TopCells: Keyword-Based Search of Top-k Aggregated Documents in Text Cube

Bolin Ding, Bo Zhao, Cindy Xide Lin, Jiawei Han, Chengxiang Zhai
Department of Computer Science, University of Illinois at Urbana-Champaign
{bding3, bozhao3, xidelin2, hanj, czhai}@uiuc.edu

***Abstract*— Previous studies on supporting keyword queries in RDBMSs provide users with a ranked list of relevant linked structures (*e.g.* joined tuples) or individual tuples. In this paper, we aim to support keyword search in a data cube with text-rich dimension(s) (so-called *text cube*). Each document is associated with structural dimensions. A cell in the text cube aggregates a set of documents with matching dimension values on a subset of dimensions. Given a keyword query, our goal is to find the top-k most relevant cells in the text cube. We propose a relevance scoring model and efficient ranking algorithms. Experiments are conducted to verify their efficiency.**

I. Introduction

The boom of Internet has given rise to an ever increasing amount of text data associated with multiple dimensions (attributes), for example, customer reviews in shopping websites (*e.g.* Amazon) are always associated with attributes like price, model, and rate. A traditional OLAP *data cube* can be naturally extended to summarize and navigate structured data together with unstructured text data. Such a cube model is called *text cube* [1]. A *cell* in the text cube aggregates a set of documents with matching attribute values on a subset of dimensions.

Keyword query, popularly used to retrieve useful information from a collection of plain documents, is being extended to RDBMSs, *e.g.*, [2], [3], [4], [5], [6], [7]. Given a set of keywords, existing methods find relevant tuples or joins of tuples (*e.g.*, linked by foreign keys) that contain the keywords.

In this paper, we study the problem of *keyword-based top-k search in text cube*: *given a keyword query, to find the top-k most relevant cells in a text cube*. Different from keyword search in RDBMSs (finding linked tuples), keyword search in text cube discovers relationship between structural dimensions and text data, and outputs relevant cells (aggregations of tuples), which represent certain objects and are easy to be browsed.

Example 1.1 (Motivation): Table I shows a database of customer reviews of laptops. *Customer Review* is the text data, whereas *Brand*, *Model*, *CPU*, and *OS* are structural dimensions.

Jim, a marketing analyst, wants to find which laptops are commented as *light weight* and *powerful performance*. He types a set of keywords: {"*light*", "*weight*", "*powerful*"}. Using IR techniques, the system can rank all the customer reviews and output the most relevant ones. However, when there are many customer reviews relevant to the query, Jim has to browse through a lot of reviews and summarize different opinions by himself. Is it more desirable that a system provides users with "aggregated information", such as "*Acer AOA110 laptops are usually lightweight and have powerful performance*", than returning individual reviews?

The work was supported in part by NASA grant NNX08AC35A, the U.S. National Science Foundation grants IIS-08-42769 and IIS-09-05215, and the Air Force Office of Scientific Research MURI award FA9550-08-1-0265, an HP Research grant, and Microsoft research Women's Scholarship. Any opinions, findings, and conclusions expressed here are those of the authors and do not necessarily reflect the views of the funding agencies.

TABLE I
Motivation Example

Brand	Model	CPU	OS	Customer Review
Acer	AOA110	1.6GHz	Linux	d_1: **light weight** of only 2.2 lb, fun and **powerful** features
Acer	AOA110	1.8GHz	XP	d_2: **weight** just over 0.9 kg, **powerful** Intel Processor
Asus	EEE PC	1.8GHz	XP	d_3: ...in pearl white style, images are sharp, disk is large

In text cube, a cell, in the form of (Brand = Acer, Model = AOA110, CPU = *, OS = *), aggregates $\{d_1, d_2\}$, representing an object "Acer AOA110 laptops". It can be seen that this cell is more relevant to Jim's query than another cell (Brand = *, Model = *, CPU = 1.8GHz, OS = XP). Our goal is to provide Jim with a ranked list of relevant cells, instead of individual customer reviews. ■

Overview. Section II defines the keyword search problem in text cube: we introduce text cube of multidimensional text data, and define the keyword query language and relevance model in text cube. In Section III, we propose two approaches. The first one scans the inverted index of each keyword in the query only once to compute the relevance scores of all cells. The second one optimizes the search order and prune the search space by estimating relevance score upper bounds in subspaces, so as to explore as few cells in the text cube as possible for finding the top-k answers. Experimental study is reported in Section IV, followed by related work in Section V.

II. Keyword Queries in Text Cube

A. A Data Cube Model for Text Data

A set $\mathcal{D}$ of documents is stored in an *n-dimensional database* $\mathcal{DB} = (\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n, \mathcal{D})$. Each row of $\mathcal{DB}$ is in the form of $r = (a_1, a_2, \ldots, a_n, \mathsf{d})$: let $r[\mathcal{A}_i] = a_i \in \mathcal{A}_i$ be the *value* of *dimension* $\mathcal{A}_i$, and $r[\mathcal{D}] = \mathsf{d}$ be the *document* in this row.

The *data cube* model extended to the above multidimensional text database is called *text cube* [1]. Several important concepts are introduced as follows.

Cell and Aggregated Document. In the text cube built on $\mathcal{DB}$, a *cell* is in the form of $C = (v_1, v_2, \ldots, v_n : \mathsf{D})$, where either $v_i \in \mathcal{A}_i$ (a value of dimension $\mathcal{A}_i$) or $v_i = *$ (the dimension $\mathcal{A}_i$ is *aggregated* in C), and D (the *aggregated document* in C) is the set of documents in the rows of $\mathcal{DB}$ having the same dimension values as C on all the non-$*$ dimensions. Formally, $\mathsf{D} = \{r[\mathcal{D}] \mid \text{for } r \in \mathcal{DB} \text{ s.t. } r[\mathcal{A}_i] = v_i \text{ if } v_i \neq *\}$. We use $C[\mathcal{A}_i]$ to denote the value v_i of dimension $\mathcal{A}_i$ in the cell C, and $C[\mathcal{D}]$ to denote the aggregated document D in C. Define

the *support* of a cell C, denoted by $|C|$, to be the number of documents in $C[\mathcal{D}]$. A cell is said to be *empty* if $C[\mathcal{D}] = \varnothing$. For simplicity, a cell is also written as $C = (v_1, v_2, \ldots, v_n)$.

Cells with m non-$*$ dimensions are called *m-dim cells*. An n-dim cell is called a *base cell*, with no aggregated dimension.

Ancestor and Descendant. Cell C' is an *ancestor* of C (or C is a *descendant* of C') iff "$\forall i : C'[\mathcal{A}_i] \neq * \;\Rightarrow\; C'[\mathcal{A}_i] = C[\mathcal{A}_i]$". We use $\mathsf{ans}(C)$ to denote the set of ancestors of a cell C, and $\mathsf{des}(C')$ to denote the set of descendants of a cell C'.

Parent and Child. *Parents* and *children* are immediate ancestors and descendants of a cell, respectively. Cell C' is a *parent* of C (or C is a *child* of C') iff (i) $C' \in \mathsf{ans}(C)$, and (ii) C' is an i-dim cell while C is an $(i+1)$-dim cell. Let $\mathsf{par}(C)$ ($\mathsf{chd}(C')$) denote the set of parents of a cell C (children of a cell C').

Example 2.1 (Text Cube): Consider the text cube built on the database $\mathcal{DB}$ in Table I. (Brand = Acer, Model = AOA110, CPU = 1.8GHz, OS = *: $\{\mathsf{d}_2\}$) is a 3-dim cell. Both (Brand = Acer, Model = AOA110, CPU = *, OS = *: $\{\mathsf{d}_1, \mathsf{d}_2\}$) and (Brand = *, Model = *, CPU = 1.8GHz, OS = *: $\{\mathsf{d}_2, \mathsf{d}_3\}$) are its ancestors, and the former is also its parent. ■

B. Keyword Search Problem in Text Cube

A *keyword query* is a set of terms, *i.e.*, $\mathsf{q} = \{t_1, t_2, \ldots, t_{|\mathsf{q}|}\}$. Given a keyword query q and a minimum support requirement minsup, the goal of *keyword search problem* is to *find the top-k cells C's s.t. supports $|C| \geq \mathsf{minsup}$ with the top-k highest relevance scores* $\mathrm{rel}(\mathsf{q}, C)$ *in the text cube of $\mathcal{DB}$*. We allow users to specify k and support threshold minsup, if they are interested in only top-k relevant cells which are "popular" enough.

Relevance Scoring Formula. The *relevance score* $\mathrm{rel}(\mathsf{q}, C)$ of a cell C w.r.t. the given query q is defined as a function of q and the aggregated document $C[\mathcal{D}]$. Recall an aggregated document $C[\mathcal{D}]$ is a set of documents. Here, we treat every document in $C[\mathcal{D}]$ equally, and define $\mathrm{rel}(\mathsf{q}, C)$ as the average of all the relevance scores of documents in $C[\mathcal{D}]$:

$$\mathrm{rel}(\mathsf{q}, C) = \frac{1}{|C[\mathcal{D}]|} \sum_{\mathsf{d} \in C[\mathcal{D}]} s(\mathsf{q}, \mathsf{d}). \tag{1}$$

$s(\mathsf{q}, \mathsf{d})$ is the relevance score of a document d w.r.t. q:

$$s(\mathsf{q}, \mathsf{d}) = \sum_{t \in \mathsf{q}} \ln \frac{N - \mathrm{df}_t + 0.5}{\mathrm{df}_t + 0.5} \frac{(k_1+1)\mathrm{tf}_{t,\mathsf{d}}}{k_1((1-b) + b\frac{\mathrm{dl}_\mathsf{d}}{\mathrm{avdl}}) + \mathrm{tf}_{t,\mathsf{d}}} \frac{(k_3+1)\mathrm{qtf}_{t,\mathsf{q}}}{k_3 + \mathrm{qtf}_{t,\mathsf{q}}}, \tag{2}$$

(Okapi weighting [8])

where $N = |\mathcal{DB}|$ (# rows), $\mathrm{tf}_{t,\mathsf{d}}$ is the *term frequency* of $t \in \mathsf{q}$ in d (the number of times t appearing in d), df_t is the number of documents in $\mathcal{DB}$ containing t, dl_d is the length of document d, avdl is the average length of documents in $\mathcal{DB}$, $\mathrm{qtf}_{t,\mathsf{q}}$ is the term frequency of t in q, and, k_1, b, k_3 are constants.

Our algorithms can also handle other formulas $s(\mathsf{q}, \mathsf{d})$ for scoring documents, *e.g.*, Pivoted normalization weighting.

Extended Form of Keyword Query. Users may want to retrieve top-k answers from a certain part of the text cube, by specifying a subset dimensions of interests and/or values of some dimensions (e.g. OS = XP). We omit the discussion on how to handle such extended forms of keyword queries, because adding such constraints actually shrinks the search space.

Challenges. The major computational challenge of this keyword search problem is the huge number (increasing exponentially w.r.t. the dimensionality) of cells in a text cube. Also, because the relevance score of a cell is query-dependent, it is hard to precompute these scores for all possible queries.

III. Algorithms for Searching Top-k Cells

Two algorithms for the keyword search problem are introduced. The first one scans inverted index once to compute relevance scores of all cells. The second one explore as few cells in text cube as possible to find the top-k answers.

A. Naive One-Scan Inverted Index Algorithm

Assume inverted index $\mathsf{IV}(t) = \{(r, \mathrm{tf}_{t,r[\mathcal{D}]}) \mid \mathrm{tf}_{t,r[\mathcal{D}]} > 0\}$ is built for each term t in $\mathcal{DB}$. Recall our goal is to rank all cells with support $|C| \geq \mathsf{minsup}$ in the text cube of $\mathcal{DB}$ according to relevance $\mathrm{rel}(\mathsf{q}, C)$ for the given query q.

We first compute the score $s(\mathsf{q}, \mathsf{d})$ of document d in each row r using the inverted indexes (lines 1). Then, for each row r (one scan), we can efficiently identify all the cells C's, s.t. $r[\mathcal{D}] \in C[\mathcal{D}]$ and support $|C| \geq \mathsf{minsup}$ (lines 2-3: note $|C|$ can be precomputed and materialized). These cells (*ancestors of* r, denoted by $\mathsf{ans}(r)$) are actually the ancestors of the base cell C_r that has the same dimension values as r ($C_r[\mathcal{A}] = r[\mathcal{A}]$ for any $\mathcal{A}$). Note $|\mathsf{ans}(r)| = 2^n$ in an n-dimensional text cube. We update $\mathrm{rel}(\mathsf{q}, C)$ using $s(\mathsf{q}, r[\mathcal{D}])$ for each C (line 4). Finally, if using a priority queue (*e.g.* heap) to maintain the cells, the top-k relevant ones can be output (line 5) at the end.

Algorithm 1 One-Scan Inverted Index Algorithm

1: Compute relevance score $s(\mathsf{q}, r[\mathcal{D}])$ for each row r.
2: **for each** row $r \in \mathcal{DB}$ **do**
3: **for each** cell $C \in \mathsf{ans}(r)$ and $|C[\mathcal{D}]| \geq \mathsf{minsup}$ **do**
4: Update $\mathrm{rel}(\mathsf{q}, C)$ using $s(\mathsf{q}, r[\mathcal{D}])$;
5: Output cells C's with the top-k highest $\mathrm{rel}(\mathsf{q}, C)$.

Algorithm 1 is efficient only when the number of dimensions is small (from 2 to 4). Because, to output the top-k, it actually scores all the non-empty cells (the number of non-empty cells increases exponentially w.r.t. the dimensionality).

B. Search-Space Ordering Algorithm

Our second algorithm explores as small number of cells in a text cube as possible to find the top-k answers. It is based on the following ideas: Although the search space is huge (especially if the dimensionality is reasonably large, say, ≥ 10), when only the top-k answers are interesting to users, only a small portion of the text cube is "closely" relevant to the given query q. So, (i) we estimate the upper bound of relevance scores in a subspace of the text cube, and (ii) we *order* and *prune* subspaces using these upper bounds, s.t. we can follow a shortcut to the top-k relevant cells with support $\geq \mathsf{minsup}$ in the search space, and output them in the order of relevance.

Two-Side Bound Property. To estimate the relevance upper bound of a subspace, we first introduce the *two-side bound property* of relevance scores in a text cube: the relevance of a cell is bounded by the max/min relevance of its children.

TABLE II
A 2-DIMENSIONAL TEXT DATABASE (BASE CELLS)

M	P	$C[\mathcal{D}]$	$\mathrm{rel}(\mathsf{q}, C)$
m1	p1	$\{\mathsf{d}_1\}$	8
m1	p2	$\{\mathsf{d}_2\}$	6
m1	p3	$\{\mathsf{d}_3, \mathsf{d}_7\}$	4
m2	p1	$\{\mathsf{d}_4\}$	8
m2	p2	$\{\mathsf{d}_5\}$	6
m2	p3	$\{\mathsf{d}_6, \mathsf{d}_8\}$	4

Lemma 1: Consider $\mathrm{rel}(\cdot,\cdot)$ in Equations (1), for any non-base cell C in text cube and any query q, there exist two children C_1 and C_2 of C s.t. $\mathrm{rel}(\mathsf{q}, C_1) \leq \mathrm{rel}(\mathsf{q}, C) \leq \mathrm{rel}(\mathsf{q}, C_2)$.

Proof: (Sketch) It suffices to prove, for any query q and any two disjoint document sets $\mathsf{D}_1, \mathsf{D}_2$ s.t. $\mathrm{rel}(\mathsf{q}, \mathsf{D}_1) \leq \mathrm{rel}(\mathsf{q}, \mathsf{D}_2)$, we have $\mathrm{rel}(\mathsf{q}, \mathsf{D}_1) \leq \mathrm{rel}(\mathsf{q}, \mathsf{D}_1 \cup \mathsf{D}_2) \leq \mathrm{rel}(\mathsf{q}, \mathsf{D}_2)$. ■

Bottom-up Aggregation. Our algorithm works in a bottom-up manner. Initially, the relevance score of any non-base cell is unknown. Starting from base cells (whose relevance scores are computed with the inverted indexes), in each iteration, it aggregates a highly-relevant cell into its parent, since a highly-relevant cell is likely to have highly-relevant children.

Partially-aggregated/Fully-aggregated Cells. In each bottom-up aggregation, we aggregate the cell C, *with max relevance and not aggregated yet*, into its parents C'. We update the *partial relevance score* $\overline{\mathrm{rel}}(\mathsf{q}, C')$ of each C' (in constant time), which is defined to be the *average relevance score of all the documents already aggregated into* C'. Cell C is a *$\mathcal{A}$-child of C'*, if C is a child of C' and they differ only on one dimension $\mathcal{A}$. When all the $\mathcal{A}$-children are aggregated into C' for some dimension $\mathcal{A}$, C' is said to be *fully-aggregated* (*i.e.* all documents in $C'[\mathcal{D}]$ have been aggregated into C'), and we can obtain the exact relevance score of C' as $\mathrm{rel}(\mathsf{q}, C') = \overline{\mathrm{rel}}(\mathsf{q}, C')$ now; before that, C' is said to be *partially-aggregated*.

Relevance Upper-Bound & Outputting Condition. The *upper-bound of relevance of all partially-aggregated cells* is estimated as $\max\{\overline{\mathrm{rel}}(\mathsf{q}, C'') : C''$ is partially-aggregated$\}$. The correctness can be proved based on Lemma 1 and how we chose the cell C to be aggregated in each iteration. So, a fully-aggregated cell C' with support $\geq \mathsf{minsup}$ can be can be output as one of the top-k cells, if *its relevance score $\geq$ the upper bound of relevance scores of all partially-aggregated cells*.

It is important to note that we do *not* aggregate all the children of a cell C' into it. Otherwise, some document will be aggregated into C' multiple times. It suffices to aggregate the $\mathcal{A}$-children of C' for some dimension $\mathcal{A}$. In our algorithm, $\mathcal{A}$-children for different dimensions $\mathcal{A}$ are kept and aggregated into C' separately. For example, suppose a cell $C' = (\mathrm{a1}, *, *)$ has four children $C_1 = (\mathrm{a1}, \mathrm{b1}, *)$, $C_2 = (\mathrm{a1}, \mathrm{b2}, *)$, $C_3 = (\mathrm{a1}, *, \mathrm{c1})$, and $C_4 = (\mathrm{a1}, *, \mathrm{c2})$. C' is fully-aggregated if either C_1 and C_2, or, C_3 and C_4 are aggregated into it.

Theorem 1: Given any query q against the text cube of $\mathcal{DB}$, Algorithm 2 outputs the top-k cells with the highest relevance scores and support $\geq \mathsf{minsup}$ in the non-increasing order of $\mathrm{rel}(\mathsf{q}, C)$.

More technique details and proofs are deferred to the full version. Example 3.1 shows the main ideas of how to find the top-k cells before we compute the relevance scores of all cells.

Example 3.1: Consider the 2-dimensional text database in Table II, and a keyword query $\mathsf{q} = \{\mathrm{w1}, \mathrm{w2}\}$ with $\mathsf{minsup} = 2$. Suppose the relevance score $\mathrm{rel}(\mathsf{q}, C)$ (defined in Equation (1)-(2)) of each base cell is computed as in the fourth column of Table II.

Algorithm 2 Search-Space Ordering Algorithm

$\mathsf{Q_A}$ / $\mathsf{Q_C}$: priority queues of fully-aggregated cells (in the descending order of relevance) to be aggregated / output
$\mathbf{top}(\mathsf{Q})$: the cell with the highest relevance in Q
$\mathsf{P_A}$: the set of partially-aggregated cells
Input: keyword query q, parameter k, and threshold minsup

```
1:  Compute relevance scores rel(q, C)'s for all base cells;
2:  cnt ← 0, Q_A ← {all base cells},
    Q_C ← {base cells with support ≥ minsup};
3:  while cnt < k do
4:    Let C be the top cell in Q_A, and delete C from Q_A;
5:    for each partially-aggregated parent C' ∈ par(C) do
6:      if C' ∉ P_A then insert C' into P_A;
7:      Aggregate C into C':
        Update partial relevance score rel-bar(q, C') based on C;
8:      if C' is fully-aggregated then
9:        Compute rel(q, C');
10:       Delete C' from P_A;
11:       Insert C' into Q_A;
          Insert C' into Q_C if support |C'| ≥ minsup;
12:   while rel(q, top(Q_C)) ≥ max_{C''∈P_A}{rel-bar(q, C'')} and
          rel(q, top(Q_C)) ≥ rel(q, top(Q_A)) do
13:     Output the top cell of Q_C, and delete it from Q_C;
14:     cnt ← cnt + 1;
```

We show how to maintain $\mathsf{Q_A}$, $\mathsf{Q_C}$, and $\mathsf{P_A}$ in the following table. The number in the bracket in the first column is the number of iterations, *e.g.*, $\mathsf{P_A}(2)$ means the status of $\mathsf{P_A}$ at the end of the 2^{nd} iteration; the number after each cell in the second column is the relevance score (or the partial relevance score in $\mathsf{P_A}$).

Initialization: All base cells are put into $\mathsf{Q_A}$. Two cells with support $\geq$ minsup are put into $\mathsf{Q_C}$. $\mathsf{P_A}$ is empty.	
$\mathsf{P_A}(0)$	$\varnothing$
$\mathsf{Q_A}(0)$	(m1,p1):8, (m2,p1):8, (m1,p2):6, (m2,p2):6, (m1,p3):4, (m2,p3):4
$\mathsf{Q_C}(0)$	(m1,p3):4, (m2,p3):4

1^{st} *iteration*: Cell (m1,p1) is popped from $\mathsf{Q_A}(0)$, and aggregated into its two parents, (m1,*) and (*,p1) (inserted into $\mathsf{P_A}(1)$). The partial relevance scores of (m1,*) and (*,p1) are computed as 8.	
$\mathsf{P_A}(1)$	(m1,*):8, (*,p1):8
$\mathsf{Q_A}(1)$	(m2,p1):8, (m1,p2):6, (m2,p2):6, (m1,p3):4, (m2,p3):4
$\mathsf{Q_C}(1)$	(m1,p3):4, (m2,p3):4

2^{nd} *iteration*: (m2,p1) is popped from $\mathsf{Q_A}(1)$, and aggregated into the two parents, (m2,*) and (*,p1): (m2,*) is inserted into $\mathsf{P_A}(2)$ with the partial relevance score computed as 8; (*,p1) are already in $\mathsf{P_A}(1)$. Then, (*,p1) is *fully aggregated*, since it has only two M-children, (m1,p1) and (m2,p1). So (*,p1) is deleted from $\mathsf{P_A}(1)$ and inserted into $\mathsf{Q_A}(2)$. It is also inserted into $\mathsf{Q_C}(2)$ for its support $\geq \mathsf{minsup}$. Finally, the top cell in $\mathsf{Q_C}(2)$, (*,p1), has relevance score no less than the maximum partial relevance score in $\mathsf{P_A}(2)$ and the maximum relevance score in $\mathsf{Q_A}(2)$, so (*,p1) is output.	
$\mathsf{P_A}(2)$	(m1,*):8, (m2,*):8
$\mathsf{Q_A}(2)$	(*,p1):8, (m1,p2):6, (m2,p2):6, (m1,p3):4, (m2,p3):4
$\mathsf{Q_C}(2)$	(*,p1):8, (m1,p3):4, (m2,p3):4

After two iterations, the top-1 cell, $(*, \mathrm{p1})$, is output. ■

IV. Experimental Study

We crawled laptop reviews from `Amazon.com`. Each review associated with 10 attributes (structural dimensions), *e.g.*, Brand, Disk, and CPU, forms a row in our text database $\mathcal{DB}$. We have 14K reviews, and each review has 73.4 keywords with stop words removed on average. The total length of documents in this text database is 1,022,462. Experiments were conducted on a PC with a 2.5 GHz Intel Core 2 Duo CPU and 3.0 GB RAM under XP. All algorithms were implemented in C++.

We test our two algorithms, **One-Scan** (Algorithm 1) and **SS-Ordering** (Algorithm 2), and report the time (milliseconds) and *running-time memory* (KB). Running-time memory refers to the memory consumed excluding inverted indexes and datasets. All the results reported are the averages for 20 typical keyword queries for laptops, each with 4 terms on average.

1) Number of Dimensions (Figure 1): We set $k = 80$, $\mathsf{minsup} = 1$. The database $\mathcal{DB}$ is projected into the first n dimensions, for $n = 2, 4, 6, 8, 10$.

SS-Ordering is consistently at least 50 times faster than One-Scan, and consumes much less memory (except when $n = 2$). When n increases (the number of non-empty cells could increases exponentially), both algorithms use more time and memory. SS-Ordering is less sensitive than One-Scan, because SS-Ordering only explores a small portion of the cells, while One-Scan always explores all the cells in the text cube.

2) Size of Database (Figure 2): $\mathcal{DB}_1, \ldots, \mathcal{DB}_5$ are generated from $\mathcal{DB}$ (by copying and randomly perturbing rows in $\mathcal{DB}$) with sizes varying from from 28K to 140K. Top-80 answers over them are output with $\mathsf{minsup} = 1$.

Time and memory consumed by both the two algorithms increase only linearly w.r.t. the size of the database. For SS-Ordering, this is because of our search space ordering strategy. Again, One-Scan always explores all the cells, so it is much slower and consumes more memory than SS-Ordering does.

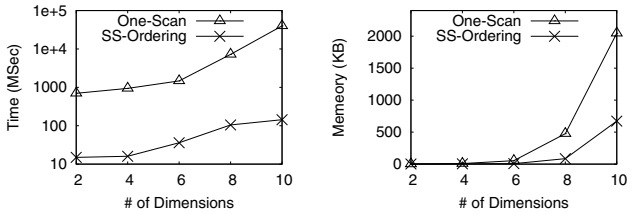

Fig. 1. Varying the Number of Dimensions

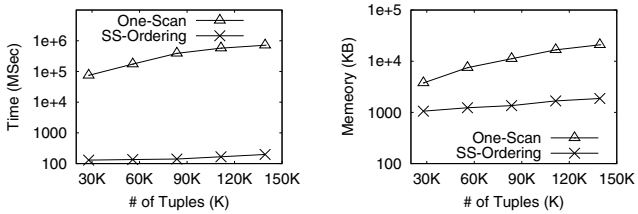

Fig. 2. Varying the Size of Database $|\mathcal{DB}|$

V. Related Work

Keyword Search in RDBMSs. Although based on different applications and motivations, keyword search in text cube is related to keyword search in RDB, which has attracted a lot of attention [9]. The first type uses SQL to find joins of tuples (relevance scoring function for "joins" are designed), *e.g.*, [2], [4], [6], [10], [11]. The second type materializes the whole RDB as a graph and proposes algorithms to enumerate (top-k) relevant subgraphs, *e.g.*, [3], [7], [5]. A more complete survey on this topic is deferred to the full version of this paper.

OLAP on Multidimensional Text Data. Inspired by data cube [12], the text cube model is firstly proposed in [1]. [1] focuses on how to partially materialize inverted indexes and term frequency vectors as measures in cells of text cube, and how to support OLAP query (not keyword query) efficiently. For keyword search, iceberg cube computation, *e.g.*, [13], cannot be directly applied because it is query-independent; neither does ranking cube [14], because it focuses on certain subspace.

Keyword-based Search and OLAP in Data Cube. [15] studies answering keyword queries on RDB using minimal group-bys, which is the work most relevant to ours. For a keyword query against a multidimensional text database, it aims to find the *minimal cells* containing all or some of the query terms in the aggregated text data (no descendant of this cell contains the same number of query terms). But, [15] does not score or rank the answers. So when the number of returned answers is large (e.g. ≥ 1000), it is difficult for users to browse all the answers.

Another relevant work is keyword-driven analytical processing (KDAP) [16]. Motivated by an application scenario different from [15] and our work, KDAP supports interactive data exploration using keywords. Candidate subspaces are output to disambiguate the keyword terms. [15] and our work focus on efficient answering of keyword queries, while efficiency is not a major concern in KDAP ([16] does not report any experiment on the efficiency of algorithms).

References

[1] C. X. Lin, B. Ding, J. Han, F. Zhu, and B. Zhao, "Text cube: Computing ir measures for multidimensional text database analysis," in *ICDM*, 2008.

[2] S. Agrawal, S. Chaudhuri, and G. Das, "Dbxplorer: A system for keyword-based search over relational databases," in *ICDE*, 2002.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword searching & browsing in dbs using banks," in *ICDE*, 2002.

[4] V. Hristidis, L. Gravano, and Y. Papakonstantinou, "Efficient ir-style keyword search over relational databases," in *VLDB*, 2003.

[5] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou, "Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data," in *SIGMOD*, 2008.

[6] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury, "Effective keyword search in relational databases," in *SIGMOD*, 2006.

[7] B. Kimelfeld and Y. Sagiv, "Finding and approximating top-k answers in keyword proximity search," in *PODS*, 2006.

[8] S. E. Robertson, S. Walker, and M. Hancock-Beaulieu, "Okapi at trec-7: Automatic ad hoc, filtering, vlc and interactive," in *TREC*, 1998.

[9] G. Weikum, "Db&ir: both sides now," in *SIGMOD*, 2007.

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

[11] H. He, H. Wang, J. Yang, and P. S. Yu, "Blinks: ranked keyword searches on graphs," in *SIGMOD Conference*, 2007, pp. 305–316.

[12] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total," in *ICDE*, 1996.

[13] K. S. Beyer and R. Ramakrishnan, "Bottom-up computation of sparse and iceberg cubes," in *SIGMOD*, 1999.

[14] D. Xin, J. Han, H. Cheng, and X. Li, "Answering top-k queries with multi-dim selections: The ranking cube approach," in *VLDB*, 2006.

[15] B. Zhou and J. Pei, "Answering aggregate keyword queries on relational databases using minimal group-bys," in *EDBT*, 2009.

[16] P. Wu, Y. Sismanis, and B. Reinwald, "Towards keyword-driven analytical processing," in *SIGMOD*, 2007.