Efficient Keyword-Based Search for Top-K Cells in Text Cube

Bolin Ding, Bo Zhao, Cindy Xide Lin, Jiawei Han, *Fellow*, *IEEE*, Chengxiang Zhai, Ashok Srivastava, *Senior Member*, *IEEE*, and Nikunj C. Oza, *Member*, *IEEE*

Abstract—Previous studies on supporting free-form keyword queries over RDBMSs provide users with linked structures (e.g., a set of joined tuples) that are relevant to a given keyword query. Most of them focus on ranking individual tuples from one table or joins of multiple tables containing a set of keywords. In this paper, we study the problem of keyword search in a data cube with text-rich dimension(s) (so-called *text cube*). The text cube is built on a multidimensional text database, where each row is associated with some text data (a document) and other structural dimensions (attributes). A cell in the text cube aggregates a set of documents with matching attribute values in a subset of dimensions. We define a keyword-based query language and an IR-style relevance model for scoring/ranking cells in the text cube. Given a keyword query, our goal is to find the top-k most relevant cells. We propose four approaches: *inverted-index one-scan*, *document sorted-scan*, *bottom-up dynamic programming*, and *search-space ordering*. The *search-space ordering* algorithm explores only a small portion of the text cube for finding the top-k answers, and enables early termination. Extensive experimental studies are conducted to verify the effectiveness and efficiency of the proposed approaches.

Index Terms—Keyword search, multidimensional text data, data cube.

1 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/tuples with matching attribute values in a subset of dimensions.

Keyword query, one of the most popular and easy-to-use ways to retrieve useful information from a collection of plain documents, is being extended to RDBMSs to retrieve information from text-rich attributes [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16]. Given a set of keywords, existing methods aim to find relevant tuples or joins of tuples (e.g., linked by foreign keys) that contain all or some of the keywords.

In this paper, we study supporting keyword-based search in text cube. Unlike in plain documents nor RDBMSs, the goal of keyword search in text cube is to find the top-k most *relevant cells* for a given keyword query.

- *B. Ding, B. Zhao, C.X. Lin, J. Han, and C. Zhai are with the Department of Computer Science, University of Illinois at Urbana-Champaign, 201 North Goodwin Avenue, Urbana, IL 61801. E-mail: {bding3, bozhao3, xidelin2, hanj, czhai}@uiuc.edu.*
- *A. Srivastava and N.C. Oza are with Intelligent Systems Division, NASA Ames Research Center, Moffett Field, CA 94035. E-mail: ashok@email.arc.nasa.gov, nikunj.c.oza@nasa.gov.*

Manuscript received 30 Apr. 2010; revised 3 Nov. 2010; accepted 20 Dec. 2010; published online 1 Feb. 2011.
Recommended for acceptance by S. Chaudhuri, Y. Chen, and J.X. Yu.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDESI-2010-04-0253.
Digital Object Identifier no. 10.1109/TKDE.2011.34.

Example 1.1 (Motivation). Table 1 shows a small sample database of customer reviews about laptops. *Customer Review* is the text attribute, whereas *Brand*, *Model*, *CPU*, and *OS* are structural attributes.

Jim, a marketing analyst, wants to find which laptops are commented as *lightweight* 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. As multidimensional attributes are associated with each review, 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? This is our intention to study such a new mechanism.

A cell in the text cube is in the form of $(\text{Brand} = \text{Acer}, \text{Model} = \text{AOA110}, \text{CPU} = *, \text{OS} = *)$, which aggregates all the customer reviews (the first two in Table 1) for Acer AOA110 laptops. Another cell $(\text{Brand} = *, \text{Model} = *, \text{CPU} = 1.8\,\text{GHz}, \text{OS} = \text{XP})$ aggregates the second two reviews. It can be seen that the first cell is more relevant to Jim's query (Acer AOA110 laptops are lightweight and powerful) than the second. The goal of our system is to provide Jim with such aggregated information (i.e., the cell "Acer AOA110 laptops" that is relevant to his query), instead of a ranked list of individual customer reviews.

If Jim is interested in gaming computers, he can type {"fast," "gaming"} into the system, and high-performance computer models with good customer reviewers are expected to be output, e.g., $(\text{Brand} = *, \text{Model} = \text{XtremeXT}, \text{CPU} = \text{IntelCorei7}, \text{OS} = *)$.

TABLE 1
Motivation Example

Brand	*Model*	*CPU*	*OS*	*Customer Review*
Acer	AOA110	1.6GHz	Linux	**light weight** as little as 2.2 lb, fun and **powerful** computing features
Acer	AOA110	1.8GHz	XP	**weight** just over two pounds, with **powerful** Intel Atom Processor
Asus	EEE PC	1.8GHz	XP	comes in pearl white style, images are sharp, disk is large

Traditional IR techniques can be used to rank documents according to the relevance. In a large text database, however, the number of relevant documents to a query could be large, and a user has to spend much time reading them.

If a document is associated with attribute information, in a data cube model (a multidimensional space induced by the attributes), e.g., the text cube, a *cell* aggregates the documents with matching values in a subset of attributes. Such a collection of documents is associated with each cell, corresponding to an object, e.g., "Acer AOA110 laptops" or "1.8 GHz XP laptops" (in the above example), that can be directly recommended to the user for the given query.

This paper studies the problem of *keyword-based top-k search in text cube*, i.e., *given a keyword query, find the top-k most relevant cells in a text cube*. When users want to retrieve information from a text cube using keyword queries, we believe that relevant cells, rather than relevant documents, are preferred as the answers, because: 1) relevant cells are easy for users to browse; and 2) relevant cells provide users insights about the relationship between the values of relational attributes and the text data.

1.1 Overview of Model and Techniques

Following is an overview of our work, with important issues highlighted.

1.1.1 Relevance Score

Given a keyword query, the first question is how to compute the "*relevance*" of a cell in a text cube for ranking. Note that a cell corresponds to a collection of documents. We employ the *average model* studied in [17] to assign the relevance scores of the cells. Specifically, *ANY* IR scoring function (e.g., Okapi) can be used to compute the relevance score of each single document, and the *relevance score of a cell* (a document collection) is the (weighted) average of relevance scores of documents in this cell. An alternative is the *big document model*: documents in a cell are concatenated into a *big document*, and the relevance of the cell is the relevance of this big document. Elsas et al. [17] show that the average model is better than the big document model.

1.1.2 Support Requirement

We allow a user to specify a *support threshold* minsup, and we only output the relevant cells with no less than minsup documents. Users may not prefer a cell with too few documents, since this cell is not "popular" (e.g., if there are too few reviews on a computer model, the user is unlikely to choose this one); users may feel more confident browsing a cell with larger number of documents (reviews). Also, when high-level information is preferred, minsup can be set to a relatively large value.

1.1.3 Challenges

The major computational challenge of this problem is the huge number of cells in a text cube. It increases exponentially w.r.t. the dimensionality and is much larger than the number of documents in the database.

1.1.4 Efficient Algorithms

We design four approaches for the problem of *keyword-based top-k search in text cube*. The first one (*inverted-index one-scan*) scans the inverted index of each keyword in the query once to compute the relevance scores of all cells in the text cube. The second one (*document sorted-scan*) scans the documents in the order of relevance, and at the same time, updates the relevance scores of cells containing each document being visited; early stop for top-k is possible because upper bounds of relevance scores can be estimated. The third one (*bottom-up dynamic programming*) computes relevance scores of cells from bottom to top in a level-by-level manner. The fourth one (*search-space ordering*), based on the third one, explores cells in the order of relevance and prunes the search space by estimating relevance score upper bound in subspaces, so as to explore as few cells in the text cube as possible before outputting the top-k answers.

1.2 Contributions

In this paper, we propose and study the problem of *keyword-based top-k search in text cube* (or multidimensional text data): *finding the top-k cells relevant to a user-given keyword query*. Flexible keyword-based query language and relevance scoring formula of cells (aggregation of text data) are developed. We analyze the computational challenges and propose four approaches, *inverted-index one-scan*, *document sorted-scan*, *bottom-up dynamic programming*, and *search-space ordering*, to support the query language in text cube. We compare the four approaches and study their efficiency and effectiveness experimentally using both real and synthetic data sets.

1.3 Organization

Section 2 introduces the text cube model of multidimensional text data, defines the keyword search problem and keyword-based query language in text cube, and analyzes computational challenges. The four approaches for finding the top-k most relevant cells for a given keyword query are introduced in Sections 3 and 4. Experimental study is reported in Section 5, followed by discussion on how to extend our approach in Section 6 and related work in Section 7. Section 8 concludes this paper.

2 Keyword Queries in Text Cube

In this section, we introduce our data cube model and the keyword search problem. Preprocessing for our system and challenges for query processing are also discussed.

2.1 A Data Cube Model for Text Data

A set of documents $\mathcal{D}$ 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 $r = (a_1, a_2, \ldots, a_n, \mathrm{d})$: let $r[\mathcal{A}_i] = a_i \in \mathcal{A}_i$ be the *value* of

TABLE 2
A Four-Dimensional Text Database $\mathcal{DB}$

Dimensions				Text Data
M (Model)	P (Price)	T (Time)	S (Score)	d (Document)
m1	p1	t1	s1	d_1 = { w1, w1, w2, w2, w8}
m1	p2	t1	s2	d_2 = { w2, w2, w5, w6, w7}
m1	p3	t2	s2	d_3 = { w3, w4, w3, w6, w7}
m2	p1	t2	s2	d_4 = { w1, w1, w1, w1, w4}
m2	p2	t2	s2	d_5 = { w5, w6, w7, w8, w8}
m2	p3	t1	s1	d_6 = { w4, w5, w8, w8, w9}

attribute (or *dimension*) $\mathcal{A}_i$, and $r[\mathcal{D}] = \mathrm{d}$ be the *document* in this row.

A document d is a multi(sub)set of the *term set* $\mathcal{W} = \{w_1, \ldots, w_M\}$: a term w_i may appear multiple times in d.

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

2.1.1 Cell and Aggregated Document

In the text cube built on $\mathcal{DB}$, a *cell* is in the form of $C = (v_1, v_2, \ldots, v_n : \mathrm{D})$, 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, for a cell $C = (v_1, v_2, \ldots, v_n : \mathrm{D})$,

$$\mathrm{D} = \{r[\mathcal{D}] | \text{ for } r \in \mathcal{DB} \text{ s.t. } r[\mathcal{A}_i] = v_i \text{ if } v_i \neq *\}.$$

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

All the cells with the same set of non-* dimensions form a *cuboid*. A cuboid with m non-* dimensions is an *m-dim cuboid*. The n-dim cuboid (with no aggregated dimension) is called the *base cuboid*. Cells in an m-dim cuboid are *m-dim cells*, and cells in a base cuboid are *base cells*. Note that each base cell may aggregate multiple rows/documents, since different rows may have the same values of attributes.

2.1.2 Ancestor and Descendant

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

2.1.3 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 1) $C' \in \mathrm{ans}(C)$, and 2) C' is an i-dim cell while C is an $(i+1)$-dim cell. Let $\mathrm{par}(C)$ ($\mathrm{chd}(C')$) denote the set of parents of a cell C (children of a cell C'). Note that the 0-dim cell (i.e., the cell in the form of $(*, *, \ldots, * : \mathrm{D})$) has no parent.

2.1.4 $\mathcal{A}$-Parent and $\mathcal{A}$-Child

The *$\mathcal{A}$-parent* of a cell C is C' (or, C is an *$\mathcal{A}$-child* of C'), denoted by $\mathrm{par}_{\mathcal{A}}(C) = C'$, iff 1) C' is a parent of C, 2) $C[\mathcal{A}] \neq *$, and 3) $C'[\mathcal{A}] = *$. A cell may have more than one *$\mathcal{A}$-child*, and we use $\mathrm{chd}_{\mathcal{A}}(C')$ to denote the set of all *$\mathcal{A}$-children* of C'.

It is well known all the cells in a data cube (or text cube) form a lattice, according to the parent-child relationship.

TABLE 3
Some Cells in the Text Cube

Cell	M	P	T	S	D
C_0	*	*	*	*	$\{d_1, d_2, d_3, d_4, d_5, d_6\}$
C_1	m1	*	*	*	$\{d_1, d_2, d_3\}$
C_2	m2	*	*	*	$\{d_4, d_5, d_6\}$
C_3	m1	*	t1	*	$\{d_1, d_2\}$
C_4	m1	*	t2	*	$\{d_3\}$
C_5	m1	*	*	s1	$\{d_1\}$
C_6	m1	*	*	s2	$\{d_2, d_3\}$
C_7	m2	*	t1	*	$\{d_6\}$
C_8	m2	*	t2	*	$\{d_4, d_5\}$
C_9	m2	*	t2	s2	$\{d_4, d_5\}$
C_{10}	m1	*	t1	s1	$\{d_1\}$
C_{11}	m1	p1	t1	s1	$\{d_1\}$
C_{12}	m1	p2	t1	s2	$\{d_2\}$
C_{13}	*	p2	*	s1	$\emptyset$ (empty cell)
C_{14}	m1	*	t2	s1	$\emptyset$ (empty cell)

Example 2.1 (Text Cube). Table 2 shows a text database $\mathcal{DB}$, with four dimensions, M, P, T, and S. Term set $\mathcal{W} = \{\mathrm{w1}, \mathrm{w2}, \ldots, \mathrm{w8}\}$. A total of six documents are stored.

Table 3 shows some cells in the text cube generated from $\mathcal{DB}$. C_0 has support 6 and C_1 has support 3, i.e., $|C_0| = 6$ and $|C_1| = 3$. C_0 is the 0-dim cell and C_{11}, C_{12} are base cells. C_3, C_4, C_5, C_6, C_{10}, C_{11}, and C_{12} are descendants of C_1, while only C_3, C_4, C_5, and C_6 are children of C_1. Note C_1 has some other descendants and children not listed in this table. A cell may have more than one parent, e.g., both C_3 and C_5 are parents of C_{10}.

Cell C_1 is the T-parent of C_3 and C_4, i.e., $\mathrm{par}_{\mathrm{T}}(C_3) = \mathrm{par}_{\mathrm{T}}(C_4) = C_1$. Cells C_3 and C_4 are the T-children of C_1, i.e., $\mathrm{chd}_{\mathrm{T}}(C_1) = \{C_3, C_4\}$. Cells C_5 and C_6 are the S-children of C_1, i.e., $\mathrm{chd}_{\mathrm{S}}(C_1) = \{C_5, C_6\}$. And similarly, $\mathrm{par}_{\mathrm{S}}(C_5) = \mathrm{par}_{\mathrm{S}}(C_6) = C_1$.

2.1.5 Multiple Documents in a Row

In a database where each row contains more than one document, to generalize our model, we can create a linked list of documents and keep the total number of documents in each row. By aggregating this count into cells on higher levels, we can easily calculate the support of each cell. Note that this modification will not affect our algorithms introduced later.

2.2 Keyword Search Problem in Text Cube

In traditional data cubes, operations like *drill-down* and *roll-up* suffice for users to explore multidimensional data. In text cube, however, a large portion of data is text. Since keyword query is an effective way for users to explore text data, we propose the keyword search problem in text cube.

2.2.1 Keyword Search Problem

A *keyword query* is a set of terms, i.e., $\mathrm{q} = \{t_1, t_2, \ldots, t_{|\mathrm{q}|}\} \subseteq \mathcal{W}$. Given a keyword query q and a minimum support requirement minsup, the goal is to *find the top-k cells C's s.t. supports $|C| \geq \mathrm{minsup}$ with the top-k highest relevance scores in the text cube of $\mathcal{DB}$.*

Note that a cell relevant to the query q may contain all or some of the terms $t_1, \ldots, t_{|\mathrm{q}|}$. The *relevance score* of a cell C w.r.t. the query q is defined as a function $\mathrm{rel}(\mathrm{q}, C[\mathcal{D}])$ of the aggregated document $C[\mathcal{D}]$ and the query q. For brevity, it is also written as $\mathrm{rel}(\mathrm{q}, C)$. Because the total number of cells in the text cube could be huge and it is not possible for a user to browse all of them, we return the top-k cells in the nonincreasing order of relevance scores, where k can be specified by a user.

Recall the support of a cell C is the number of documents aggregated in $C[\mathcal{D}]$. We allow users to specify the minimum support minsup, because they may be only interested in cells "popular" enough. Users feel more confident when browsing a cell containing a large number of documents (e.g., customer reviews) than a cell with only one or two.

2.2.2 Relevance Scoring Formula

To rank all the cells and find the top-k ones, we define the relevance scoring function $\mathrm{rel}(\mathrm{q}, C[\mathcal{D}])$ (or $\mathrm{rel}(\mathrm{q}, C)$ for brevity). Recall an aggregated document $C[\mathcal{D}] \subseteq \mathcal{D}$ is a set of documents aggregated in C. Here, we treat every document in $C[\mathcal{D}]$ equally when calculating relevance score of C w.r.t. a keyword query q. $\mathrm{rel}(\mathrm{q}, C[\mathcal{D}])$ is defined as the average of all the relevance scores of documents in $C[\mathcal{D}]$:

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

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

$$\begin{aligned} &s(\mathrm{q}, \mathrm{d}) = \\ &\sum_{t \in \mathrm{q}} \ln \left(\frac{N - \mathrm{df}_t + 0.5}{\mathrm{df}_t + 0.5} \frac{(k_1+1)\mathrm{tf}_{t,\mathrm{d}}}{k_1((1-b) + b\frac{\mathrm{dl}_\mathrm{d}}{\mathrm{avdl}}) + \mathrm{tf}_{t,\mathrm{d}}} \frac{(k_3+1)\mathrm{qtf}_{t,\mathrm{q}}}{k_3 + \mathrm{qtf}_{t,\mathrm{q}}} \right) \\ &\quad \text{(Okapi weighting [18])} \\ &\quad = \sum_{t \in \mathrm{q}} w_{\mathrm{idf}}(t) \cdot w_{\mathrm{tf}}(t, \mathrm{d}) \cdot w_{\mathrm{qtf}}(t, \mathrm{q}), \end{aligned} \tag{2}$$

where $N = |\mathcal{DB}|$, $\mathrm{tf}_{t,\mathrm{d}}$ is the *term frequency* of term $t \in \mathrm{q}$ 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 $\mathcal{DB}$, $\mathrm{qtf}_{t,\mathrm{q}}$ is the number of times t appearing in q, and, k_1, b, k_3 are constants.

Our algorithms introduced later can also handle other formulas $s(\mathrm{q}, \mathrm{d})$ for scoring documents, say pivoted normalization weighting [19]. In general, our algorithms allow $s(\mathrm{q}, \mathrm{d})$ to be in the form of $s(\mathrm{q}, \mathrm{d}) = s(\langle \mathrm{tf}_{t,\mathrm{d}} \rangle, \Delta_\mathrm{q}, \Delta_\mathrm{d})$, where Δ_q is the set of parameters about q, e.g., inverted document frequency (idf), and Δ_d is the set of parameters about d, e.g., document length (for normalization). This form covers most IR models for evaluating the relevance of a document w.r.t. a query.

Elsas et al. [17] show that the weighted version of (1) (document with higher weight in $C[\mathcal{D}]$ contributes more to the relevance score $\mathrm{rel}(\mathrm{q}, C)$) is effective for ranking document corpuses. In this paper, we use the unweighted version (i.e., (1)) for the ease of explanation. Our algorithms can be easily extended to the weighted version, if the weights of documents can be precomputed.

Note that, for query q, $\mathrm{rel}(\mathrm{q}, C)$ is the relevance score of a set (corpus) of documents (i.e., $C[\mathcal{D}]$), while $s(\mathrm{q}, \mathrm{d})$ is the relevance score of one document. In the rest part of this paper, we call $\mathrm{rel}(\mathrm{q}, C)$ a *relevance scoring formula*, and $s(\mathrm{q}, \mathrm{d})$ a *document scoring formula* to distinguish them.

2.2.3 Extended Form of Keyword Query

Users may want to retrieve answers from a certain part of the text cube, by specifying a subset dimensions of interests and/or values of some dimensions. Motivated by this, the simplest form of keyword queries q can be extended by adding *dimension-value constraints*.

In an n-dimensional text cube, *an extended keyword query* is in the form of $Q = (u_1, u_2, \ldots, u_n : \mathrm{q})$, where $u_i \in \mathcal{A}_i \cup \{*, ?\}$. We also use $Q[\mathcal{A}_i]$ to denote u_i. $Q[\mathcal{A}_i] \in \mathcal{A}_i$ specifies the value of dimension $\mathcal{A}_i$ in a cell C; $Q[\mathcal{A}_i] = *$ means the dimension $\mathcal{A}_i$ in a cell C must be aggregated; and $Q[\mathcal{A}_i] = ?$ (question mark) imposes no constraint on the dimension $\mathcal{A}_i$ of a cell C. A cell C is said to be *feasible* w.r.t. the query Q iff

1. for dimension $\mathcal{A}_i$ s.t. $Q[\mathcal{A}_i] = *$, $C[\mathcal{A}_i] = *$ ($\mathcal{A}_i$ is aggregated in C);
2. for $\mathcal{A}_i$ s.t. $Q[\mathcal{A}_i] \in \mathcal{A}_i$, $C[\mathcal{A}_i] = Q[\mathcal{A}_i]$; and
3. for $\mathcal{A}_i$ s.t. $Q[\mathcal{A}_i] = ?$, no constraint on $C[\mathcal{A}_i]$.

Given an *extended keyword query* $Q = (u_1, u_2, \ldots, u_n : \mathrm{q})$ and a minimum support minsup, our goal is to find the top-k *feasible* cells C's s.t. support $|C| \geq \mathrm{minsup}$ and *relevance scores* $\mathrm{rel}(\mathrm{q}, C[\mathcal{D}])$'s are the top-$k$ highest.

Example 2.2 (Queries, Answers, and Relevance Scores). In the text cube of $\mathcal{DB}$ in Table 2, consider a keyword query $\mathrm{q} = \{\mathrm{w1}, \mathrm{w2}\}$ and $\mathrm{minsup} = 2$.

Using Okapi (2) with $k_1 = k_3 = 1$ and $b = 0.5$: $s(\mathrm{q}, \mathrm{d}_1) = 1.6$, $s(\mathrm{q}, \mathrm{d}_2) = 0.8$, $s(\mathrm{q}, \mathrm{d}_4) = 0.9$. Base cells $(\mathrm{m1}, \mathrm{p1}, \mathrm{t1}, \mathrm{s1}, \{\mathrm{d}_1\})$, $(\mathrm{m1}, \mathrm{p2}, \mathrm{t1}, \mathrm{s2}, \{\mathrm{d}_2\})$, and $(\mathrm{m2}, \mathrm{p1}, \mathrm{t2}, \mathrm{s2}, \{\mathrm{d}_4\})$ have relevance scores 1.6, 0.8, and 0.9, respectively. But they will not be output because their supports are all 1, less than the threshold minsup. Cell $(*, \mathrm{p1}, *, *, \{\mathrm{d}_1, \mathrm{d}_4\})$ has relevance score $(1.6 + 0.9)/2 = 1.25$, and $(\mathrm{m1}, *, \mathrm{t1}, *, \{\mathrm{d}_1, \mathrm{d}_2\})$ has relevance score $(1.6 + 0.8)/2 = 1.2$. They are the top-2 cells in the output. So we may infer, when the price $\mathrm{P} = \mathrm{p1}$, documents in the text cube are highly relevant to the given query q.

An extended keyword query $Q = (?, *, ?, * : \mathrm{q})$ (i.e., no constraint on dimensions M and T, while dimensions P and S must be aggregated). C_0-C_4 in Table 3 are feasible cells, but C_5 and C_6 are not, since $C_5[\mathrm{S}] \neq *$ and $C_6[\mathrm{S}] \neq *$.

Another query is $Q' = (?, *, ?, \mathrm{s2} : \mathrm{q})$, which imposes an additional constraint that a cell must value s2 on dimension S, i.e., $C[\mathrm{S}] = \mathrm{s2}$. Then, C_9 in Table 3 is a feasible cell ($C_9[\mathrm{S}] = \mathrm{s2}$); but C_{10} is not, for $C_{10}[\mathrm{S}] = \mathrm{s1} \neq \mathrm{s2}$.

2.3 Preprocessing

In the algorithms introduced later, some parameters are assumed to be precomputed to speed up the query processing.

First, the support of each cell, i.e., the number of documents aggregated in this cell, is precomputed and stored, since supports are query independent. This only requires additional $O(1)$ space for each cell. Later, we can use this to efficiently check whether the support of a cell is above the threshold, and whether a cell is *fully aggregated*.

Second, note that (2) can be rewritten as a more general form: $s(\mathrm{q}, \mathrm{d}) = \sum_{t \in \mathrm{q}} w_{\mathrm{idf}}(t) \cdot w_{\mathrm{tf}}(t, \mathrm{d}) \cdot w_{\mathrm{qtf}}(t, \mathrm{q})$. Both $w_{\mathrm{idf}}(t)$ and $w_{\mathrm{tf}}(t, \mathrm{d})$ are precomputed for all terms and documents,

and stored in inverted indexes. So, as an online query q comes, $s(\mathrm{q}, \mathrm{d})$ can be computed conveniently.

Note that the above two steps can be done at the same time, as documents are scanned one by one. All algorithms presented later can benefit from this precomputation.

2.4 Challenges of Query Processing

There are two challenges of this keyword search problem: first, the size of a text cube could be huge; and second, it is impractical to precompute relevance scores for all queries.

2.4.1 Size of Text Cube

There is an n-dimensional database $\mathcal{DB} = (\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n, \mathcal{D})$ with N rows, s.t. the nonempty cells in the text cube of $\mathcal{DB}$ is $\Omega(N \cdot 2^n)$ or $\Omega(\prod_{i=1}^{n}(|\mathcal{A}_i| + 1))$, where $|\mathcal{A}_i|$ is the number of different values in dimension $\mathcal{A}_i$.

Even when minsup (>0) is nontrivial, the number of cells to be considered (those with support $\geq \mathrm{minsup}$) is still huge, since a data cube is "fat" in the middle. For example, suppose minsup is large enough s.t. only d-dim cells with $d \leq n/2$ have support $\geq \mathrm{minsup}$, we may still need to consider at least $N \cdot \binom{n}{n/2} = \Omega(N \cdot 2^{n/2})$ cells to select the top-k relevant ones w.r.t. a keyword query.

So *only when* the number of dimensions is small (2 to 4), we can compute the relevance scores of all cells and then sort them to find the top-k cells efficiently.

2.4.2 Limits of Preprocessing

In text cube, relevance scores $\mathrm{rel}(\mathrm{q}, C)$ of cells have to be computed online because they are query dependent, although $w_{\mathrm{idf}}(t)$ and $w_{\mathrm{tf}}(t, \mathrm{d})$ can be precomputed (as in Section 2.3). Also, the ranking of cells according to $\mathrm{rel}(\mathrm{q}, C)$ cannot be precomputed either, since for different queries, the rankings could be different. Online computation of $\mathrm{rel}(\mathrm{q}, C)$ and ranking of cells dominate the cost of processing a query; so, we aim to explore only a small portion of the text cube for the top-k answers.

3 Document Linear-Scan Approaches

Two approaches for the keyword-based top-k search problem based on linear scans of rows are introduced in Sections 3.1 and 3.2. The first one scans all rows to compute the relevance scores of all cells, and then output the top-k cells. The second one scans the rows in the descending order of relevance scores; in this process, cells containing each document are updated with upper bounds of relevance scores estimated to enable the early stop for top-k. We first focus on simple keyword query $\mathrm{q} = \{t_1, \ldots, t_{|\mathrm{q}|}\}$ with support requirement minsup and show how to handle extended keyword queries in Section 3.3.

3.1 Inverted-Index One-Scan Approach

We construct a *row-based inverted index* for each term. Recall $\mathrm{tf}_{t,\mathrm{d}}$ denotes the term frequency of t in the document d. From the *row-based inverted index* $\mathrm{IV}(t)$ of term t, we can efficiently retrieve all the rows $r = (a_1, \ldots, a_n, \mathrm{d})$'s in $\mathcal{DB}$ with term frequencies $\mathrm{tf}_{t,\mathrm{d}} > 0$. The size of row-based inverted indices for all terms in $\mathcal{W}$ is bounded by the total length of all documents in $\mathcal{DB}$.

Given a query q against the text cube of $\mathcal{DB}$ and minimum support minsup, the *inverted-index one-scan* approach is outlined in Algorithm 1. We first compute the score $s(\mathrm{q}, \mathrm{d})$ of each document in a row that contains at least one term in q (since otherwise $s(\mathrm{q}, \mathrm{d})$ is 0) using the inverted indexes (line 1). Then, for each row r, we can efficiently identify all the cells C's in the text cube s.t. $r[\mathcal{D}] \in C[\mathcal{D}]$ and support $|C[\mathcal{D}]| \geq \mathrm{minsup}$ (lines 2-3: note $|C[\mathcal{D}]|$ can be precomputed and materialized). These cells are the ancestors of the base cell corresponding to row r. We use $\mathrm{ans}(r)$ to denote this set of cells, and note that $|\mathrm{ans}(r)| \leq 2^n$ in an n-dimensional text cube. We update $\mathrm{rel}(\mathrm{q}, C)$ using $s(\mathrm{q}, r[\mathcal{D}])$ (line 4). Finally, after all rows are scanned, each $\mathrm{rel}(\mathrm{q}, C)$ is correctly computed as the average of $s(\mathrm{q}, r[\mathcal{D}])$'s with r contained in C (some rows with $s(\mathrm{q}, r[\mathcal{D}] = 0$ are not scanned, but we know the total number of documents in C, so the average still can be computed), and the top-k ones are output (line 5).

Algorithm 1. Inverted-Index One-Scan Algorithm
Input: keyword query q against $\mathcal{DB}$, parameter k, minsup
1: Compute relevance score $s(\mathrm{q}, r[\mathcal{D}])$ for each row r;
2: **for each** row $r \in \mathcal{DB}$ with $s(\mathrm{q}, r[\mathcal{D}]) > 0$ **do**
3: **for each** cell $C \in \mathrm{ans}(r)$ and $|C[\mathcal{D}]| \geq \mathrm{minsup}$ **do**
4: Update $\mathrm{rel}(\mathrm{q}, C)$ using $s(\mathrm{q}, r[\mathcal{D}])$;
5: Output the top-k relevant cells with support $\geq \mathrm{minsup}$ (for cells with the same relevance scores, output them in the descending order of support).

Example 3.1. Consider a row $r = (\mathrm{m1}, \mathrm{p1}, \mathrm{t1}, \mathrm{s1}, \mathrm{d_1})$ in the text database in Table 2a. A cell C with $\mathrm{d_1} \in C[\mathcal{D}]$ must be one of the 2^4 ancestors of the base cell $(\mathrm{m1}, \mathrm{p1}, \mathrm{t1}, \mathrm{s1})$, e.g., $(\mathrm{m1}, \mathrm{p1}, *, \mathrm{s1})$, $(\mathrm{m1}, *, \mathrm{t1}, *)$, and $(*, *, \mathrm{t1}, *)$, which are considered in lines 3-4.

Theorem 1. *Algorithm 1 uses $O(N \cdot |\mathrm{q}| + 2^n \cdot N + k \cdot \log M)$ time and $O(n \cdot M)$ space, where $N = |\mathcal{DB}|$ is the total number of rows, n is the dimensionality, and M is the total number of nonempty cells.*

Proof. Algorithm 1 scans the row-based inverted index of each query term only once. The size of each row-based inverted index is bounded by $N = |\mathcal{DB}|$. Also, the 2^n ancestors of a row r can be enumerated in linear time. So, Algorithm 1 requires $O(N \cdot |\mathrm{q}| + 2^n \cdot N)$ time to compute the relevance scores of all nonempty cells. After that, it constructs a (Fibonacci) heap of all nonempty cells (ordered by relevance) in linear time $O(M) \leq O(2^n \cdot N)$ [20] and outputs the top-k in $O(k \cdot \log M)$ time. Totally, we need $O(N \cdot |\mathrm{q}| + 2^n \cdot N + k \cdot \log M)$ time.

Since there are at most $M \leq 2^n \cdot N$ nonempty cells, and for each cell we store its dimension values, relevance score, and support, we need totally $O(n \cdot M)$ space (besides the storage of inverted indexes and text cube). □

3.2 Document Sorted-Scan Approach

Our second approach is outlined in Algorithm 2. Compared to Algorithm 1, it uses an additional data structure, a priority queue Q, to keep cells (candidates) in the descending order of relevance. Given a query q, initially, Q is empty. All rows in $\mathcal{DB}$ are scanned in the descending order of relevance (line 1): $r_1, r_2, \ldots, r_N$ such that $s(\mathrm{q}, r_i[\mathcal{D}]) \geq s(\mathrm{q}, r_{i+1}[\mathcal{D}])$. To obtain this ranked list, we simply compute scores of all rows first and then construct a heap to keep them in the descending order of scores; a row is popped if it

needs to be scanned.[1] Recall $\mathrm{ans}(r_i)$ is the set of ancestors of r_i, i.e., the set of cells containing document $r[\mathcal{D}]$. When r_i is scanned, for each $C \in \mathrm{ans}(r_i)$, we compute $\mathrm{rel}(\mathrm{q}, C)_i$ (line 4) as the average relevance score of documents among $r_1[\mathcal{D}], \ldots, r_i[\mathcal{D}]$ that are contained in C (recall that $\mathrm{rel}(\mathrm{q}, C)$ is the average relevance score of all documents that are contained in C). If C contains NO document in $\{r_{i+1}, \ldots, r_N\}$, we can conclude $\mathrm{rel}(\mathrm{q}, C) = \mathrm{rel}(\mathrm{q}, C)_i$ and insert C into the priority queue Q (lines 5-6). For a cell containing some rows that are already scanned, we have $\mathrm{rel}(\mathrm{q}, C) \leq \mathrm{rel}(\mathrm{q}, C)_i$, as rows are scanned in the descending order of relevance; for a cell C not touched yet (containing no row that is scanned), $\mathrm{rel}(\mathrm{q}, C)$ is bounded by $s(\mathrm{q}, r_{i+1}[\mathcal{D}])$, the maximum relevance score among the rest individual documents. So, for the cell with max relevance score in Q, $C' = \mathbf{top}(\mathrm{Q})$, once $\mathrm{rel}(\mathrm{q}, C') \geq \mathrm{rel}(\mathrm{q}, C)_i$ for all cells C (line 7(i)), and $\mathrm{rel}(\mathrm{q}, C') \geq s(\mathrm{q}, r_{i+1}[\mathcal{D}])$ (line 7(ii)), C' can be output and popped from Q (line 8).

Algorithm 2. Document Sorted-Scan Algorithm
Q: a priority queue of cells (sorted in the non-increasing order of relevance scores)
$\mathbf{top}(\mathrm{Q})$: the top (first) cell of a priority queue Q
Input: keyword query q against $\mathcal{DB}$, parameter k, minsup

1: Rows are scanned in the descending order of relevance: $r_1, r_2, \ldots, r_N$ such that $s(\mathrm{q}, r_i[\mathcal{D}]) \geq s(\mathrm{q}, r_{i+1}[\mathcal{D}])$;
2: **for** $i = 1$ to N **do**
3: **for each** cell $C \in \mathrm{ans}(r_i)$ and $|C[\mathcal{D}]| \geq \mathrm{minsup}$ **do**
4: Aggregate r_i into C: compute $\mathrm{rel}(\mathrm{q}, C)_i$ as the average of relevance scores of documents in both $r_1, \ldots, r_i$ and C;
5: **if** C contains NO doc in $\{r_{i+1}, \ldots, r_N\}$ **then**
6: Insert C into Q ($\mathrm{rel}(\mathrm{q}, C) = \mathrm{rel}(\mathrm{q}, C)_i$);
(for other cells, we have $\mathrm{rel}(\mathrm{q}, C)_i = \mathrm{rel}(\mathrm{q}, C)_{i-1}$)
7: **while** (i) $\mathrm{rel}(\mathrm{q}, \mathbf{top}(\mathrm{Q})) \geq \max_C\{\mathrm{rel}(\mathrm{q}, C)_i\}$ and (ii) $\mathrm{rel}(\mathrm{q}, \mathbf{top}(\mathrm{Q})) \geq s(\mathrm{q}, r_{i+1}[\mathcal{D}])$ **do**
8: Output $\mathbf{top}(\mathrm{Q})$, and delete it from Q;
9: $cnt \leftarrow cnt + 1$; **if** $cnt = k$ **then** terminate.

Theorem 2. *Algorithm 2 outputs the top-k cells with the highest relevance scores and support* $\geq \mathrm{minsup}$ *in the nonincreasing order of* $\mathrm{rel}(\mathrm{q}, C)$.

Proof. We only need to prove that, in any iteration of lines 3-9, if $C' = \mathbf{top}(\mathrm{Q})$ is output, then $\mathrm{rel}(\mathrm{q}, C')$ is the maximum among all cells that are not output yet.

For any cell $C \in \mathrm{Q}$, by the definition of Q, $\mathrm{rel}(\mathrm{q}, C') \geq \mathrm{rel}(\mathrm{q}, C)$. For any other cell C, we may have either

$$\mathrm{rel}(\mathrm{q}, C) \leq \mathrm{rel}(\mathrm{q}, C)_i \text{ or } \mathrm{rel}(\mathrm{q}, C) \leq s(\mathrm{q}, r_{i+1}[\mathcal{D}]).$$

So from line 7(i)-(ii), we have $\mathrm{rel}(\mathrm{q}, C') \geq \mathrm{rel}(\mathrm{q}, C)$. □

3.2.1 Implementation Issues

On line 4, $\mathrm{rel}(\mathrm{q}, C)_i$ can be efficiently computed from $\mathrm{rel}(\mathrm{q}, C)_j$ (r_j is the last row aggregated into C). On line 5, if the number of documents in a cell C is precomputed, we can test whether C contains NO document in $\{r_{i+1}, \ldots, r_N\}$ by counting the number of documents that are already aggregated into C. On line 7, $\max_C\{\mathrm{rel}(\mathrm{q}, C)_i\}$ can be efficiently fetched if $\mathrm{rel}(\mathrm{q}, C)_i$'s are maintained in another priority queue.

3.2.2 Tie Breaker

Ties (for cells with the same relevance scores) break arbitrarily in Algorithm 2. If for cells with the same relevance scores, we want to output them in the descending order of support, then we modify Algorithm 2 as follows: first, the priority queue Q keeps cells (candidates) in the descending order of relevance, and keeps cells with the same relevance scores in the descending order of support. Second, "$\geq$" should be changed into ">" on line 7, so that, when a cell is about to be output, all the cells (not output yet) with the same relevance scores are in Q.

Theorem 3. *Algorithm 2 uses* $O(N \cdot (|\mathrm{q}| + \log N) + 2^n \cdot N \cdot \log M)$ *time and* $O(n \cdot M)$ *space, where* $N = |\mathcal{DB}|$ *is the total number of rows,* n *is the dimensionality, and* M *is the total number of nonempty cells.*

Proof. In the worst case, it needs to scan all rows with relevance score computed, in the descending order of relevance, which takes $O(N \cdot (|\mathrm{q}| + \log N))$ time. Lines 4-6 repeat at most $O(2^n \cdot N)$ times, and each iteration takes $O(\log M)$ time since we need to main a priority queue Q and another queue to maintain $\max_C\{\mathrm{rel}(\mathrm{q}, C)_i\}$. Lines 7-9 also repeat at most $O(M) \leq O(2^n \cdot N)$ times and each iteration takes $O(\log M)$ time. So, the time complexity follows. The space complexity is the same as Algorithm 1, as additional priority queues need linear space. □

The complexity of Algorithm 2 is worse than Algorithm 1, but may terminate earlier before scanning all rows.

3.3 Handling Extended Keyword Queries

Algorithms 1 and 2 can be easily adapted to handle extended keyword query $Q = (u_1, \ldots, u_n : \mathrm{q})$. With the dimension-value constraints $u_1, \ldots, u_n$, we actually restrict our attention to a subspace of the text cube. Lines 2-3 of Algorithm 1 are modified a bit: we update $\mathrm{rel}(\mathrm{q}, C)$ only for the *feasible* cells in $\mathrm{ans}(r)$. For example, if $Q = (?, ?, \mathrm{t1}, * : \mathrm{q})$ over the text cube of $\mathcal{DB}$ in Table 2 is given, then for row $r = (\mathrm{m1}, \mathrm{p1}, \mathrm{t1}, \mathrm{s1})$, we only update relevance scores of its ancestors $(\mathrm{m1}, \mathrm{p1}, \mathrm{t1}, *)$, $(*, \mathrm{p1}, \mathrm{t1}, *)$, $(\mathrm{m1}, *, \mathrm{t1}, *)$, and $(*, *, \mathrm{t1}, *)$ in lines 3-4. Similar modification can be also applied to lines 2-3 of Algorithm 2.

3.4 Deficiencies of Linear-Scan Approaches

Algorithm 1 is efficient only when the number of dimensions is small (from 2 to 4). Because, it scores all the nonempty cells but the number of nonempty cells increases exponentially w.r.t. the dimensionality (see Section 2.4).

Algorithm 2 aims to improve Algorithm 1 by enabling early stop. However, once a row is scanned, all the cells containing it are explored. There are totally 2^n cells containing a document in a n-dim cube. So, the numbers of candidate cells and explored cells increase very quickly.

In the next section, we will introduce a more delicately designed top-k algorithm, *search-space ordering*, which carries out cell-based search and explores as small number of cells in the cube as possible to find the top-k answers.

1. There are other top-k algorithms to obtain this ranked list, e.g., [21]. But the cost of Algorithm 2 is dominated by the cost of ranking cells, instead of ranking documents, so we just apply this simple method.

4 Search-Space Ordering Approach

Unlike the two algorithms in Section 3 which compute the relevance score of a cell from documents/rows in the database, the algorithms introduced in this section compute the score from its children cells, in a dynamic programming manner. We introduce the basic *bottom-up dynamic programming* algorithm in Section 4.1, followed by the design and analysis of an more efficient approach, *search-space ordering*. We start with simple keyword queries and discuss how to extend our algorithm in Section 4.4.

4.1 Bottom-Up Dynamic Programming

Consider a cell C' and all of its $\mathcal{A}$-children $\mathrm{chd}_{\mathcal{A}}(C')$ for some aggregated dimension $\mathcal{A}$ ($C'[\mathcal{A}] = *$ but $C[\mathcal{A}] \neq *$ for $C \in \mathrm{chd}_{\mathcal{A}}(C')$). It is clear that the sets of documents aggregated in different cells in $\mathrm{chd}_{\mathcal{A}}(C')$ are disjoint and their union is equal to $C'[\mathcal{D}]$, i.e., $C'[\mathcal{D}] = \bigcup_{C \in \mathrm{chd}_{\mathcal{A}}(C')} C[\mathcal{D}]$. From the definition of relevance (1), we have recursion:

$$\mathrm{rel}(\mathrm{q}, C') = \frac{\sum_{C \in \mathrm{chd}_{\mathcal{A}}(C')} \mathrm{rel}(\mathrm{q}, C)|C[\mathcal{D}]|}{\sum_{C \in \mathrm{chd}_{\mathcal{A}}(C')} |C[\mathcal{D}]|}, \tag{3}$$

where $|C[\mathcal{D}]|$ is the number of documents in $C[\mathcal{D}]$.

4.1.1 Naive Dynamic Programming Algorithm

A naive dynamic programming algorithm that directly utilizes (3) is as follows: we first compute the relevance scores $\mathrm{rel}(\mathrm{q}, C)$ for all n-dim (base) cells, as in (1). Then, we compute the relevance scores $\mathrm{rel}(\mathrm{q}, C')$ of $(n-1)$-dim cells, $(n-2)$-dim cells, ..., and 0-dim cell, in the decreasing order of dimensionality, using the recursion (3); this is because for a d-dim cell C', the right-hand side of (3) involves only $d+1$ cells. Finally, after the relevance scores of all cells are obtained, output the top-k relevant ones with supports no less than the threshold minsup.

4.1.2 Bottom-Up Aggregation: From Partially Aggregated to Fully Aggregated

To compute $\mathrm{rel}(\mathrm{q}, C')$, a direct implementation of (3) could be inefficient if $\min_{\mathcal{A}}|\mathrm{chd}_{\mathcal{A}}(C')| = O(\min_{\mathcal{A}}|\mathcal{A}|)$ is large. We will introduce a bottom-up aggregation approach whose complexity is independent of $\min_{\mathcal{A}}|\mathcal{A}|$, but only dependent on n (the total number of dimensions). It is also an important basis of our search-space ordering algorithm which will be introduced in Section 4.3.

For each aggregated dimension $\mathcal{A}$ of a cell C' ($C'[\mathcal{A}] = *$), we maintain a *partial relevant score* $\overline{\mathrm{rel}}_{\mathcal{A}}(\mathrm{q}, C')$ based on the $\mathcal{A}$-children that are already aggregated into C'. Let $L_{\mathcal{A}}(C')$ be *the number of documents contained by the $\mathcal{A}$-children that are already aggregated into C'*. A nonbase cell C' is said to be *fully aggregated* iff for *some dimension* $\mathcal{A}$, $L_{\mathcal{A}}(C') = |C'[\mathcal{D}]|$ and $\overline{\mathrm{rel}}_{\mathcal{A}}(\mathrm{q}, C') = \mathrm{rel}(\mathrm{q}, C')$; otherwise, C' is said to be *partially aggregated*.

Initially, all cells are partially aggregated and $\mathrm{rel}(\mathrm{q}, C)$'s are *unknown* for all cells; and we set $\overline{\mathrm{rel}}_{\mathcal{A}}(\mathrm{q}, C) = \infty$ and $L_{\mathcal{A}}(C) = 0$ for each cell C and each of its aggregated dimension $\mathcal{A}$. We compute $\mathrm{rel}(\mathrm{q}, C)$ for all base cells from the database. After that, *all base cells are fully aggregated*.

For each fully aggregated cell C and each of its nonaggregated dimensions $\mathcal{A}$ ($C[\mathcal{A}] \neq *$), we aggregate C to its $\mathcal{A}$-parent C' as follows (define $\infty \times 0 = 0$):

Aggregate (C, C'): $C \in \mathrm{chd}_{\mathcal{A}}(C')$

1.
$$\overline{\mathrm{rel}}_{\mathcal{A}}(\mathrm{q}, C') \leftarrow \frac{\overline{\mathrm{rel}}_{\mathcal{A}}(\mathrm{q}, C')L_{\mathcal{A}}(C) + \mathrm{rel}(\mathrm{q}, C)|C[\mathcal{D}]|}{L_{\mathcal{A}}(C') + |C[\mathcal{D}]|}; \tag{4}$$

2.
$$L_{\mathcal{A}}(C') \leftarrow L_{\mathcal{A}}(C') + |C[\mathcal{D}]|; \tag{5}$$

3.
$$\textbf{if } L_{\mathcal{A}}(C') = |C'[\mathcal{D}]| \textbf{ then } \mathrm{rel}(\mathrm{q}, C') \leftarrow \overline{\mathrm{rel}}_{\mathcal{A}}(\mathrm{q}, C'). \tag{6}$$

When $L_{\mathcal{A}}(C') = |C'[\mathcal{D}]| = \sum_{C \in \mathrm{chd}_{\mathcal{A}}(C')} |C[\mathcal{D}]|$, i.e., all documents in $C'[\mathcal{D}]$ have been aggregated into C' through its $\mathcal{A}$-children, from (3), we have $\mathrm{rel}(\mathrm{q}, C') = \overline{\mathrm{rel}}_{\mathcal{A}}(\mathrm{q}, C')$ obtained and thus C' becomes fully aggregated. We repeat this procedure until all cells are fully aggregated. Note that from the definition of "fully aggregated," we only need to store $\overline{\mathrm{rel}}_{\mathcal{A}}(\mathrm{q}, C')$ and $L_{\mathcal{A}}(C')$ on one aggregated dimension $\mathcal{A}$ (e.g., the one with the smallest index) for each cell.

Algorithm 3 describes this approach. In each iteration of lines 2-6, all of nonempty d-dim cells are fully aggregated, because all of nonempty n-dim, $(n-1)$-dim, ..., $(d+1)$-dim cells are scanned. We keep all nonempty cells in a queue so that we do not need to touch any empty cell. Note that a nonempty cell C has $C[\mathcal{D}] \neq \emptyset$ but possibly $\mathrm{rel}(\mathrm{q}, C) = 0$. From the recursion (3) how and $\overline{\mathrm{rel}}_{\mathcal{A}}(\mathrm{q}, C')$ is updated in (4-5), we can prove that Algorithm 3 is correct.

Algorithm 3. Bottom-up Dynamic Programming Algorithm
Input: keyword query q against $\mathcal{DB}$, parameter k, minsup
1: Compute $\mathrm{rel}(\mathrm{q}, C)$'s for all non-empty base cells;
2: **for** $d = n$ downto 1 **do**
3: **for each** *non-empty* d-dim cell C **do**
4: **for each** dimension $\mathcal{A}$ **do**
5: Let C' be C's $\mathcal{A}$-parent;
6: Call **Aggregate**(C, C');
7: Output the top-k relevant cells with support $\geq \mathrm{minsup}$ (for cells with the same relevance scores, output them in the descending order of support).

Example 4.1. We show how $\overline{\mathrm{rel}}(\mathrm{q}, C')$ is computed in Algorithm 3. Suppose there are five documents $\mathrm{d}_1, \ldots, \mathrm{d}_5$ with document scores $s(\mathrm{q}, \mathrm{d}_1) = 8$, $s(\mathrm{q}, \mathrm{d}_2) = 4$, $s(\mathrm{q}, \mathrm{d}_3) = 4$, $s(\mathrm{q}, \mathrm{d}_4) = 6$, and $s(\mathrm{q}, \mathrm{d}_5) = 2$. Also, suppose $C' = (\mathrm{a}1, *, *)$ has three B-children and three C-children, which are all fully aggregated. Documents in C''s children and cell relevance scores are shown in Table 4.

When $C = (\mathrm{a}1, \mathrm{b}1, *)$ and $(\mathrm{a}1, *, \mathrm{c}1)$ are aggregated into C', using **Aggregate**(C, C'), we have $L_{\mathrm{B}}(C') = L_{\mathrm{C}}(C') = 2$, and partial scores $\overline{\mathrm{rel}}_{\mathrm{B}}(\mathrm{q}, C') = \overline{\mathrm{rel}}_{\mathrm{C}}(\mathrm{q}, C') = 6$.

TABLE 4
B-Children and C-Children of $C' = (a1, *, *)$

$\mathrm{chd_B}(C')$					$\mathrm{chd_C}(C')$				
A	B	C	D	rel	A	B	C	D	rel
a1	b1	*	$\{d_1, d_2\}$	6	a1	*	c1	$\{d_1, d_2\}$	6
a1	b2	*	$\{d_3, d_4\}$	5	a1	*	c2	$\{d_3\}$	4
a1	b3	*	$\{d_5\}$	2	a1	*	c3	$\{d_4, d_5\}$	4

After B-child $(a1, b2, *)$ is aggregated into C', we have $\overline{\mathrm{rel}}_B(q, C') = (6 \times 2 + 5 \times 2)/4 = 5.5$, and $L_B(C') = 4$.

After C-child $(a1, *, c2)$ is aggregated into C', we have $\overline{\mathrm{rel}}_C(q, C') = (6 \times 2 + 4 \times 1)/3 = 5.33$, and $L_C(C') = 3$.

Later, when at least one of $(a1, b3, *)$ and $(a1, *, c3)$ is aggregated into C', C' is fully aggregated. But actually, we only need to keep either ($\overline{\mathrm{rel}}_B$, L_B) or ($\overline{\mathrm{rel}}_C$, L_C).

Theorem 4. *Algorithm 3 uses $O(N \cdot |q| + n \cdot M)$ time and $O(n \cdot M)$ space, where $N = |\mathcal{DB}|$, M is the total number of nonempty cells, and n is the number of dimensions.*

Proof. Nonempty base cells are computed in $O(N \cdot |q|)$ time. All nonempty cells are kept in queue s.t. the ordered access of each takes $O(1)$ time. Lines 4-6 need $O(n)$ time. So, we need $O(n \cdot M)$ time to compute $\mathrm{rel}(q, C)$ for all nonempty cells. We use Hoare's selection algorithm to select the top-k from all nonempty cells in $O(M)$ time.

A simple implementation needs $O(n \cdot M)$ space: for each cell, we keep $\overline{\mathrm{rel}}_{\mathcal{A}}(q, C)$ and $L_{\mathcal{A}}(C)$ for every dimension $\mathcal{A}$. But we actually only need to keep them on one aggregated dimension (e.g., the one with the smallest index). □

In the rest of this paper, for each cell C, we store $\overline{\mathrm{rel}}_{\mathcal{A}}(C)$ and $L_{\mathcal{A}}(C)$ for only one dimension. So, we will also write $\overline{\mathrm{rel}}_{\mathcal{A}}(C)$ as $\overline{\mathrm{rel}}(C)$, if not otherwise specified.

4.2 Upper/Lower Bounds of Relevance Score

We first introduce a pair of upper/lower bounds of relevance scores, which will be used to prune search space of our algorithm in Section 4.3. The main idea is: the relevance score of the union of two document sets lies between the relevance scores of these two document sets.

Lemma 1. *For any query q and any two disjoint document sets D_1, D_2 s.t. $\mathrm{rel}(q, D_1) \le \mathrm{rel}(q, D_2)$, we have $\mathrm{rel}(q, D_1) \le \mathrm{rel}(q, D_1 \cup D_2) \le \mathrm{rel}(q, D_2)$.*

Proof. From the definition in (1),

$$\mathrm{rel}(q, D_1 \cup D_2) = \frac{|D_1| \cdot \mathrm{rel}(q, D_1) + |D_2| \cdot \mathrm{rel}(q, D_2)}{|D_1| + |D_2|}.$$

Since $\mathrm{rel}(q, D_1) \le \mathrm{rel}(q, D_2)$,

$$\mathrm{rel}(q, D_1 \cup D_2) \le \frac{|D_1| + |D_2|}{|D_1| + |D_2|} \cdot \mathrm{rel}(q, D_2) = \mathrm{rel}(q, D_2),$$

and similarly, $\mathrm{rel}(q, D_1 \cup D_2) \ge \mathrm{rel}(q, D_1)$. □

Example 4.2. To verify Lemma 1, consider $D_1 = \{d_1, d_2\}$ and $D_2 = \{d_3\}$. Suppose for some query q, we have $s(q, d_1) = 1, s(q, d_2) = 3, s(q, d_3) = 8$. Then, $\mathrm{rel}(q, D_1) = 2$, $\mathrm{rel}(q, D_2) = 8$, and $\mathrm{rel}(q, D_1 \cup D_2) = 4$.

Lemma 2 (Two-Side Bound Property). *For any nonbase cell C in text cube and any query q, there exist two children C_1 and C_2 of C s.t. $\mathrm{rel}(q, C_1) \le \mathrm{rel}(q, C) \le \mathrm{rel}(q, C_2)$.*

Proof. Consider $\mathcal{A}$-children of C: $\bigcup_{C' \in \mathrm{chd}_{\mathcal{A}}(C)} C'[\mathcal{D}] = C[\mathcal{D}]$, where $C'[\mathcal{D}]$'s are disjoint. So from Lemma 1, the proof can be completed by induction on $|\mathrm{chd}_{\mathcal{A}}(C)|$. □

Remark. Lemmas 1 and 2 are still true for different document scoring formulas $s(q, d)$ and the weighted version of $\mathrm{rel}(q, C)$ (i.e., $\mathrm{rel}(q, C)$ is a weighted average of $s(q, d)$ over all documents d in $C[\mathcal{D}]$, where the weight of a document w_d is fixed and positive).

4.3 Search-Space Ordering Algorithm

Our *search-space ordering* approach is based on the following idea. Although the search space is huge (especially if the dimensionality is reasonably large, say, $\ge$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, 1) we estimate the upper bound of relevance scores in a subspace of the text cube, and 2) we *order* and *prune* subspaces so that we can follow a shortcut to the top-k relevant cells with support $\ge$ minsup.

4.3.1 Algorithm Framework

The main idea is: *order* and *prune* the search space of cells using (the upper bound of) relevance scores.

Bottom-up aggregation. Our algorithm works in a similar manner as the bottom-up dynamic programming algorithm (Algorithm 3). Relevance scores are computed from the base cells (whose relevance scores can be computed with the row-based inverted indexes efficiently) to higher levels.

Search-space ordering. But unlike line 3 in Algorithm 3 (where the next cell C to be aggregated to its parents is selected arbitrarily), we pick the most relevant *fully aggregated cell* in each iteration and aggregate it to its parents. This is because a highly relevant cell is likely to have highly relevant children. Note that a fully aggregated cell needs to be aggregated to its parents at most once.

Search-space pruning. The relevance score of a fully aggregated cell is known. For a partially aggregated cell, we have the upper bound from Lemma 2. When *a fully aggregated cell has its relevance score no lower than the upper bound of relevance scores of all partially aggregated cells*, it is output as one of the top-k (if support $\ge$ minsup).

We will discuss the details of this algorithm and upper bound estimation for relevance scores in Section 4.3.2.

4.3.2 Algorithm Description

Three structures are used in our algorithm: initially, all base cells are put into *aggregation queue* (Q_A), and the base cells with support no less than minsup are put into *candidate queue* (Q_C), both maintained in the nonincreasing order of relevance scores. The *partially aggregated pool* (P_A) is empty. More cells will be inserted into Q_A and Q_C later. If an ancestor has the same relevance score as its descendant, the ancestor has higher priority in these two queues.

- *Aggregation queue.* In each iteration, the cell C with the highest score $\mathrm{rel}(q, C)$ in Q_A is popped out and aggregated into each of its parents C' using

Aggregate(C, C'), i.e., (4-6), with the partial relevance score $\overline{\text{rel}}(\text{q}, C')$ computed/updated (since a highly relevant cell likely has highly relevant children). After that, if C' is partially aggregated, it is put into the partially aggregated pool P_A if C' is not in P_A; if C' becomes fully aggregated, then: 1) C' is deleted from P_A and inserted into Q_A, and 2) if its support $\geq$ minsup, C' is inserted into Q_C.

- *Partially aggregated pool*. It maintains partially aggregated cells (and partial relevance scores $\overline{\text{rel}}(\text{q}, \cdot)$), each of which has at least one child aggregated into the cell. Note that it is possible that no child of a cell C is aggregated into C, and thus C (still called partially aggregated cell) is not in P_A.
- *Candidate queue*. All fully aggregated cells are maintained in Q_C if support $\geq$ minsup. When a cell in Q_C has a relevance score higher than the upper bound of relevance scores of all partially aggregated cells, it can be output as one of the top-k. We will prove that such an upper bound is: max {*maximum partial relevance score in* P_A, *maximum relevance score in* Q_A}.

The framework of this algorithm is outlined in Algorithm 4. Given a keyword query q, the relevance scores of base cells can be computed using inverted indexes (line 1). Initially, P_A is empty, all base cells are inserted into Q_A, and the ones with support $\geq$ minsup are inserted into Q_C. In each iteration of lines 3-14, the top cell C in Q_A is popped (deleted from Q_A) for aggregation (line 4). For each partially aggregated parent C' of C, we insert it into P_A if necessary (line 6); and we can compute/update the partial relevance score $\overline{\text{rel}}(\text{q}, C')$ using **Aggregate**(C, C') (line 7), as in (4-6). If the cell C' is *fully aggregated* (for some dimension $\mathcal{A}$, all of its $\mathcal{A}$-children have already been aggregated into C'), we can compute its exact relevance score $\text{rel}(\text{q}, C')$, delete it from P_A, and add it into Q_A; it is also added into Q_C if its support $\geq$ minsup (lines 8-11). At the end of each iteration (lines 12-14), the top cells in Q_C can be output (also deleted from Q_C) if *their scores are no less than the maximum partial relevance score in* P_A *and the maximum relevance score in* Q_A (i.e., *the upper bound of relevance scores of partially aggregated cells*).

Algorithm 4. Search-Space Ordering Algorithm
Q_A and Q_C: priority queues of cells (sorted in the non-increasing order of relevance scores)
top(Q): the top (first) cell of a priority queue Q
P_A: the set of partially-aggregated cells
Input: keyword query q against $\mathcal{DB}$, parameter k, minsup

1: Compute $\text{rel}(\text{q}, C)$'s for all non-empty base cells;
2: $cnt \leftarrow 0$, $\text{Q}_\text{A} \leftarrow \{$all non-empty base cells$\}$,
$\text{Q}_\text{C} \leftarrow \{$*non-empty base cells with support* $\geq$ 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 $C' \in \text{par}(C)$ **do**
6: **if** $C' \notin \text{P}_\text{A}$ **then** insert C' into P_A;
7: Aggregate C into C' using **Aggregate**(C, C'):
Update partial relevance score $\overline{\text{rel}}(\text{q}, C')$;
8: **if** C' is fully-aggregated (all documents in $C'[\mathcal{D}]$ are aggregated into C' on some dimension) **then**

TABLE 5
Base Cuboid of a Two-Dimensional Text Database

M	P	C[D]	rel(q, C)
m1	p1	{d_1}	8
m1	p2	{d_2}	6
m1	p3	{d_3, d_7}	4
m2	p1	{d_4}	8
m2	p2	{d_5}	6
m2	p3	{d_6, d_8}	4

9: Compute $\text{rel}(\text{q}, C')$;
10: Delete C' from P_A;
11: Insert C' into Q_A;
Insert C' into Q_C **if** support $|C'| \geq$ minsup;
12: **while** $\text{rel}(\text{q}, \textbf{top}(\text{Q}_\text{C})) \geq \max_{C' \in \text{P}_\text{A}}\{\overline{\text{rel}}(\text{q}, C')\}$ and $\text{rel}(\text{q}, \textbf{top}(\text{Q}_\text{C})) \geq \text{rel}(\text{q}, \textbf{top}(\text{Q}_\text{A}))$ **do**
13: Output the top cell of Q_C, and delete it from Q_C;
14: $cnt \leftarrow cnt + 1$;

We give some intuitions on why the above upper bound is valid. In our algorithm, a loop invariant is: for any partially aggregated cell C (no matter whether it is in P_A), the union of P_A and Q_A forms a cut between C and the base cells in the cube lattice. So, from the two-side bound property (Lemma 2), the relevance score of C is upper bounded by the (partial) relevance score of a descendant cell in $\text{P}_\text{A} \cup \text{Q}_\text{A}$. Formal proofs will be given in Section 4.3.3.

Example 4.3. This example shows the basic idea of our algorithm, using a two-dimensional text database. Table 5 shows the base cuboid of this database—each base cell may contain more than one document. Consider a keyword query $\text{q} = \{\text{w1}, \text{w2}\}$ and $\text{minsup} = 2$. Suppose the relevance score $\text{rel}(\text{q}, C)$ (defined in (1-2)) of each base cell is computed as in the fourth column of Table 5.

Initially, all base cells are put into Q_A, and the ones with support no less than minsup are put into Q_C, both in the nondecreasing order of relevance scores. P_A is empty.

We show how to maintain $\text{Q}_\text{A}, \text{Q}_\text{C}, \text{P}_\text{A}$ in Table 6. Number in bracket in the first column is the number of iterations, e.g., $\text{P}_\text{A}(2)$ means the status of P_A at the end of the second iteration; the number after each cell in the second column is the relevance score (or partial relevance score in P_A).

Enlarging k at running time without recomputation. With k unspecified, repeating the iteration of lines 3-15, cells with support $\geq$ minsup will be output in the nonincreasing order of relevance scores. Users can stop it at any time when they feel the results are enough.

Tie breaker. Similar to Algorithm 2, ties (for cells with the same relevance scores) break arbitrarily in Algorithm 4. If we want to rank cells with the same relevance scores in the descending order of their support, Algorithm 4 is revised as follows: first, the candidate queue Q_C keeps cells in the descending order of relevance, and keeps cells with the same relevance scores in the descending order of support. Second, "$\geq$" should be changed into ">" on line 12, so that, when a cell is about to be output, all the cells (not output yet) with the same relevance scores are in Q_C.

TABLE 6
Running Example

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

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

2^{nd} *iteration*: (m2,p1) is popped from $Q_A(1)$, and aggregated into the two parents, (m2,*) and (*,p1): (m2,*) is inserted into $P_A(2)$ with the partial relevance score computed as 8; (*,p1) are already in $P_A(1)$. Then, (*,p1) is *fully aggregated*, since it has only two M-children, (m1,p1) and (m2,p1). So (*,p1) is deleted from $P_A(1)$ and inserted into $Q_A(2)$. It is also inserted into $Q_C(2)$ for its support $\geq$ minsup. Finally, the top cell in $Q_C(2)$, (*,p1), has relevance score no less than the maximum partial relevance score in $P_A(2)$ and the maximum relevance score in $Q_A(2)$, so (*,p1) is output.	
$P_A(2)$	(m1,*):8, (m2,*):8
$Q_A(2)$	(*,p1):8, (m1,p2):6, (m2,p2):6, (m1,p3):4, (m2,p3):4
$Q_C(2)$	(*,p1):8, (m1,p3):4, (m2,b3):4

3^{rd} *iteration*: Cell (*,p1) is popped from $Q_A(2)$, and its parent (*,*) is added into $P_A(3)$.	
$P_A(3)$	(m1,*):8, (m2,*):8, (*,*):8
$Q_A(3)$	(m1,p2):6, (m2,p2):6, (m1,p3):4, (m2,p3):4
$Q_C(3)$	(m1,p3):4, (m2,p3):4

4^{th} *iteration*: (m1,p2) is popped from $Q_A(3)$, and aggregated into its two parents, (m1,*) and (*,p2). (m1,*) is already in $P_A(3)$, and its partial relevance score is updated as 7 in $P_A(4)$, since we have seen two children of (m1,*), (m1,p1) and (m1,p2), with score 8 and 6, respectively. (*,p2) is inserted into $P_A(4)$ with partial score 6.	
$P_A(4)$	(m2,*):8, (*,*):8, (m1,*):7, (*,p2):6
$Q_A(4)$	(m2,p2):6, (m1,p3):4, (m2,p3):4
$Q_C(4)$	(m1,p3):4, (m2,p3):4

Theorem 5. *Algorithm 4 needs $O(N \cdot |q| + n \cdot T \cdot \log T)$ time and $O(n \cdot T)$ space, where $N = |\mathcal{DB}|$, n is the number of dimensions, and T is the number of cells explored by this algorithm (i.e., the ones used to be added into Q_A, Q_C, P_A).*

Proof. Again, nonempty base cells are computed in $O(N \cdot |q|)$ time. After a cell explored is fully aggregated, on lines 5-11, it is aggregated into its (at most n) parents (using $O(n)$ time), and one operation on Q_A, Q_C, P_A (using $O(\log T)$ time) is invoked for each parent.

The space needed is $O(n \cdot T)$. This is because, for each cell explored, we keep its n-dim identifier and $\overline{\text{rel}}_{\mathcal{A}}(q, C)$ and $L_{\mathcal{A}}(C)$ for one dimension $\mathcal{A}$. And, a pointer to this cell is kept in Q_A, Q_C, P_A using $O(1)$ space. □

The total number of nonempty cells, M (recall Theorem 4), is an upper bound of T. However, T is determined by both data distribution in the database and the query q. So, we cannot get a preciser estimation of it beforehand. We will report the actual values of T in our experiments, and show that Algorithm 4 explores a much smaller portion of cells than other algorithms.

4.3.3 Correctness

We prove the correctness of Algorithm 4 in the following part ($s(\cdot, \cdot)$ not restricted to the form of (2)).

Lemma 3 (Estimating Upper Bound of Relevance Scores). *For any query q, on line 12 of Algorithm 4, if*

1. $\text{rel}(q, \mathbf{top}(Q_C)) \geq \max_{C' \in P_A}\{\overline{\text{rel}}(q, C')\}$ *and*
2. $\text{rel}(q, \mathbf{top}(Q_C)) \geq \text{rel}(q, \mathbf{top}(Q_A))$,

then cell $\mathbf{top}(Q_C)$ has a relevance score no less than the scores of all partially aggregated cells.

Proof. Let $\Delta = \text{rel}(q, \mathbf{top}(Q_C))$. For the purpose of contradiction, suppose there is a cell C'' that is *partially aggregated* and the relevance score $\text{rel}(q, C'') > \Delta$.

We focus on $\mathcal{A}$-children of C'' for its selected aggregated dimension $\mathcal{A}$, s.t. $\overline{\text{rel}}(q, C'') = \overline{\text{rel}}_{\mathcal{A}}(q, C'')$.

Let $P(C'')$ denote the set of C'''s $\mathcal{A}$-children that are partially aggregated, and $F(C'')$ denote the set of C'''s $\mathcal{A}$-children that are fully aggregated but not aggregated into C yet. Recall $L_{\mathcal{A}}(C'')$ is the number of documents that are already aggregated into C'' from its $\mathcal{A}$-children.

If $L_{\mathcal{A}}(C'') = 0$, we know that no child is aggregated into C''. Then, $P(C'') \cup F(C'')$ covers all of C'''s $\mathcal{A}$-children. Note $\text{rel}(q, C'') > \Delta$. From Lemma 2, there exists $C \in P(C'') \cup F(C'')$ s.t. $\text{rel}(q, C) \geq \text{rel}(q, C'') > \Delta$.

If $L_{\mathcal{A}}(C'') \neq 0$, $C'' \in P_A$. From condition 1 in the lemma, $\overline{\text{rel}}(q, C'') \leq \max_{C' \in P_A} \overline{\text{rel}}(q, C') \leq \Delta$. Again note $\text{rel}(q, C'') > \Delta$. From Lemmas 1-2, there exists a child $C \in P(C'') \cup F(C'')$ s.t. $\text{rel}(q, C) > \Delta$.

For the selected children C of C'', there are two cases:

1. $C \in F(C'')$. We must have $C \in Q_A$, since otherwise, C is already aggregated into C'' when C is popped from Q_A. Then, $\text{rel}(q, \mathbf{top}(Q_A)) \geq \text{rel}(q, C) > \Delta$, which contradicts with $\Delta \geq \text{rel}(q, \mathbf{top}(Q_A))$.
2. $C \in P(C'')$. C is partially aggregated. Repeat the above argument about C'' for C (or induction on the number of aggregated dimensions in C'').

Since the text cube is finite, that means, if $\text{rel}(q, C'') > \Delta$ for some C'', we will always terminate with a descendant C of C'' with contradiction. So the proof is completed. □

The above proof holds if we choose $\overline{\text{rel}}(q, C'') = \overline{\text{rel}}_{\mathcal{A}}(q, C'')$ for any aggregated dimension $\mathcal{A}$. So if choosing $\overline{\text{rel}}(q, C'') = \min_{\mathcal{A}} \overline{\text{rel}}_{\mathcal{A}}(q, C'')$, our algorithm has a tighter upper bound (it may terminate earlier but incurs more space to store $\overline{\text{rel}}_{\mathcal{A}}$ for each dimension).

Theorem 6. *Algorithm 4 outputs the top-k cells with the highest relevance scores and support $\geq$ minsup in the nonincreasing order of $\text{rel}(q, C)$.*

Proof. For any cell C with support $\geq$ minsup, before it is output, it is either partially aggregated or in Q_C. From Lemma 3, when the cell $\mathbf{top}(Q_C)$ is output, its relevance score is no less than the relevance score of any partially aggregated cell. So, from the way that Q_C is ordered (nonincreasing order of relevance scores), *for any $cnt = k$* on lines 13-14 of Algorithm 4, $\mathbf{top}(Q_C)$ is the cell with the top-k highest relevance score among all cells with support $\geq$ minsup; thus, top-k cells are output in the nonincreasing order of relevance scores. □

TABLE 7
Laptop Review Database: Dimensions and Statistics

Dimension	Description	Cardinality
1. Laptop Brand	brand of the laptop	20
2. Laptop Family	family of the laptop	26
3. Screen	size of the screen	26
4. Memory	size of the main memory	9
5. Disk	size of the hard drive	28
6. CPU Brand	brand of the processor	5
7. CPU Model	model of the processor	148
8. CPU Speed	speed of the processor	33
9. OS	pre-installed os	12
10. Color	color of the laptop	10

4.4 Handling Extended Keyword Queries

We modify Algorithm 4 as follows to handle extended keyword queries $Q = (u_1, u_2, \ldots, u_n : \mathrm{q})$:

1. On lines 1-2, instead of inserting base cells into $\mathrm{Q_A}$ and $\mathrm{Q_C}$, we insert the cells C's with a) $C[\mathcal{A}_i] = *$ if $u_i = *$ (dimension $\mathcal{A}_i$ must be aggregated), b) $C[\mathcal{A}_i] = u_i$ if $u_i \in \mathcal{A}_i$ is a specific value, and c) $C[\mathcal{A}_i] \in \mathcal{A}_i$ is any specific value if $u_i = ?$. One scan of inverted indexes of terms in q suffices to identify these cells and insert them into $\mathrm{Q_A}$ and $\mathrm{Q_C}$.
2. On lines 5-14, we consider $C' \in \mathrm{par}(C)$, only if C' is a feasible cell, which satisfies the constraint in Q.

Algorithm 4 is even *faster* when handling an extended keyword query than when handling a simple query with the same set of keywords, because more constraints on dimensions actually reduce the size of search space.

5 Experimental Study

5.1 Data Sets and Environment Setup

A real data set of multidimensional text data is used in the experiments. We crawled laptop reviews with multidimensional attributes from Amazon.com, and then select 10 attributes as the structural dimensions in our database (Table 7). Each review (*document*) together with its categorical attributes (*dimensions*) forms a row in our database. We have 13,930 reviews in sum, 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.

Synthetic data sets are generated by adding randomly generated documents into the real data set. To this end, we estimate the overall distributions of terms, document length, and attribute values for different dimensions. Five synthetic data sets, $\mathcal{DB}_1, \mathcal{DB}_2, \ldots, \mathcal{DB}_5$ are used in Section 5.2 to test the scalability of our algorithms w.r.t. the size of text database. The numbers of rows are: $|\mathcal{DB}_1| = 28$ K, $|\mathcal{DB}_2| = 56$ K, $|\mathcal{DB}_3| = 84$ K, $|\mathcal{DB}_4| = 112$ K, and $|\mathcal{DB}_5| = 140$ K. $\mathcal{DB}_i$ is a subset of $\mathcal{DB}_{i+1}$ for $i = 1, 2, 3, 4$.

All the experiments were conducted on a PC running the Microsoft Windows XP SP2 Professional OS, with a 2.5 GHz Intel Core 2 Duo T9300 CPU, 4 GB of RAM, and 150 GB hard drive. Our algorithms were implemented in C/C++ and compiled on Microsoft Visual Studio 2008.

We use Okapi weighting (2) for $s(\mathrm{q}, \mathrm{d})$ in experiments.

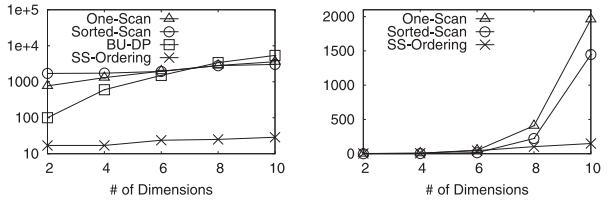


Fig. 1. Varying the number of dimensions. (a) Time (msec). (b) Memory (KB).

5.2 Performance Study on Efficiency

We test our four algorithms, **One-Scan** (Algorithm 1), **Sorted-Scan** (Algorithm 2), **BU-DP** (Algorithm 3) and **SS-Ordering** (Algorithm 4), to demonstrate their scalability on both the real data set (laptop reviews) and the synthetic data sets. We report the time and memory used by these four algorithms for finding top-k answers to given keyword queries. Time is measured in terms of milliseconds (msec), and memory is measured in terms of KBs. We also report the number of cells that are explored by these algorithms, to demonstrate the pruning power of our algorithms.

We report only the *running-time* memory, i.e., the amount of memory consumed by data structures for query processing (excluding the inverted indexes). **One-Scan** and **BU-DP** consume almost the same amount memory and explore the same number of cells. So, we only report **One-Scan** for memory and the number of cells.

In the real data set, we vary the number of dimensions (Fig. 1), the parameter of k (Fig. 4), the number of keywords $|\mathrm{q}|$ (Fig. 5), and the minimum support requirement minsup (Fig. 6). Also, the five synthetic data sets are used to test the scalability of these three algorithms w.r.t. the sizes of databases (Fig. 2).

All the results reported below are the averages for 20 typical keyword queries (with no dimension-value constraint) for laptops. Each query has four keywords on average.

We omit the results of efficiency experiments for extended keyword queries, because as discussed in Sections 3.3 and 4.4, adding such constraints will not slow down our algorithms. In fact, to handle an extended keyword query $Q = (u_1, u_2, \ldots, u_n : \mathrm{q})$ with m dimensions $\mathcal{A}_i$'s s.t. $Q[\mathcal{A}_i] = u_i = ?$ (dimension $\mathcal{A}_i$ has no constraint), it is equivalent for our algorithms to handle a simple keyword query q in an m-dim text cube.

5.2.1 Preprocessing

As discussed in Section 2.3, our algorithms benefit from preprocessing (i.e., precomputing the support of each cell

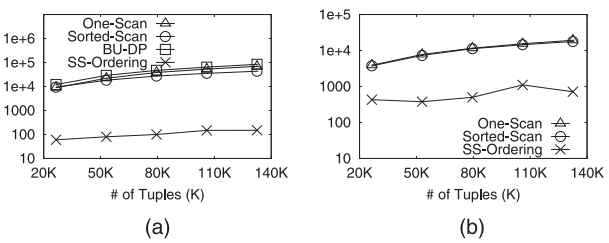


Fig. 2. Varying the size of database $|\mathcal{DB}|$. (a) Time (msec). (b) Memory (KB).

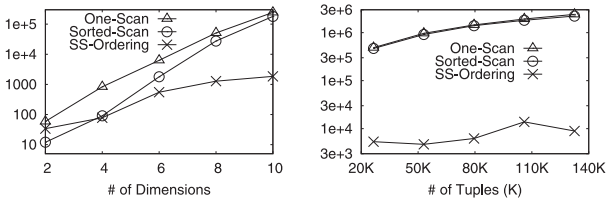


Fig. 3. Number of cells explored. (a) Varying # of dimensions. (b) Varying the size of database.

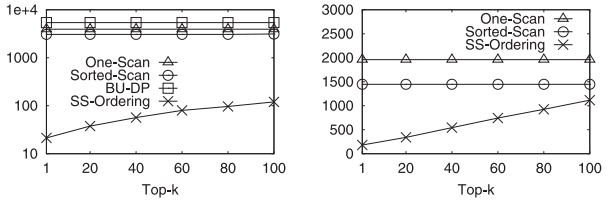


Fig. 4. Varying the parameter k. (a) Time (msec). (b) Memory (KB).

and constructing inverted indexes). In our implementation, it takes $4\frac{1}{2}$ minutes to preprocess the real data set described above, which is nontrivial but affordable since we only need to preprocess a data set once.

5.2.2 Experiment I: Number of Dimensions (Fig. 1)

We use the real data set (with about 14 K rows). Set $k = 10$ and $\text{minsup} = 1$. The database is projected into the first n dimensions, for $n = 2, 4, 6, 8, 10$ (refer to Table 7).

SS-Ordering is consistently at least 10 times faster than the others, and consumes much less memory (except when $n = 2$). BU-DP is the second fastest one when $n \leq 6$ but becomes worst later on.

Sorted-Scan has an improvement of 20 percent over One-Scan for 10-dim cube, but is slower than One-Scan for 4-dim cube. This is because they have the same worst case complexity and Sorted-Scan needs to maintain additional data structures; so sometimes when its pruning power is not strong, it needs more time than One-Scan.

When n increases, the number of nonempty cells increases exponentially, and thus all algorithms use more time and memory. SS-Ordering is less sensitive than the rest three, because SS-Ordering only explores a small portion of the cells, while One-Scan and BU-DP always explore all the nonempty cells. Sorted-Scan explores less cells than One-Scan, but for the additional operation required, the improvement over One-Scan on efficiency is not significant.

The experimental result of testing extended keyword queries $Q = (u_1, u_2, \ldots, u_n : \text{q})$ with m no-constraint dimensions $\mathcal{A}_i$'s (i.e., $Q[\mathcal{A}_i] = u_i = ?$) is similar to the above, when we vary m from 2 to 10.

5.2.3 Experiment II: Size of Database (Fig. 2)

Synthetic data sets $\mathcal{DB}_1, \ldots, \mathcal{DB}_5$ are used. These data sets have 10 dimensions and their sizes vary from 28 to 140 K. Top-10 answers are output with $\text{minsup} = 1$. Sorted-Scan is always faster than One-Scan and BU-DP by 10-40 percent. Again, SS-Ordering is much faster and consumes less memory than the other three for its power of search-space pruning.

5.2.4 Experiment III: Number of Cells Explored (Fig. 3)

We report the number of cells explored by our algorithms in Experiments I-II. SS-Ordering explores much less cells than others thanks to its pruning techniques, especially for data sets that have more dimensions or more tuples (that is why it needs much less time than other algorithms). Both One-Scan and BU-DP (not in the figure) explore all the nonempty cells which contain at least one keyword, and Sorted-Scan explores a bit less than them but still much more and SS-Ordering in most cases.

We will report the running-time memory consumed by our algorithms for a fair comparison, since different algorithms consume different amounts of memory per cell.

5.2.5 Experiment IV: Parameter k (Fig. 4)

We use the real data set with 10 dimensions and set $\text{minsup} = 1$. The time and memory used in One-Scan and BU-DP nearly unchange because they always explore all nonempty cells in the text cube. For Sorted-Scan, the time and memory nearly unchange, because the early stop condition (lines 7-9 of Algorithm 2) tends to output a batch of cells at certain time. Again, SS-Ordering consumes much less time and memory than others. Sorted-Scan has an improvement of 20-40 percent over One-Scan and BU-DP in terms of the running time.

It can be also noticed that SS-Ordering uses more time and memory when k increases, because when more answers are required, more cells need to be explored in SS-Ordering. However, the increment is not too much, because, SS-Ordering algorithm follows a "shortcut" to the top-k cells, by first aggregating highly relevant cells into their parents.

5.2.6 Experiment V: Number of Keywords (Fig. 5)

We study how the number of keywords $|\text{q}|$ affects the performance. We set $k = 10$ and $\text{minsup} = 1$, using the 10-dimensional laptop review data set. Results are reported in Fig. 5. SS-Ordering is always faster and consumes less memory than the other three, as in other experiments.

Note that the time and memory used by Sorted-Scan and SS-Ordering vibrate irregularly. We explain this as follows:

When the number of keywords increases, there are two factors which affect the performance of the four algorithms: 1) at the beginning, more inverted indexes need to be scanned to compute the relevance of rows/cells; and 2) the variance of relevance scores of different cells could be larger, so it is easier for Sorted-Scan to satisfy the early

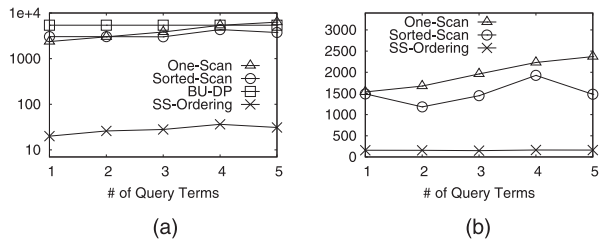


Fig. 5. Varying the number of query terms $|\text{q}|$. (a) Time (msec). (b) Memory (KB).

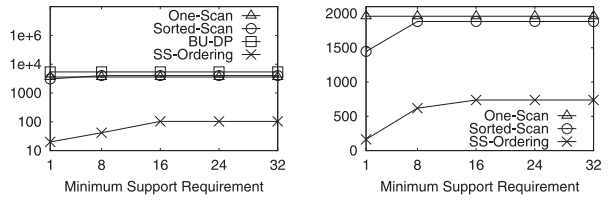


Fig. 6. Varying minimum support requirement minsup. (a) Time (msec). (b) Memory (KB).

stop condition, and for the SS-Ordering algorithm to reach the highly relevant cells (for it always seeks for a cell with high relevance score to aggregate it into its parent).

One-Scan and BU-DP are only affected by 1), so its running time and memory increase consistently as the number of keywords increases. Sorted-Scan and SS-Ordering are affected by both 1) and 2), so there is not a strong correlation between the efficiency of these two algorithms and the number of keywords. This feature could be regarded as an advantage of Sorted-Scan and SS-Ordering.

5.2.7 Experiment VI: Minimum Support Requirement (Fig. 6)

We vary the minimum support minsup from 1 to 32 in the 10-dimensional real data set. As minsup increases, the performances of One-Scan and BU-DP do not change much because they always search every cell (to compute its relevance score and to check whether its support $\geq \mathrm{minsup}$). Sorted-Scan and SS-Ordering consume more time and memory for larger minsup, because they will skip some cells with low support, and thus explore more cells before outputting k cells with support $\geq \mathrm{minsup}$. But still, SS-Ordering outperforms others a lot even when minsup is large, because it follows a shortcut to the relevant cells. Data are sparse in the 10-dimensional text cube, so the support 32 for a cell is already very high.

5.2.8 Experiment VII: Skewness of Dimension Cardinalities

We modify values on the 10 dimensions of the original real database to create another two synthetic databases skew and even. skew is obtained by modifying the dimension values in original s.t. five dimensions have cardinalities 3 and the other five have cardinalities 20. even has all dimension cardinalities equal to 5. Let $k = 80$ and $\mathrm{minsup} = 2$. Using the same set of queries, the average running time (in msec) is reported in Table 8. All the four algorithms perform (a slightly) worse in original than in the other two essentially because it contains the most nonempty cells. Skewness of dimension cardinalities does not play a very important role in affecting the performance.

5.3 Case Study

In this section, we verify the effectiveness of our model and algorithms by showing a few example queries and the meaningful retrieval results. Here, we use extended keyword queries with dimension-value constraints.

5.3.1 Case Study I: Finding Stylish Laptops

We want to find the brands, families, and/or colors of the most stylish laptops. One example query could be $(?, ?, *, *, *, *, *, *, *, ?$: {"stylish," "beautiful," "fashion"}), meaning that we only care about three dimensions (Laptop Brand, Laptop Family, Color) thus requiring all other dimensions to be aggregated. Some cells,[2] (dell, inspiron, pink), (toshiba, *, red), (*, macbook, *), (apple, *, *), (sony, vaio, red), and (sony, vaio, pink) have high relevance scores. Also, more than half of the top-20 cells have bright colors, e.g., pink, white, red, indicating that customers usually think laptops with bright colors are more stylish.

TABLE 8
Varying Skewness of Dimension Cardinalities

	One-Scan	Sorted-Scan	BU-DP	SS-Ordering
skew	2466	3467	4555	202
original	3623	3033	5422	212
even	2232	2902	3667	197

5.3.2 Case Study II: Finding Unreliable Laptops

Then, we are interested in finding laptops that are easy to crash, and we particularly care about the OS preinstalled on those laptops and their memory sizes. So, we require dimensions other than {Memory, OS} to be aggregated (*), and include keywords {"crash," "slow," "reinstall," "restore"} in the query. Among the top-20 cells, we found that cells (*, windows xp), (*, vista premium), and (*, vista home) have higher ranks than (*, linux). Another interesting fact is that the relevance score of "linux" is less sensitive w.r.t the main memory size than the "vista" series. For instance, both cells (512 MB, linux) and (1,024 MB, linux) have low relevance scores, but (1,024 MB, vista premium) has much higher relevance score than (3,072 MB, vista premium) does. This observation could be helpful to potential buyers, since they need to choose the memory size according to the operating system type (e.g., vista needs large memory).

6 Discussion

6.1 Handling Different Relevance Functions

Algorithm 4 (SS-Ordering) can be extended to handle other relevance scoring formulas. Note that we only utilize Lemmas 1-2 to prove the correctness of Algorithm 4 (as in Section 4.3.3). So, generally, Algorithm 4 can handle any relevance scoring formula that can be computed level by level (from $(i+1)$-dim cells to i-dim cells) and satisfies the property in Lemma 1, i.e., $\mathrm{rel}(\mathrm{q}, \mathrm{D}_1) \leq \mathrm{rel}(\mathrm{q}, \mathrm{D}_1 \cup \mathrm{D}_2) \leq \mathrm{rel}(\mathrm{q}, \mathrm{D}_2)$. One example of such relevance scores is the *weighted average relevance model* (as discussed at the end of Section 4.2). Another example is the *min/max relevance model* (i.e., the relevance of a cell w.r.t. a query is the min/max relevance of documents in this cell).

When the relevance scoring formula rel does *not* satisfy the property in Lemma 1, how to efficiently answer the top-k keyword query is still an open question.

6.2 About Support Requirement

As discussed in Section 1, the purpose of specifying the support requirement minsup is to help users filter out cells/objects that are *not* "popular" enough (i.e., containing too

2. We use a compact representation of cells to save the space in this section. As we care about three dimensions ($\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$), a cell C is represented by (v_1, v_2, v_3) iff $C[\mathcal{A}_i] = v_i$ for $i = 1, 2, 3$, and $C[\mathcal{A}] = *$ for any other dimension $\mathcal{A} \neq \mathcal{A}_i$.

few documents). However, when the user has little knowledge about the database (e.g., the size of this database), it is undesirable for her/him to provide such a parameter. We discuss two directions to handle this issue.

One way to address this issue is to integrate the support of a cell into its relevance score. But the resulting scoring formula may not satisfy the property in Lemma 1, and thus cannot be handled by Algorithm 4 (although still can be handled by less efficient Algorithms 1 and 3).

Another way is more direct. Consider several typical support requirements: for example, "normal" (support $\geq \text{minsup} = x_1$), "somewhat popular" (support $\geq \text{minsup} = x_2$), "popular" (support $\geq \text{minsup} = x_3$), and "very popular" (support $\geq \text{minsup} = x_4$). Parameters $x_1 < \cdots < x_4$ are specified by the database administrator, e.g., $x_i = 2^i$. For each incoming query (without specifying minsup), the system returns top-k answers for each of the four support requirements (i.e., running Algorithm 4 once for each value of minsup), so that the user can browse top-k cells in different levels (in terms of supports) at the same time.

7 Related Work

7.1 Keyword Search in RDBMs

Although based on different applications and motivations, keyword search in text cube is related to keyword search in RDBMSs, which has attracted a lot of attention recently [22], [23], [24]. Most previous studies on keyword search in RDBMSs model the RDB as a graph (tuples/tables as nodes, and foreign-key links as edges) and focus on finding minimal connected tuple trees that contain all the keywords. They can be categorized into two types. The first type uses SQL to find the connected trees [2], [7], [6], [10], [12], [13]. The second type materializes the RDB graph and proposes algorithms to enumerate (top-k) subtrees in the graph [3], [4], [8], [14], [9].

Different from these two types of works, two recent studies [11] and [5] find single-center subgraphs from the RDB graph, and multicenter induced subgraphs, respectively.

7.2 OLAP on Multidimensional Text Data

The text cube model is first proposed in [1]. Lin et al. [1] mainly focus on how to partially materialize inverted indexes and term frequency vectors in cells of text cube, and how to support OLAP queries (not keyword query) efficiently.

The topic cube model is proposed in [25]. Different from the text cube, the topic cube materializes the language model of the aggregated document in each cell. Efficient algorithms are proposed to compute this topic cube.

The techniques in [1] and [25] cannot be used directly to support keyword search, because the information materialized in text cube (term frequencies and inverted indexes) and in topic cube (language model) is query independent.

7.3 Faceted Search

Several faceted search systems have been developed to help users navigate the search results if some metadata about the results are available [26], [27], [28], [29], [30], [31], [32], [33]. An initial faceted search system [26] only uses the number of results in different facets to guide users. Ben-Yitzhak et al. [30] introduce a model that allows dynamic constraints to be applied on the facets. Roy et al. [29], [32] build a decision tree with minimum height on the data so that the efforts users spend in order to reach the relevant tuples will be minimized.

Faceted search is different from our search problem on both the goal and the methodology. Faceted search systems aim to enhance the retrieval of individual documents or tuples by providing users exploration guide, and need a sequence of user inputs. With an orthogonal purpose, our work aims to rank aggregated objects/cells in a cube using only a one-time keyword query, i.e., the input is a set of keywords, and the output is a ranked list of objects/cells. For example, in Section 5.3, to find a stylish laptop using keywords, our system directly tells the user which models are commented to be stylish (e.g., (toshiba, *, red)), while faceted search tells the user which dimension(s) she/he should drill down to find relevant tuples/documents.

7.4 Keyword-Based Search and OLAP in Data Cube

Zhou and Pei [34] study 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. "Minimal" here means there is no descendant of this cell containing more query terms. But, it is unnecessary that documents (cells) with more query terms are more relevant. And, Zhou and Pei [34] do not score or rank the answers. So when the number of returned answers is large (e.g., a thousand), it is difficult for the user to browse all the answers.

Another relevant work is keyword-driven analytical processing (KDAP) [35]. Motivated by a different scenario, KDAP supports interactive data exploration using keywords. Candidate subspaces are output to disambiguate the keyword terms. Efficiency is not a major concern in KDAP.

A keyword-driven OLAP system is proposed in [36] and [37]. It populates static and dynamic dimensions, allowing for roll-up and drill-down operations based on the content and the link structure of the dynamically selected document subset. Simitsis et al. [36], [37] still focus on the ranking of rows in the databases, instead of aggregated cells in our work.

7.5 Ranking Cube

Another related work is Ranking Cube [38], which addresses top-k queries with multidimensional selection using semimaterialization and semionline computation model. Different from our work, it does not support keyword queries. More importantly, it does not aim to find aggregations (cells) of rows. It can be applied for ranking base cells or rows (initialization part of our algorithms), which, however, is not the major computational bottleneck in our problem of ranking cells in all levels.

8 Conclusions and Future Work

We have proposed and studied the problem of keyword-based top-k search in text cube (i.e., multidimensional text data). Flexible query language and relevance scoring formula are developed. Four efficient algorithms are designed for this problem. Among them, the most efficient one, search-space ordering approach, uses different search-space pruning techniques for finding the top-k relevant cells by exploring

only a small portion of the whole text cube (when k is small). Extensive performance studies are conducted to verify the efficiency and effectiveness of these approaches.

For future work, 1) it is interesting to compare the effectiveness of different scoring formulas experimentally in different real data sets; 2) it is also interesting to design efficient algorithms for different forms of relevance scoring formula rel; and 3) while our approaches work in a bottom-up manner (starting from rows or base cells), it is interesting to study whether there is any efficient top-down approach (e.g., starting from the 0-dim cell).

ACKNOWLEDGMENTS

The work was supported in part by the NASA Aviation Safety Program, Integrated Vehicle Health Management Project by NASA grant NNX08AC35A, the US National Science Foundation (NSF) grant IS-09-05215, an HP Research grant, Microsoft research Women's Scholarship, and the Network Science Collaborative Technology Alliance Program (NS-CTA/INARC) of US Army Research Lab (ARL) under the contract number W911NF-09-2-0053. Any opinions, findings, and conclusions expressed here are those of the authors and do not necessarily reflect the views of the funding agencies. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. This paper is an extended version of [39]. The authors thank the anonymous reviewers for their numerous insights and suggestions that immensely improved the paper.

REFERENCES

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

[2] S. Agrawal, S. Chaudhuri, and G. Das, "Dbxplorer: A System for Keyword-Based Search over Relational Databases," *Proc. IEEE Int'l Conf. Data Eng. (ICDE),* pp. 5-16, 2002.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword Searching and Browsing in Databases Using Banks," *Proc. IEEE Int'l Conf. Data Eng. (ICDE),* pp. 431-440, 2002.

[4] B. Ding, J.X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin, "Finding Top-k Min-Cost Connected Trees in Databases," *Proc. IEEE Int'l Conf. Data Eng. (ICDE),* pp. 836-845, 2007.

[5] L. Qin, J.X. Yu, L. Chang, and Y. Tao, "Querying Communities in Relational Databases," *Proc. IEEE Int'l Conf. Data Eng. (ICDE),* pp. 724-735, 2009.

[6] V. Hristidis, L. Gravano, and Y. Papakonstantinou, "Efficient Ir-Style Keyword Search over Relational Databases," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 850-861, 2003.

[7] V. Hristidis and Y. Papakonstantinou, "Discover: Keyword Search in Relational Databases," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 670-681, 2002.

[8] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, "Bidirectional Expansion for Keyword Search on Graph Databases," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 505-516, 2005.

[9] K. Golenberg, B. Kimelfeld, and Y. Sagiv, "Keyword Proximity Search in Complex Data Graphs," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 927-940, 2008.

[10] H. He, H. Wang, J. Yang, and P.S. Yu, "Blinks: Ranked Keyword Searches on Graphs," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 305-316, 2007.

[11] G. Li, B.C. Ooi, J. Feng, J. Wang, and L. Zhou, "Ease: An Effective 3-in-1 Keyword Search Method for Unstructured, Semi-Structured and Structured Data," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 903-914, 2008.

[12] F. Liu, C.T. Yu, W. Meng, and A. Chowdhury, "Effective Keyword Search in Relational Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 563-574, 2006.

[13] Y. Luo, X. Lin, W. Wang, and X. Zhou, "Spark: Top-k Keyword Query in Relational Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 115-126, 2007.

[14] B. Kimelfeld and Y. Sagiv, "Finding and Approximating Top-k Answers in Keyword Proximity Search," *Proc. ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems (PODS),* pp. 173-182, 2006.

[15] B. Kimelfeld and Y. Sagiv, "Efficient Engines for Keyword Proximity Search," *Proc. Int'l Workshop Web and Databases (WebDB),* pp. 67-72, 2005.

[16] B. Kimelfeld and Y. Sagiv, "Efficiently Enumerating Results of Keyword Search," *Proc. Int'l Workshop Database Programming Languages (DBPL),* pp. 58-73, 2005.

[17] J.L. Elsas, J. Arguello, J. Callan, and J.G. Carbonell, "Retrieval and Feedback Models for Blog Feed Search," *Proc. Int'l Conf. Research and Development in Information Retrieval (SIGIR),* pp. 347-354, 2008.

[18] S.E. Robertson, S. Walker, and M. Hancock-Beaulieu, "Okapi at Trec-7: Automatic Ad Hoc, Filtering, VLC and Interactive," *Proc. Text REtrieval Conf. (TREC),* pp. 199-210, 1998.

[19] A. Singhal, J. Choi, D. Hindle, D. Lewis, and F. Pereira, "AT&T at TREC-7," *Proc. Text REtrieval Conf. (TREC),* pp. 239-252, 1998.

[20] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms,* second ed. The MIT Press, 2001.

[21] R. Fagin, A. Lotem, and M. Naor, "Optimal Aggregation Algorithms for Middleware," *J. Computer and System Sciences,* vol. 66, no. 4, pp. 614-656, 2003.

[22] S. Chaudhuri, R. Ramakrishnan, and G. Weikum, "Integrating DB and IR Technologies: What Is the Sound of One Hand Clapping?" *Proc. Conf. Innovative Data Systems Research (CIDR),* pp. 1-12, 2005.

[23] S. Amer-Yahia, P. Case, T. Rölleke, J. Shanmugasundaram, and G. Weikum, "Report on the DB/IR Panel at SIGMOD 2005," *ACM SIGMOD Record,* vol. 34, no. 4, pp. 71-74, 2005.

[24] G. Weikum, "DB&IR: Both Sides Now," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 25-30, 2007.

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

[26] M.A. Hearst, A. Elliott, J. English, R.R. Sinha, K. Swearingen, and K.-P. Yee, "Finding the Flow in Web Site Search," *Comm. ACM,* vol. 45, no. 9, pp. 42-49, 2002.

[27] V. Sinha and D.R. Karger, "Magnet: Supporting Navigation in Semistructured Data Environments," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 97-106, 2005.

[28] D. Dash, J. Rao, N. Megiddo, A. Ailamaki, and G.M. Lohman, "Dynamic Faceted Search for Discovery-Driven Analysis," *Proc. Int'l Conf. Information and Knowledge Management (CIKM),* pp. 3-12, 2008.

[29] S.B. Roy, H. Wang, G. Das, U. Nambiar, and M.K. Mohania, "Minimum-Effort Driven Dynamic Faceted Search in Structured Databases," *Proc. Int'l Conf. Information and Knowledge Management (CIKM),* pp. 13-22, 2008.

[30] O. Ben-Yitzhak, N. Golbandi, N. Har'El, R. Lempel, A. Neumann, S. Ofek-Koifman, D. Sheinwald, E.J. Shekita, B. Sznajder, and S. Yogev, "Beyond Basic Faceted Search," *Proc. Int'l Conf. Web Search and Web Data Mining (WSDM),* pp. 33-44, 2008.

[31] J. Koren, Y. Zhang, and X. Liu, "Personalized Interactive Faceted Search," *Proc. Int'l World Wide Web Conf. (WWW),* pp. 477-486, 2008.

[32] S.B. Roy, H. Wang, U. Nambiar, G. Das, and M.K. Mohania, "Dynacet: Building Dynamic Faceted Search Systems over Databases," *Proc. IEEE Int'l Conf. Data Eng. (ICDE),* pp. 1463-1466, 2009.

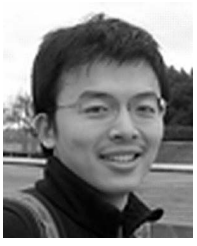
[33] C. Li, N. Yan, S.B. Roy, L. Lisham, and G. Das, "Facetedpedia: Dynamic Generation of Query-Dependent Faceted Interfaces for Wikipedia," *Proc. Int'l World Wide Web Conf. (WWW),* pp. 651-660, 2010.

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

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

[36] A. Simitsis, A. Baid, Y. Sismanis, and B. Reinwald, "Multidimensional Content Exploration," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 660-671, 2008.
[37] A. Baid, A. Balmin, H. Hwang, E. Nijkamp, J. Rao, B. Reinwald, A. Simitsis, Y. Sismanis, and F. van Ham, "Dbpubs: Multidimensional Exploration of Database Publications," *Proc. VLDB Endowment*, vol. 1, no. 2, pp. 1456-1459, 2008.
[38] D. Xin, J. Han, H. Cheng, and X. Li, "Answering Top-k Queries with Multi-Dimensional Selections: The Ranking Cube Approach," *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, pp. 463-475, 2006.
[39] B. Ding, B. Zhao, C.X. Lin, J. Han, and C. Zhai, "Topcells: Keyword-Based Search of Top-k Aggregated Documents in Text Cube," *Proc. IEEE Int'l Conf. Data Eng. (ICDE)*, pp. 381-384, 2010.

Bolin Ding received the BS degree in maths and applied mathematics from Renmin University of China, in 2005, and the MPhil degree in system engineering from the Chinese University of Hong Kong in 2007. He is currently working toward the PhD degree in the Department of Computer Science, University of Illinois at Urbana-Champaign. His research interests include the mining and analysis of structured data, efficient and effective searching and indexing, and the design of randomized algorithms and approximation algorithms with applications in databases.



Bo Zhao received the BS degree in computer science from Fudan University, Shanghai, in 2007. He is currently working toward the PhD degree in the Department of Computer Science, University of Illinois at Urbana-Champaign. His research interests include data mining, text mining, information retrieval, and social networks.

Cindy Xide Lin received the BS degree from Tsinghua University in 2007 and the master's degree from the University of Illinois at Urbana-Champaign in 2009, both in computer science. She is currently working toward the PhD degree in the Department of Computer Science, University of Illinois at Urbana-Champaign. Her research interests include data mining and information retrieval, especially topics related to web mining and social network. She has published more than 10 papers in international conferences, including SIGKDD, VLDB, WWW, ICDE, ICDM, EMNLP, CIKM, PAKDD, and CIDU.

Jiawei Han is a professor of computer science at the University of Illinois. He has served on the program committees of the major international conferences in the fields of data mining and database systems, and also served or is serving on the editorial boards for *Data Mining and Knowledge Discovery*, *IEEE Transactions on Knowledge and Data Engineering*, *Journal of Computer Science and Technology*, and *Journal of Intelligent Information Systems*. He is the founding editor-in-chief of *ACM Transactions on Knowledge Discovery from Data* (TKDD). He has received IBM Faculty Awards, HP Innovation Awards, ACM SIGKDD Innovation Award (2004), IEEE Computer Society Technical Achievement Award (2005), and IEEE W. Wallace McDowell Award (2009). He is currently the director of Information Network Academic Research Center (INARC) supported by the Network Science-Collaborative Technology Alliance (NS-CTA) program of US Army Research Lab. His book *Data Mining: Concepts and Techniques* (Morgan Kaufmann) has been used worldwide as a textbook. He is a fellow of the ACM and the IEEE.

Chengxiang Zhai received the PhD degree in computer science from Nanjing University in 1990, and the PhD degree in language and information technologies from Carnegie Mellon University in 2002. He is an associate professor of computer science at the University of Illinois at Urbana-Champaign, where he also holds a joint appointment at the Institute for Genomic Biology, Statistics, and the Graduate School of Library and Information Science. He worked at Clairvoyance Corp. as a research scientist and a senior research scientist from 1997 to 2000. His research interests include information retrieval, text mining, natural language processing, machine learning, and bioinformatics. He is an associate editor of the *ACM Transactions on Information Systems*, and *Information Processing and Management*, and serves on the editorial board of *Information Retrieval Journal*. He is a program cochair of ACM CIKM 2004, NAACL HLT 2007, and ACM SIGIR 2009. He is an ACM Distinguished Scientist, and received the 2004 Presidential Early Career Award for Scientists and Engineers (PECASE), the ACM SIGIR 2004 Best Paper Award, an Alfred P. Sloan Research Fellowship in 2008, and an IBM Faculty Award in 2009.

Ashok Srivastava is the principal investigator for the Integrated Vehicle Health Management research project at NASA. His current research focuses on the development of data mining algorithms for anomaly detection in massive data streams, kernel methods in machine learning, and text mining algorithms. He is also the leader of the Intelligent Data Understanding group at NASA Ames Research Center. The group performs research and development of advanced machine learning and data mining algorithms in support of NASA missions. He performs data mining research in a number of areas in aviation safety and application domains such as earth sciences to study global climate processes and astrophysics to help characterize the large-scale structure of the universe. He is the author of many research articles in data mining, machine learning, and text mining, and has edited a book on *Text Mining: Classification, Clustering, and Applications* (with Mehran Sahami, 2009). He is currently editing two more books: *Advances in Machine Learning and Data Mining for Astronomy* (with Kamal Ali, Michael Way, and Jeff Scargle) and *Data Mining in Systems Health Management* (with Jiawei Han). He has given seminars at numerous international conferences. He has won numerous awards including the IEEE Computer Society Technical Achievement Award for "pioneering work in Intelligent Information Systems," the NASA Exceptional Achievement Medal for contributions to state-of-the-art data mining and analysis, the NASA Distinguished Performance Award, several NASA Group Achievement Awards, the IBM Golden Circle Award, and the Department of Education Merit Fellowship. He is a senior member of the IEEE.

Nikunj C. Oza received the BS degree in mathematics with computer science from the Massachusetts Institute of Technology (MIT) in 1994. He received the MS degree in 1998 and the PhD degree in 2001 in computer science from the University of California at Berkeley. He then joined NASA Ames Research Center as a research scientist, a member of the Intelligent Data Understanding group, and the lead of a team applying data mining to the problem of health management for aerospace vehicles. His 40+ research papers represent his research interests which include machine learning (especially ensemble learning and online learning), data mining, fault detection, combinations of machine learning with other areas of Artificial Intelligence, and their applications to Aeronautics and Earth Science. He was the cochair of the 2005 International Workshop on Multiple Classifier Systems (MCS), program committee member of several MCS and SIAM Data Mining (SDM) conferences, and serves as an associate editor of the journal *Information Fusion* (Elsevier). He also received the 2007 Arch T. Colwell Award in 2007 for coauthoring one of the five most innovative technical papers selected out of more than 3,300 SAE technical papers published in 2005. He is a member of the IEEE.