VLDB'02), Morgan Kaufmann, pp 670–681

Hristidis V, Gravano L, Papakonstantinou Y (2003) Efficient IR-style keyword search over relational databases. In: Proceedings of the 29st international conference on Very large data bases (VLDB'03), pp 850–861

Kacholia V, Pandit S, Chakrabarti S, Sudarshan S, Desai R, Karambelkar H (2005) Bidirectional expansion for keyword search on graph databases. In: Proceedings of the 31st international conference on Very large data bases (VLDB'05), ACM, pp 505–516

Kimelfeld B, Sagiv Y (2006) Finding and approximating top-k answers in keyword proximity search. In: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS'06), ACM, New York, NY, USA, pp 173–182

Li G, Ooi BC, Feng J, Wang J, Zhou L (2008) Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD'08), ACM, New York, NY, USA, pp 903–914

Liu F, Yu C, Meng W, Chowdhury A (2006) Effective keyword search in relational databases. In: Proceedings of the 2006 ACM SIGMOD international conference on Management of data (SIGMOD'06), ACM, New York, NY, USA, pp 563–574

Liu Z, Chen Y (2007) Identifying meaningful return information for xml keyword search. In: Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD'07), ACM, New York, NY, USA, pp 329–340

Luo Y, Lin X, Wang W, Zhou X (2007) Spark: top-k keyword query in relational databases. In: Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD'07), ACM, New York, NY, USA, pp 115–126

Ng RT, Wagner AS, Yin Y (2001) Iceberg-cube computation with PC clusters. In: Proc. 2001 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'01), Santa Barbara, CA

Park J, goo Lee S (2010) Keyword search in relational databases. Knowledge and Information Systems (Online First) , dOI: 10.1007/s10115-010-0284-1

Qin L, Yu JX, Chang L (2009a) Keyword search in databases: the power of rdbms. In: Proceedings of the 35th SIGMOD International Conference on Management of Data (SIGMOD'09), ACM Press, Providence, Rhode Island, USA, pp 681–694

Qin L, Yu JX, Chang L, Tao Y (2009b) Querying communities in relational databases. In: Proceedings of the 25th International Conference on Data Engineering (ICDE'09), IEEE, pp 724–735

Taha K, Elmasri R (2010) Bussengine: a business search engine. Knowledge and Information Systems 23(2):153–197

Tong H, Faloutsos C, Pan JY (2008) Random walk with restart: fast solutions and applications. Knowledge and Information Systems 14(3):327–346

Vu QH, Ooi BC, Papadias D, Tung AKH (2008) A graph method for keyword-based selection of the top-k databases. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD'08), ACM, New York, NY, USA

Weikum G (2007) DB&IR: both sides now. In: Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD'07), ACM, New York, NY, USA, pp 25–30

Wu P, Sismanis Y, Reinwald B (2007) Towards keyword-driven analytical processing. In: Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD'07), ACM, New York, NY, USA, pp 617–628

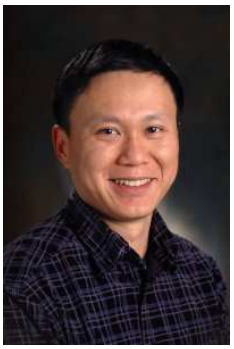
Xin D, Han J, Li X, Wah BW (2003) Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In: Proc. 2003 Int. Conf. on Very Large Data Bases (VLDB'02), Berlin, Germany, pp 476–487

Yu B, Li G, Sollins K, Tung AKH (2007) Effective keyword-based selection of relational databases. In: Proceedings of the 2007 ACM SIGMOD international conference on Man-

agement of data (SIGMOD'07), ACM, New York, NY, USA, pp 139–150
Zhou B, Pei J (2009) Answering aggregate keyword queries on relational databases using minimal group-bys. In: Proceedings of the 12th International Conference on Extending Database Technology (EDBT'09), Saint-Petersburg, Russia

## Author biographies

**Bin Zhou** received his B.Sc. degree in Computer Science from Fudan University, China, in 2005 and his M.Sc. degree in Computing Science from Simon Fraser University, Canada, in 2007. He is currently a Ph.D. candidate at the School of Computing Science at Simon Fraser University, Canada. His research interests lie in graph analysis, graph mining, data privacy, and their relations to Web-scale data management and mining, as well as their applications in Web search engines.



**Jian Pei** is an Associate Professor at the School of Computing Science at Simon Fraser University, Canada. His research interests can be summarized as developing effective and efficient data analysis techniques for novel data intensive applications. He is currently interested in various techniques of data mining, Web search, information retrieval, data warehousing, online analytical processing, and database systems, as well as their applications in social networks, health-informatics, business and bioinformatics. His research has been supported in part by government funding agencies and industry partners. He has published prolifically and served regularly for the leading academic journals and conferences in his fields. He is an associate editor of ACM Transactions on Knowledge Discovery from Data (TKDD) and an associate editor-in-chief of IEEE Transactions of Knowledge and Data Engineering (TKDE). He is a senior member of the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronics Engineers (IEEE). He is the recipient of several prestigious awards.