Answering Aggregate Keyword Queries on Relational Databases Using Minimal Group-bys*

Bin Zhou
School of Computing Science
Simon Fraser University, Canada
bzhou@cs.sfu.ca

Jian Pei
School of Computing Science
Simon Fraser University, Canada
jpei@cs.sfu.ca

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, and further extend our methods to allow partial matches. 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.

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. 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 (e.g., 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 (e.g., Texas instead of the whole country) and a short period (e.g., December instead of year 2008).

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 key-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. 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

*The research was supported in part by an NSERC Discovery grant and an NSERC Discovery Accelerator Supplements grant. 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.

EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.

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.

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 (e.g., matching m' of m keywords ($m' \leq m$) in a query) are allowed.

Last, we empirically evaluate our techniques using both real data sets and synthetic data sets. 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 for partial matching. A systematic performance study is reported in Section 7. Section 8 concludes the paper.

2. AGGREGATE KEYWORD QUERIES

To formulate the problem, we use the terminology in online analytic processing (OLAP) and data cubes [11].

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, $Cov(*,$ Texas, $*, *, *) \supseteq Cov($December, 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 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 need 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 $\mathcal{D}$, i.e., $c[A] = *$ 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 $t[A]$.* ∎

For example, 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 (i.e., 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.* ∎

It is well known that all 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.

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

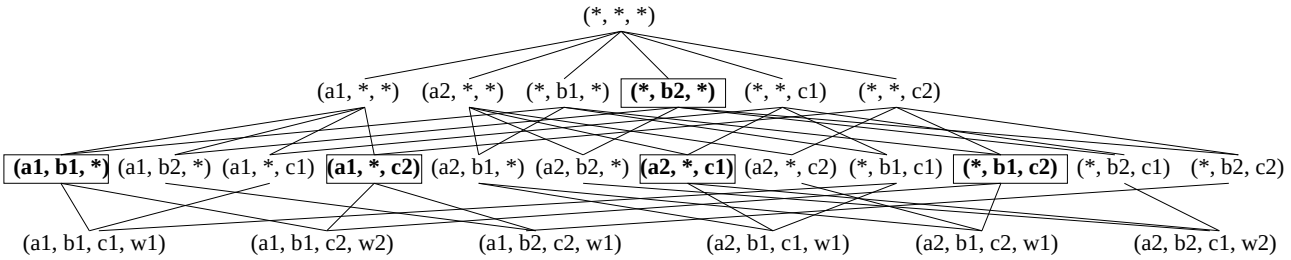


Figure 1: The aggregate lattice on ABC.

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 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 [5, 23, 2]. A few critical challenges have been identified such as how to address the flexibility in scoring and ranking models. [5, 23] provide 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.

DBXplorer [1] 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 [16] 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.* [15] developed efficient methods which can handle queries with both AND and OR semantics and exploited ranking techniques to retrieve top-k answers.

BANKS [4] 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 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, [17] introduces the bidirectional expansion techniques to improve the search efficiency on large graph databases. Various effective ranking criteria and search methods are also developed, such as [19, 20, 7].

In addition, BLINKS [14] builds a bi-level index for fast keyword search on graphs. The quality of approximation in top-k keyword proximity search is studied in [18]. [26] uses a keyword relationship matrix to evaluate keyword relationships in distributed databases. Most recently, [22] extends [26] by summarizing each database using a keyword relationship graph, which can help to select top-k most promising databases effectively in query processing.

All the previous work about keyword search on relational databases looks for *individual tuples* (or a set of tuples interconnected by primary key-foreign key relationships) matching the set of keywords in the query. The existing methods do not consider aggregate cells (group-bys). Therefore, the existing studies 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. Some of the existing methods may be extended to compute joining networks where tuples from the same table are joined (i.e., self-join of a table). However, such extensions cannot compute the minimal answers to aggregate keyword queries. Moreover, the number of joining networks generated by the self-join can be much larger than the number of minimal answers due to the monotonicity of aggregate cells.

3.2 Keyword-Driven Analytical Processing

Keyword-driven analytical processing (KDAP) [24] 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 (e.g., a star schema). In the explore phase, for each subspace, the aggregated values for some pre-defined measure are calculated and the top-k interesting group-by attributes to partition the subspace are found.

For instance, as an example in [24], 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 [24] does 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 2, aggregate keyword search finds aggregate cells in a data cube lattice (i.e., 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 [11]. In [8], Fang *et al.* proposed iceberg queries which find in a cube lattice the aggregate cells satisfying some given constraints (e.g., 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 [3] exploits monotonic constraints like `COUNT(*)` $\geq v$ and conducts a bottom-up search (i.e., from the most general aggregate cells to the most specific ones). Han *et al.* [12] tackle non-monotonic constraints like `AVG(*)` $\geq v$ by using some weaker but monotonic constraints in pruning. More efficient algorithms for iceberg cube computation are proposed in [25, 9]. The problem of iceberg cube computation on distributed network environment is investigated in [21].

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 most-general-to-most-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 [13] 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 $t = t_x \vee_{\mathcal{A}} t_y$ 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 $B \notin \mathcal{A}$, $t[B] = *$.* ∎

For example, in Table 2, $(a_1, b_1, c_1, \{w_1, w_3\}) \vee_{ABC} (a_1, b_2, c_2, \{w_2, w_5\}) = (a_1, *, *, *)$. We call this operation maximum join since it keeps the common values between the two operant tuples on as many attributes as possible. 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.

Corollary 2 (Properties). *The maximal-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 of 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 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.

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
```

To remove the non-minimal answers, the second step takes time $O(\prod_{i=1}^{m} |IL_{\mathcal{C}}(w_i)|^2)$. 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 $t_2 \in IL(w_2)$ such that $t_1[\mathcal{D}] = t_2[\mathcal{D}]$ are found, $t_1 \vee t_2$ 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 so far 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 $D \in \mathcal{D}$ such that $t_1[D] \neq t[D]$, $t_2[D] \neq t_1[D]$, then $t_1 \vee t_2$ is not a minimal answer to the query.*

PROOF. *For every attribute $D \in \mathcal{D}$ such that $t_1[D] \neq t[D]$, since $t_1[D] \neq t_2[D]$, $(t_1 \vee t_2)[D] = *$. On every other attribute $D' \in \mathcal{D}$ such that $t_1[D'] = t[D']$, 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 3 (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.

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 a 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.

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, which is inefficient. Here, we propose a method using inverted lists.

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*

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

and $Descendant(t) = \cap_{D \in \mathcal{D}: t[D] \neq *} IL_{D=t[D]}$. 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 $Descendant(t)$. 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.

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

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
```

If a database is large or the number of keywords in an aggregate keyword query is not small, 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 for efficient query answering. For example, Yu *et al.* [26] propose 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.* [22] extend [26] 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. 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.*

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 easily see that it is not sufficient.

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 (i.e., $m = 2$), the minimal answers can be found directly from edge (w_1, w_2) since they are materialized. If the query involves more than 2 keywords (i.e., $m \geq 3$), 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 $O(|Answer(Q_1)| \cdot |Answer(Q_2)|)$.

By using Lemma 1 repeatedly, to answer query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, \ldots, w_m\})$, 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 wants to minimize the sum of the edge weights. Several greedy algorithms, such as Prim's algorithm and Kruskal's algorithm [6], 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.

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 $Q_{u,v} =$

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 ∈ T do
3:    initialize the candidate tuple set to Cand = ∅;
4:    let Cand = ∪_D IL_{D=t[D]};
5:    let W_t be the set of keywords contained in t;
6:    for each tuple t' ∈ Cand do
7:       if t = t' then
8:          for each pair w1, w2 ∈ W_t do
9:             add t to edge (w1, w2), and remove non-
               minimal answers on edge (w1, w2) (Sec-
               tion 4.4);
10:         end for
11:      else
12:         let W_t' be the set of keywords contained in t';
13:         r = t ∨ t';
14:         for each pair w1 ∈ W_t − W_t' and w2 ∈ W_t' − W_t
            do
15:            add r to edge (w1, w2);
16:            remove non-minimal answers on edge (w1, w2)
               (Section 4.4);
17:         end for
18:      end if
19:   end for
20: end for
```

$(\mathcal{T}, \mathcal{T}, \{u, v\})$ 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. PARTIAL KEYWORD MATCHING

The methods in Sections 4 and 5 look for *complete matches*, i.e., 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 a solution to handle queries which do not have a non-trivial answer or even an answer. The idea is to allow partial matches (e.g., matching m' of m keywords ($m' \leq m$)).

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. By loosening the matching requirement, the returned minimal answers may be meaningful in practice.

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 an 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 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 [10], in practice, aggregate keyword queries often contain a small number of keywords (e.g., 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.

7. EMPIRICAL STUDY

In this section, we report a systematic empirical study to evaluate our aggregate keyword search methods using both real data sets and synthetic data sets. 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 Data Set IMDB

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

7.1.1 The IMDB Data Set

The Internet Movie Database (IMDB) data set (`http://www.imdb.com/interfaces/`) has been used extensively in the previous work on keyword search on relational databases [14, 20, 7]. We use this data set to empirically evaluate our aggregate keyword search methods.

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

Query ID	Query Keywords in Text Attributes			# minimal 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.

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.

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. One text attribute may contain more than 1 keyword. On average each tuple contains 9.206 keywords in the text attributes.

In data representation, we adopted the popular packing technique [3]. 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 data set is 16,297. The average length of those inverted lists is 87.1, while the largest length is 13,442.

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

Both the inverted list index and the keyword graph index can be maintained on disk. In query answering, only those related inverted lists, or the related keywords and the answer sets on the related edges need to be loaded into main memory. Since the number of keywords in a query is often small, the I/O cost is low.

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

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. Limited by space, we only focus 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, e.g., query Q_4, 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 (e.g., 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.

Figure 2 compares the query answering time for the 12 queries using the simple nested loop algorithm (Algorithm 1, denoted by **Simple**), the fast maximum join algorithm (Al-

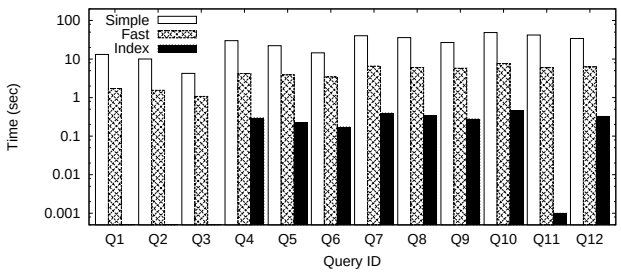


Figure 2: Query time of queries with different sizes.

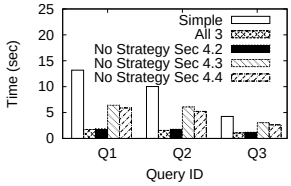


Figure 3: Effectiveness of each pruning strategy in the maximum-join method.

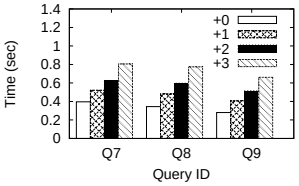


Figure 4: Query time for partial keyword matching queries.

gorithm 2, denoted by **Fast**), and the keyword graph index method (denoted by **Index**). The time is in logarithmic scale.

The simple nested loop algorithm performs the worst among the three methods. The fast algorithm adopted several speedup strategies, thus the 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. 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} does 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. The results are similar when queries contain more than 2 keywords. In Figure 3, 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 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 helpful as the other two, since the exact matching tuples are not frequently met.

7.1.4 Partial Keyword Matching Queries

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 be 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 made sure that each of those irrelevant keywords does 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 4 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.

7.2 Results on Synthetic Data Sets

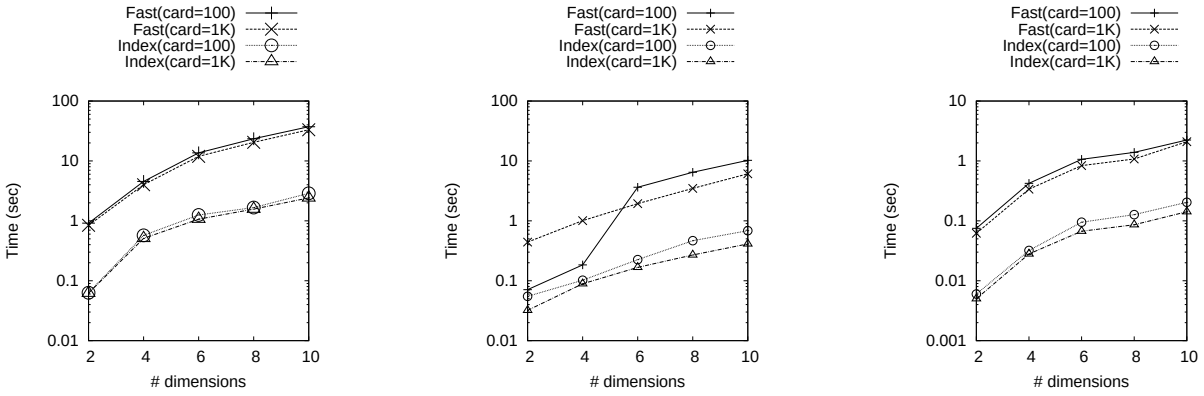
To test the efficiency and the scalability of our aggregate keyword search methods, we generated various synthetic data sets. In those data sets, we randomly generated 1 million tuples for each data set. We varied the number of dimensions from 2 to 10. We tested the data sets of cardinalities 100 and $1,000$ 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 100 to $10,000$. We also use the packing technique [3] 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 5 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 the simple nested loop method 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



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

Figure 5: Query time of the two methods on different synthetic data sets.

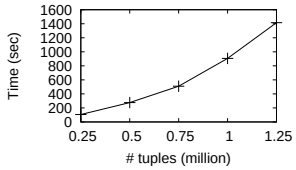


Figure 6: Running time for the keyword graph index generation.

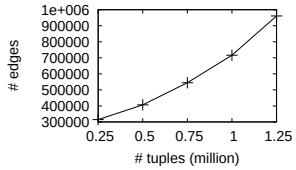


Figure 7: The number of edges in the keyword graph index.

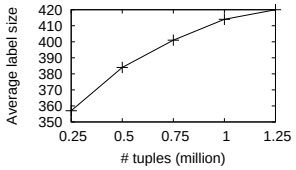


Figure 8: The average length of edge labels in the keyword graph index.

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 6. 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.

Figures 7 and Figure 8 examine the size of the keyword graph index with respect to the number of tuples, where the settings are the same as Figure 6. 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

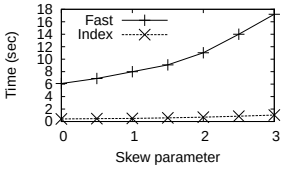


Figure 9: Skew on dimensional attributes.

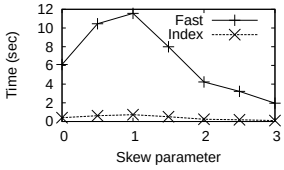


Figure 10: Skew on text attribute.

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 α to determine the skewness. When $\alpha = 0$, the data is uniform, and as α 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 9. The performance of the two methods degrades as the skewness on dimensions increases. However, the keyword graph index method still performs well.

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 10. When the parameter α is small (e.g., 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 α increases further, 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.

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. 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 [24]. 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. Last, it is useful and promising to adopt ontology-based keyword matching. For example, a keyword "fruit" in a query may be matched to a word "apple" in the data.

9. REFERENCES

[1] S. Agrawal *et al.* DBXplorer: A system for keyword-based search over relational databases. In *ICDE'02*.

[2] S. Amer-Yahia *et al.* Report on the DB/IR panel at sigmod 2005. *SIGMOD Record*, 34(4):71–74, 2005.

[3] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *SIGMOD'99*.

[4] G. Bhalotia *et al.* Keyword searching and browsing in databases using banks. In *ICDE'02*.

[5] S. Chaudhuri *et al.* Integrating DB and IR technologies: What is the sound of one hand clapping? In *CIDR'05*.

[6] T. H. Cormen *et al. Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[7] B. Ding *et al.* Finding top-k min-cost connected trees in databases. In *ICDE'07*.

[8] M. Fang *et al.* Computing iceberg queries efficiently. In *VLDB'98*.

[9] Y. Feng *et al.* Range Cube: Efficient cube computation by exploiting data correlation. In *ICDE'04*.

[10] M. Garey and D. Johnson. *Computers and Intractability: a Guide to The Theory of NP-Completeness*. Freeman and Company, New York, 1979.

[11] J. Gray *et al.* Data cube: A relational operator generalizing group-by, cross-tab and sub-totals. In *ICDE'96*.

[12] J. Han *et al.* Efficient computation of iceberg cubes with complex measures. In *SIGMOD'01*.

[13] D. Harman *et al.* Inverted files. In *Information retrieval: data structures and algorithms*, pages 28–43, Upper Saddle River, NJ, USA, 1992. Prentice-Hall, Inc.

[14] H. He *et al.* BLINKS: ranked keyword searches on graphs. In *SIGMOD'07*.

[15] V. Hristidis *et al.* Efficient ir-style keyword search over relational databases. In *VLDB'03*.

[16] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB'02*.

[17] V. Kacholia *et al.* Bidirectional expansion for keyword search on graph databases. In *VLDB'05*.

[18] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *PODS'06*.

[19] F. Liu *et al.* Effective keyword search in relational databases. In *SIGMOD'06*.

[20] Y. Luo *et al.* Spark: top-k keyword query in relational databases. In *SIGMOD'07*.

[21] R. T. Ng *et al.* Iceberg-cube computation with PC clusters. In *SIGMOD'01*.

[22] Q. H. Vu *et al.* A graph method for keyword-based selection of the top-k databases. In *SIGMOD'08*.

[23] G. Weikum. DB&IR: both sides now. In *SIGMOD'07*.

[24] P. Wu *et al.* Towards keyword-driven analytical processing. In *SIGMOD'07*.

[25] D. Xin *et al.* Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *VLDB'02*.

[26] B. Yu *et al.* Effective keyword-based selection of relational databases. In *SIGMOD'07*.