iVA-File: Efficiently Indexing Sparse Wide Tables in Community Systems

Boduo Li [#], Mei Hui [*], Jianzhong Li [#], Hong Gao [#]

[#] *School of Computer Science and Technology, Harbin Institute of Technology, China*
[*] *School of Computing, National University of Singapore, Singapore*
{boduo,lijzh,honggao}@hit.edu.cn hui_mei@comp.nus.edu.sg

Abstract— In community web management systems (CWMS), storage structures inspired by universal tables are being used increasingly to manage sparse datasets. Such a sparse wide table (SWT) typically embodies thousands of attributes, with many of them being undefined in each tuple, and low-dimensional structured similarity search on a combination of numerical and text attributes is a common operation. However, many properties of such wide tables and their associated Web 2.0 services render most multi-dimensional indexing structures irrelevant. Recent studies in this area have mainly focused on improving the storage efficiency and efficient deployment of inverted indices; so far no new index has been proposed for indexing SWTs. The inverted index is fast for scanning but not efficient in reducing random accesses to the data file as it captures little information about the content of attribute values. In this paper, we propose the iVA-file that works on the basis of approximate contents and keeps scanning efficiency within a bounded range. We introduce the nG-signature to approximately represent data strings and improve the existing approximate vectors for numerical values. We also propose an efficient query processing strategy for the iVA-file, which is different from strategies used for existing scan-based indices. To enable the use of different metrics of distance between a query and a tuple that may vary from application to application, the iVA-file has been designed to be metric-oblivious and to provide efficient filter-and-refine search based on any rational metric. Extensive experiments on real datasets show that the iVA-file outperforms existing proposals in query efficiency significantly, at the same time, keeps a good update speed.

I. INTRODUCTION

We have witnessed the increasing popularity of Web 2.0 systems such as blogs [1], Wikipedia [2] and Flickr [3], where users contribute content and value-add to the system. These systems are popular as they allow users to display their creativity and knowledge, take ownership of the content, and obtain shared information from the community. A Web 2.0 system serves as a platform for users of a community to interact and collaborate with each other. Such community web management systems (CWMS) have been successfully applied in an extensive range of communities because of their effectiveness in collecting and organizing the wisdom of crowds. CWMSs drive the design of new storage platforms that impose requirements unlike those of conventional database systems designed to support relational query processing.

A. Data

In many CWMS databases, the dataset is sparse and comprises a good mix of alphanumeric and string-based attributes.
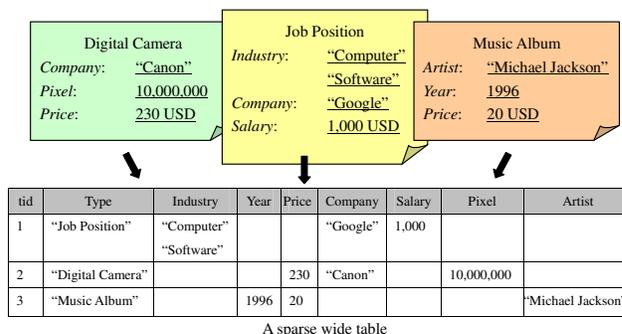

Fig. 1. Users submit freely defined meta data to the sparse wide table.

Web-based e-commerce systems store information of a wide-ranging set of products, and due to different product specifications, the dataset can be very sparse. For example, in the dataset of the CNET e-commerce system examined by Chu et al. [4], 233,304 products are described with an average 11 attributes for each product although there are 2,984 attributes in total. Similarly, most community-based data publishing systems, such as Google Base [5], allow users to define their own meta data and store as much information as they wish, as shown in Fig.1. As a result, the dataset is described with a very large and diverse set of attributes. We downloaded a subset of the Google Base data [5], where $779,019$ items define $1,147$ attributes and the average number of attributes defined in each item is 16. To facilitate fast and easy storage and efficient retrieval, the wide table storage structure has been proposed in [6], [4], [7], [8]. The wide table can be physically implemented as vertical tables and file-based storage [4], [7]. In this paper, we focus on the indexing issues of sparse datasets in CWMSs that are stored as wide tables. For ease of discussion, we shall refer to them as sparse wide tables (SWT).

B. Query

Users describe their searching intention in CWMS by providing the most expected values on some attributes. One example of such structured queries is shown in Fig.2. CWMS should rank the tuples in SWT based on their relevance to the query, and usually the top-k tuples are returned to users. In CWMSs, strings are typically short, and typos are very common because of the participation of large groups of people. For instance, "Cannon" in tuple 8 on attribute Company in Fig.2 should be "Canon". To facilitate the ranking, edit distance [9], [10], [11], a widely used typo-tolerant metric,

tid	Type	Year	Price	Company	Salary
	...				
3	"Digital Camera"		240	"Sony"	
	...				
8	"Digital Camera"		230	"Cannon"	
	...				

Type: "Digital Camera"
Company: "Canon"
Price: 200 USD

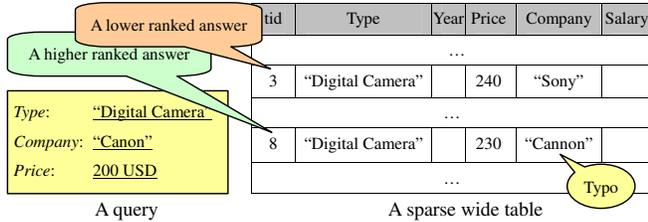Typo

A query   A sparse wide table

Fig. 2.   A structured similarity query in CWMSs.

is adopted to evaluate the similarity between two strings. Recent research [12] on relevance-based ranking in text-rich relational databases argues that unstructured queries, the popular querying mode in IR engines, are preferred for the reason that structured queries require users to have knowledge of the underlying database schema and complex query interface. But structured queries are popular in CWMSs, such as Google Base, for three reasons. First, unlike typical relational multi-table datasets [13], the SWT, which is the only table maintained for each application, has no trouble of database schema. Second, many easy-to-use APIs are provided by CWMSs for semi-professionals to construct an intermediate level between users and the CWMS. So the query interface is usually transparent to users, who can submit queries through specialized web pages that transform users' original queries into structured ones. Third, the datasets in CWMS contain both numerical and text values, which present problems to text-centric IR-based query processing.

C. Contributions of This Paper

We focus on the indexing issues of structured similarity search over the SWT in this paper. Our contribution is novel in two ways.

First, recent studies on SWTs, such as the interpreted schema [6], [4], [7], mainly focus on optimizing the storage scheme of datasets. To our knowledge, no new indexing techniques have been proposed, and so far only the inverted index has been evaluated for SWTs in [7]. For each attribute, a list of identifiers of the tuples that have definition on this attribute is maintained, and only several related lists are scanned for a query in order to filter tuples that are impossible to be a result. Such partial scan results in dramatically low I/O cost of accessing the index. However, this technique captures no information with regard to the values and may therefore be inefficient in terms of filtering.

Second, the existing multi-dimensional indices that have been designed for multi-dimensional and spatial databases have not shown to be suitable and efficient for SWTs due to the following differences between CWMS and traditional applications: 1) The scale of the SWT is much larger, and the dataset is much sparser. 2) The datasets of traditional applications are static for scientific statistics. In contrast, CWMSs have been designed to provide free-and-easy data publishing and sharing to facilitate the collaboration between users. The datasets are more dynamic as the number of users is very large and they submit and modify the information in an ad hoc manner. 3) In traditional environments, dimensionality is

fixed and a query embodies a constraint on every attribute. On the contrary, dynamic datasets result in a fluctuating number of attributes, and the SWT is high-dimensional while the query in CWMSs is low-dimensional since each tuple is described by only a few attributes. 4) The existing similarity search techniques for relational databases have been designed to handle either text or numerical attributes while we handle both of them in SWTs.[1]

In this paper, we propose the iVA-file, an inverted vector approximation file, as an indexing structure that stores approximation vectors as the rough representation of data values, and supports efficient partial scan and similarity search. We introduce the nG-signature as the approximation vector encoding scheme of strings which guarantees no false negative. We also improve the existing encoding scheme of numerical values. The iVA-file is a content-conscious and scan-efficient index that overcomes the weaknesses of the inverted index in [7]. To enable the use of different rational metrics of distance between a query and a tuple that may vary from application to application, the iVA-file has been designed to be metric-oblivious and to provide efficient filter-and-refine search.

Our contributions of this paper are summarized as follows:

- To the best of our knowledge, the iVA-file is the first content-conscious indexing mechanism designed to support structured similarity queries over SWTs prevalent in Web 2.0 applications.
- We propose the nG-Signature to encode strings and improve the code for numerical values so that the iVA-file can efficiently support similarity metrics on a mix of text and numerical attributes. Parameters of our encoding scheme can be tuned for the trade-off between the I/O cost of scanning the index and the cost of random accesses on the data file.
- We design a novel query processing strategy for queries that are defined on a mixture of text and numerical attributes, which is suitable for any rational similarity metrics, and guarantees no false negative.
- We have conducted extensive experiments using real CWMS datasets, and the results show that the iVA-file is much more efficient than the existing methods.

The rest of the paper is organized as follows. We discuss related work in Sec. II. We present the iVA-file in Sec. III. Sec. IV introduces the query processing with the iVA-file and the update of it. The performance study of the iVA-file is presented in Sec. V. We conclude the paper in Sec. VI.

II. RELATED WORKS

A. Existing Works on Sparse Wide Tables

The conventional storage of relational tables is based on the horizontal storage scheme, in which the position of each value can be obtained through the calculation of its attribute id and tuple id. However, for sparse wide tables (SWT), a horizontal storage scheme is not efficient due to the large amount of

---

[1]See detailed discussion on the suitability of existing multi-dimensional indices in Sec.II-B.

undefined values (*ndf*). The research on sparse tables has a long history [14], [15]. A few approaches, such as DSM [16], vertical schema [17], and interpreted schema [6], have been proposed to alleviate the problem of *ndf*s and the number of attributes. Beckmann et al. evaluated these approaches [6], and concluded that the best option is to store the data horizontally in an interpreted format. These works merely focus on enhancing the query efficiency through diverse organization of data storage. [4] proposes a clustering method to find the hidden schema in the wide sparse table, which not only promotes the efficiency of query processing but also assists users in choosing appropriate attributes when building structured queries over thousands of attributes. Building sparse B-tree index on all attributes is recommended in [4], too. But it is difficult to apply to multi-dimensional similarity queries. As of today, the only index that has been evaluated for indexing SWTs is a straightforward application of inverted indices over the attributes [7]. The indices are able to speed up the selection of tuples with given attributes. They however only distinguish *ndf* and non-*ndf* values, but do not take the contents of the attributes into consideration.

The SWT in our context is different from the Universal Relation [18], which has also been discussed in [4], [7]. Succinctly, the Universal Relation is a wide virtual schema that covers all physical tables whereas the SWT is a physically stored table that contains a large number of attributes. The main challenge of the Universal Relation is how to translate and run queries based on a virtual schema, whereas our challenge here is how to efficiently store data and execute search operations.

B. Traditional Multi-dimensional Indices

A cursory examination of the problem may suggest that multi- and high-dimensional indexing could resolve the indexing problem of SWTs. However, due to the presence of a proportionally large number of undefined attributes in each tuple, hierarchical indexing structures that have been designed for full-dimensional indexing or that are based on metric space [19] such as the iDistance [20] are not suitable. Further, most high-dimensional indices that are based on data and space partitioning are not efficient when the number of dimensions is very high [21], [22] due to the curse of dimensionality. Weber et al. [23] provided a detailed analysis and showed that as the number of dimensions becomes too large, a simple sequential scan of the data file would outperform the existing approaches. Consequently, they proposed the VA-file, which is a smaller approximation to the data file, for fast sequential scan to quickly filter out as many negatives as possible. Subsequently, the data file is accessed to check for the remaining tuples. The VA-file encoding method was later extended to handle *ndf*s in [24]. For the fact that the distance between data points are indistinguishable in high-dimensional spaces, the VA-file is likely to suffer the same scalability problem [22].

These indices have been proposed for the data that assume full-dimensional of the dataset even when the *ndf* values are present, and with numerical values as domain. The CWMS

characteristics invalidate any design based on such assumptions. Further, the VA-file is not efficient for the SWT as the data file that is often in some compact form [6], [4], [7] could be even smaller than the VA-file. In addition, it remains unknown how an unlimited-length string could be mapped to a meaningful vector for the VA-file. Column storage of the VA-file is proposed in [25] for multimedia retrieval, which is quite different from the sparse text-rich data here.

Another multi-dimensional index based on sequential scan is the bitmap index [26], [27], [28]. As a bit-wise index approach, the bitmap index is efficiently supported by hardware at the cost of inefficient update performance. Compression techniques [27], [28] have been proposed to manage the size of the index. The bitmap index is an efficient way to process complex select queries for read-mostly or append-only data, and is not known to be able to support similarity queries efficiently. Further, it does not support text data although many encoding schemes have been proposed [26], [29].

C. Text Indices

The inverted index and the signature file [30], [31] are two text indices that are well studied and widely used in large text databases and information retrieval for keyword-based search. Both of the two indices are used for a single text attribute where the text records are long documents. Other works on keyword search in relational databases [32], [33] treat a record as a text document ignoring the attributes.

Many non-keyword similarity measures of strings have been proposed [34], among which edit distance could be most widely adopted [35], [9], [10], [11]. One method to estimate the edit distance is to use n-grams. Gravano et al. put forward the edit distance estimation based on n-gram set to filter tuples and prevent false negatives at the same time [9]. The inverted index on n-grams [11] is designed for searching strings on a single attribute that is within an edit distance threshold to a query string. This method is also extended to variable-length-grams [36]. A multi-dimensional index for unlimited-length strings was proposed in [37] which adopts a tree-like structure and maps a string to a decimal number. However, the index focuses on exact or prefix string match within a low-dimensional space.

III. IVA-FILE

A. Problem Description

The single wide table aims to provide fast insertion of tuples with a subset attributes defined out of a much bigger set of diverse attributes and fast retrieval that does not involve expensive join operations. Suppose that $\mathcal{A}$ is the set of all attributes of such a large table. There are two types of attributes: text attributes and numerical attributes. Let $\mathcal{T}$ denote the set of all tuples in the table, and $|\mathcal{T}|$ denote the number of tuples. Logically, each cell in the table determined by a tuple T and an attribute A has a value, denoted by $v(T, A)$, where $T \in \mathcal{T}$ and $A \in \mathcal{A}$. If A is not defined in T, we say that $v(T, A)$ has a special value ndf. Otherwise, if A is a numerical attribute, $v(T, A)$ is a numerical number, and if A

is a text attribute, $v(T, A)$ is a non-empty set of finite-length strings. A real example of a text value with multiple strings is the value of tuple 1 on attribute `Industry` in the table shown in Fig.1.

In this paper, we consider the top-k structured similarity query. A query is defined with values on a subset of the attributes in the table. If Q is a query, $v(Q, A)$ represents the value in Q on attribute A. If A is not defined in Q, $v(Q, A)$ is ndf. Otherwise, if A is a numerical attribute, $v(Q, A)$ is a numerical number, and if A is a text value, $v(Q, A)$ is a string. Suppose $D(T, Q)$, about which we will give a detailed introduction later, is a distance function that measures the similarity between tuple T and query Q. Assume that all tuples $T_0, T_1, \cdots, T_{|\mathcal{T}|-1}$ in $\mathcal{T}$ are sorted by $D(T_i, Q)$ in increasing order. Note that all tuples with the same distance are in random order. The result of the query Q is:

$$\{T_0, T_1, \cdots, T_{K-1}\}$$

where $K = \min\{k, |\mathcal{T}|\}$.

The difference between two strings is evaluated by edit distance, which is "the minimum number of edit operations (i.e., insertions, deletions, and substitutions) of single characters needed to transform the first string into the second." [9] Let $ed(s_1, s_2)$ denote the edit distance between two strings s_1 and s_2. The difference between a query string in query Q on a text attribute A $(v(Q, A) \neq ndf)$ and the text value in tuple T on A is denoted by $d[A](T, Q)$. If $v(T, A) = ndf$, $d[A](T, Q)$ is a predefined constant. Otherwise, $d[A](T, Q)$ is the smallest edit distance between the query string and the data strings in $v(T, A)$. That is

$$d[A](T, Q) = \min\{ed(s, v(Q, A)) : s \in v(T, A)\}.$$

The difference between a query value in query Q on a numerical attribute A $(v(Q, A) \neq ndf)$ and the value in tuple T on A is also denoted by $d[A](T, Q)$, where $d[A](T, Q)$ is a predefined constant if $v(T, A) = ndf$, or $|v(Q, A) - v(T, A)|$ if $v(T, A) \neq ndf$.

The similarity distance $D(T, Q)$ is a function of all $\lambda_i \cdot d[A_i](T, Q)$ where $v(Q, A_i) \neq ndf$. λ_i $(\lambda_i > 0)$ is the importance weight of A_i. Let $A_1, A_2, ..., A_q$ denote all defined attributes in Q. If we use d_i instead of $d[A_i](T, Q)$ for short, $D(T, Q)$ can be written as

$$D(T, Q) = f(\lambda_1 \cdot d_1, \lambda_2 \cdot d_2, ..., \lambda_q \cdot d_q).$$

Function f determines the similarity metric. In this paper, we assume that f complies with the monotonous property described as the following property.

Property 3.1: [Monotonous] If two tuples T_1 and T_2 satisfy that for each attribute A_i that is defined in a query Q, $d[A_i](T_1, Q) \geq d[A_i](T_2, Q)$, then $D(T_1, Q) \geq D(T_2, Q)$. ■

The monotonous property, intuitively, tells that if T_1 is no closer to Q than T_2 is on all attributes that users care, T_1 is no closer to Q than T_2 is for the similarity distance. This is a natural property for any rational similarity metric f. The index proposed in this paper guarantees accurate answers for

any similarity metric that obeys the monotonous property. We test the efficiency of our index approach for some commonly used similarity metric and attribute weight settings through experiments over real datasets.

We design a new index method, the inverted vector approximation file (iVA-file), which guarantees accurate result. The iVA-file holds vectors that approximately represent numerical values or strings and organizes these vectors to support efficient access and filter-and-refine process. So the first sub-problem is the encoding scheme to map a string (Sec. III-B) or a numerical value (Sec. III-C) to an approximation vector and support filtering with no false negatives. The second sub-problem is to organize the vectors in an efficient structure to: (a) allow partial scan, (b) minimize the size of the index, and (c) ensure correct mapping between a vector and a value in the table (Sec. III-D).

B. Encoding of Strings

We propose the n-gram signature (nG-signature) to encode any single string. Given a query string s_q and the nG-signature $c(s_d)$ of a data string s_d, we should estimate the edit distance between s_q and s_d. Let $est(s_q, c(s_d))$ denote the estimated edit distance. To avoid false negatives caused by the filtering process, it is clear that $est(s_q, c(s_d))$ is required to satisfy $est(s_q, c(s_d)) \leq ed(s_q, s_d)$, according to the definition of $d[A](T, Q)$ on text attributes and the monotonous property of f. We will show how to filter tuples with this estimated distance in Sec. IV-A. We confine ourselves to introducing the encoding scheme and the calculation of $est(s_q, c(s_d))$ here.

1) nG-Signature:

n-gram is widely used for estimating the edit distance between two strings [35], [9], [10]. Suppose '#' and '$' are two symbols out of the text alphabet. To obtain the n-grams of a string s, we first extend s to s' by adding $n - 1$ '#' as a prefix and $n - 1$ '$' as a suffix to s. Any sequence of n consecutive characters in s' is an n-gram of s [10].

Example 3.1: [n-Gram] To obtain all the 3-grams of "yes", first extend it to "##yes$$". So "##y", "#ye", "yes", "es$" and "s$$" are the 3-grams of "yes". ■

The nG-signature $c(s)$ of a string s is a bit vector that consists of two parts. The lower bits $c_L(s)$ record the length of s. The higher bits, denoted by $c_H[l, t](s)$ $(0 < t < l)$, approximately record the n-grams of s. To encode $c_H[l, t](s)$ from s, we need a hash function $h[l, t](\omega)$ to hash an n-gram ω to an l-bit vector, which always contains t bits of 1 and $l - t$ bits of 0. $c_H[l, t](s)$ is the logic OR of all $h[l, t](\omega_i)$, where ω_i is an n-gram of s.

Example 3.2: [nG-Signature] Suppose a string is "ok". The 2-grams are "#o", "ok" and "k$". $l = 8$, $t = 2$ and use 4 bits to record the string length. The process of encoding the c("ok") is shown in Fig. 3. ■

2) Edit Distance Estimation with nG-Signature:

The method of calculating $est(s_q, c(s_d))$ is the extension of an existing estimation method. Let $g(s)$ denote the n-gram set of string s. For the purpose of estimating edit distance, the

$h[8,2]$("#o")	11000000
$h[8,2]$("ok")	01000100
$h[8,2]$("k\$") OR	00010001
$c_H[8,2]$("ok")	11010101
c_L("ok") +	0010
c("ok")	11010101 0010

Fig. 3. An example of generating a string's nG-signature

AND	$c_H[8,2]$("ok") = 11010101	
$h[8,2]$("#o") = 11000000	11000000	Hit
$h[8,2]$("oh") = 00001001	00000001	Not hit
$h[8,2]$("h\$") = 00000101	00000101	Hit
$\|hg$("oh",c("ok"))$\|$		2

Fig. 4. An example of estimating edit distance with nG-signature

same n-grams starting at different positions in s should not be merged in the n-gram set [9]. So we define $g(s)$ as a set of pairs in the form of (a, ω), where ω is an n-gram of s and a counts the appearance of ω in s. The size of a set Ω of such pairs is defined as:

$$|\Omega| = \sum_{(a_i, \omega_i) \in \Omega} a_i$$

Example 3.3: [n-**Gram Set**] The 2-gram set of string "www" is $\{(1,"#w"), (2,"ww"), (1,"w\$")\}$. The size of it is 4. ∎

The *common n-gram set* of two strings s_1 and s_2, denoted by $cg(s_1, s_2)$, is

$$\{(a, \omega) : \exists (a_1, \omega) \in g(s_1), (a_2, \omega) \in g(s_2), a = \min\{a_1, a_2\}\}.$$

Intuitively, $cg(s_1, s_2)$ is the intersection of $g(s_1)$ and $g(s_2)$. The notation such as $|s|$ represents the length of string s measured by the number of characters. Given a query string s_q and a data string s_d, let $|cg(s_q, s_d)|$ denote the size of their common n-gram set. Define the symbol $est'(s_q, s_d)$ as:

$$est'(s_q, s_d) = \frac{\max\{|s_q|, |s_d|\} - |cg(s_q, s_d)| - 1}{n} + 1 \quad (1)$$

According to [9]:

$$est'(s_q, s_d) \leq ed(s_q, s_d) \quad (2)$$

[9] uses $est'(s_q, s_d)$ to estimate edit distance and shows that it is efficient in filtering tuples. Moreover, the filtering causes no false negatives as the estimation is never larger than the actual edit distance.

Within the context of filtering a tuple with a query string s_q and the nG-Signature $c(s_d)$ of a data string s_d, we can easily obtain $\max\{|s_q|, |s_d|\}$ by the lower bits $c_L(s_d)$, but we have no way of calculating $|cg(s_q, s_d)|$ accurately. Therefore, we propose the concept of hit gram set to estimate $|cg(s_q, s_d)|$ based on the higher bits $c_H[l, t](s_d)$ in the signature.

Definition 3.1: [**Hit**] If ω is an n-gram of query string s_q, ω is a hit in the nG-signature of data string s_d if and only if:

$$h[l, t](\omega) \times c_H[l, t](s_d) = h[l, t](\omega)$$

where $\times$ denotes the operator of logical AND that joins two bit-strings. ∎

Obtaining the following property of hit becomes obvious:

Property 3.2: [**Self Hit**] If ω is an n-gram of a data string s_d, ω is a hit in the nG-signature of s_d. ∎

The self hit property tells that any n-gram in the common n-gram set of s_d and s_q must be a hit in the nG-signature of s_d. But an n-gram of s_q which is not an n-gram of s_d may also be a hit in the nG-signature of s_d. So, we give the following definition.

Definition 3.2: [**False Hit**] We call ω a false hit, if and only if, ω is a hit in the nG-signature of s_d and ω is not an n-gram of s_d. ∎

We define the hit gram set $hg(s_q, c(s_d))$ as follows:

Definition 3.3: [**Hit Gram Set**] $hg(s_q, c(s_d))$ is:

$$\{(a, \omega) : (a, \omega) \in g(s_q) \text{ and } \omega \text{ is a hit in } c(s_d)\}$$

where $c(s_d)$ is the nG-signature of s_d. ∎

We propose estimating $|cg(s_q, s_d)|$ in Equation 1 with $|hg(s_q, c(s_d))|$. So the edit distance estimation function for the iVA-file is:

$$est(s_q, c(s_d)) = \frac{\max\{|s_q|, |s_d|\} - |hg(s_q, c(s_d))| - 1}{n} + 1 \quad (3)$$

Example 3.4: [**Edit Distance Estimation**] Suppose that the data string is "ok" and the query string is "oh". As in Example 3.2, $l = 8$, $s = 2$, and we adopt the same hash function. So the higher bits of the nG-signature of "ok" is 11010101. The 2-grams of "oh" are "#o", "oh" and "h\$". The process of calculating $|hg(s_q, c(s_d))|$ is shown in Fig. 4. According to Equation 3, the edit distance is estimated as 0.5. We can safely loosen it to 1. ∎

We prove that our estimation causes no false negatives by the following proposition.

Proposition 3.3: Given a query string s_q and a data string s_d,

$$est(s_q, c(s_d)) \leq ed(s_q, s_d)$$

which guarantees no false negatives.

Proof: According to the definition of $cg(s_q, s_d)$, $\forall (a_i, \omega_i) \in cg(s_q, s_d)$, $\exists (a_i', \omega_i) \in g(s_q)$ such that $a_i \leq a_i'$, and $\exists (a_i'', \omega_i) \in g(s_d)$. Since ω_i is an n-gram of s_d, according to Property 3.2, ω_i is a hit in $c(s_d)$. In agreement with the definition of $hg(s_q, c(s_d))$, $(a_i', \omega_i) \in hg(s_q, c(s_d))$. Thus:

$$\sum_{(a_i, \omega_i) \in cg(s_q, s_d)} a_i \leq \sum_{(a_i, \omega_i) \in cg(s_q, s_d)} a_i' \leq \sum_{(a_j, \omega_j) \in hg(s_q, c(s_d))} a_j$$

That is:

$$|cg(s_q, s_d)| \leq |hg(s_q, c(s_d))|$$

By Equation 1 and 3, we have:

$$est(s_q, c(s_d)) \leq est'(s_q, s_d)$$

According to Equation 2, we get:

$$est(s_q, c(s_d)) \le ed(s_q, s_d)$$ ∎

3) nG-Signature Parameters:

Proposition 3.3 guarantees that no false negatives occur while filtering with nG-signatures. But we expect $est(s_q, c(s_d))$ to be as close as possible to $est'(s_q, s_d)$, which reflects the accuracy of the nG-signature. The length of the signature higher bits l and the number of 1 bits of the hash function t both influence the accuracy.

Let e denote the relative error of $est'(s_q, s_d)$. That is:

$$e = \frac{est'(s_q, s_d) - est(s_q, c(s_d))}{est'(s_q, s_d)} \quad (4)$$

Let $\bar{e}$ denote the expectation of e. We can get:

$$\bar{e} \approx \left(1 - \left(1 - \frac{t}{l}\right)^{|s_d|+n-1}\right)^t \quad (5)$$

Please refer to Appendix for how to get Equation 5. We can see that it is easy to determine t. When l is set, we can just choose a value $\hat{t}$ from all integers from 1 to $l-1$ that makes $\bar{e}$ the smallest, as we always want $\bar{e}$ to be as low as possible. The proper t for different $|s_d| + n - 1$ and l can be pre-calculated and stored in an in-memory table to save the run-time cpu burden.

Larger l will necessarily result in lower $\bar{e}$ according to Equation 5, and thus increase the efficiency of filtering, but on the other hand lower down the efficiency of scanning the index, as the space taken by nG-signatures is larger. So l controls the I/O trade-off between the filtering step and the refining step. Our experiments in later sections verify this point.

C. Encoding of Numerical Values

One solution of encoding a numerical value was proposed in the VA-file [23], [22], where the approximation code is generated through truncating some lower bits of the value. Intuitively, the domain (*absolute domain*) of the value are partitioned into slices of equal size. An approximation code indicates which slice the corresponding data value falls in, and through which, the minimum possible distance between the data value and a query value can be determined easily and false negatives are prevented. However, this method is too simple to efficiently perform the filtering task in actual applications. Although users often define large domain attributes, such as 32-bit integer, the actual values on such an attribute usually lie within a much smaller range and fall in very few slices, which lowers the distinguishability of the vector.

We propose encoding numerical values by cutting *relative domain* instead, which is the range between the minimum value and the maximum value on an attribute. In this way, shorter codes can reach the same precision as the encoding scheme using the absolute domain. If a value out of the existing relative domain is inserted, just encode it with the id of the nearest slice, which will not result in any false negative. Periodically renewing all approximation codes of an attribute with the new relative domain will ensure filtering efficiency.
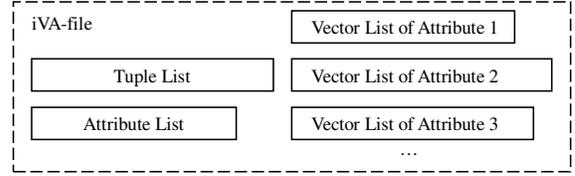

Fig. 5.   Structure of the iVA-file

D. iVA-File Structure

Now we have the nG-signature as the approximation vector for estimating edit distance, and the code on relative domain as the approximation vector for numerical values. We introduce the iVA-file to organize these vectors, which is very compatible and supports correct mapping between a vector and the value it represents in the table. An iVA-file consists of one tuple list, one attribute list, and multiple vector lists – one for each attribute, as shown in Fig. 5. Each list is organized as a sequence of list elements. We will introduce each list through describing the structure of list elements and how to order them in the list.

The tuple list holds elements corresponding to each tuple in the table. An element is a pair in the form of $< tid, ptr >$. tid is the identifier of the corresponding tuple. We assume the table file adopts the row-wise storage structure, such as the interpreted schema [6]. ptr records the starting address of the corresponding tuple in the table file. All elements are sorted in increasing order of tid. Note that the tids of two adjacent elements are not necessarily consecutive, as tuples are deleted or updated from time to time.

The attribute list holds elements corresponding to each attribute A_i in the table. An element is in the form of $< ptr_1, ptr_2, df, str, \alpha >$. ptr_1 and ptr_2 are the starting and tail addresses of A_i's vector list in the iVA-file. df records the number of tuples that have definition on A_i, and str is total number of all strings on A_i in the table (0 if A_i is a numerical attribute). α is a number ranging between 0 and 1, named *relative vector length*, that determines the length of approximation vectors on A_i. If A_i is a numerical attribute, the length of an approximation vector is $\lceil \alpha \cdot r \rceil$ where r is the length of a numerical value measured by bytes. If A_i is text attribute, the length of the nG-signature higher bits is $\lceil \alpha \cdot (|s_d| + n - 1) \rceil$ where $|s_d|$ is the length of the encoded data string measured by bytes. Since attributes are rarely deleted, we eliminate the attribute id in the element, and adopt the positional way to map any attribute to the corresponding element in the attribute list.

Each attribute has a corresponding vector list where approximation vectors are organized in increasing order of tuple ids. Partial scan is possible as any vector list can be scanned separately. The organization of vectors inside a vector list should support correct location and identification of any vector in the list during the sequential scan of the list. On the other hand, the organization should keep the size of the list as small as possible to reduce the cost of scanning. We propose four vector list organization structures suitable for different conditions, and the choice will be determined by the size.

tid	Color	Lens	Brand	Num
0		"Wide-angle"	"Sony"	
1	"White"		"Apple"	
3	"Red"			5
5		"Telephoto" "Wide-angle"	"Cannon"	
6	"Brown" "Black"		"Benz"	2

A sparse wide table

tid	Color	Lens	Brand	Num
000		000111	010001	
001	110001		110000	
011	101001			1110
101		101010 000111	000101	
110	111000 010010		110100	0000

The approximation vectors

Tuple List: 000 *ptr*   001 *ptr*   011 *ptr*   101 *ptr*   110 *ptr*
Type I for "Color":   001 110001   011 101001   110 111000   110 010010
Type II for "Lens":   000 01 000111   101 10 101010 000111
Type III for "Brand": 01 010001   01 110000   00   01 000101   01 110100
Type IV for "Num":   1111   1111   1110   1111   0000

Fig. 6.   An example of vector lists

Type I This structure is suitable for either a text attribute or a numerical one. The element in the vector list is the pair of a tuple id and the vector of the tuple on this attribute: $< tid, vector >$. The list does not hold vectors of ndfs. All elements are sorted in increasing order of tuple ids. A number of consecutive elements may have the same tid if the corresponding text value has multiple strings.

Type II This structure is only suitable for a text attribute. An element in the vector list is a tuple id, followed by the number of strings in the text value of this tuple on the corresponding attribute, and then all vectors for those strings: $< tid, num, vector_1, vector_2, ... >$. The list does not hold elements of ndf values. All elements are sorted in increasing order of tuple ids.

Type III This structure is only suitable for a text attribute. A list element is the number of strings in the text value of the corresponding tuple on this attribute, followed by all vectors for those strings: $< num, vector_1, vector_2, ... >$. The vector list holds elements for all tuples in the table, sorted by the corresponding tuple id in increasing order. The tuple corresponding to each element can be identified by counting the elements before it during the scanning of the list. Note that, in the element of a ndf value, num is 0, and no vector follows it.

Type IV This structure is only suitable for a numerical attribute. An element is $< vector >$. The vector list holds elements for all tuples, including those have ndf values on this tuple. A special vector code should be reserved to denote ndf. The elements are sorted by the corresponding tuple id in increasing order. The tuple of an element can be identified by the element position of the vector in the list.

Example 3.5: [Vector Lists] As shown in Fig. 6, we have a table and assume that we have already encoded the approximation vectors for all values in the table. If we use 3 bits to record a tuple id and 2 bits to record the number of strings of a text value, example vector lists of four types on four attributes are listed in Fig. 6, where 1111 is reserved as the approximation vector for ndf numerical value, and an underlined consecutive

---

Algorithm 1 Query Processing with iVA-file

Input: query Q, attribute list $aList[]$, tuple list $tList[]$
Output: temporary result pool $pool$
1: $pool \leftarrow$ an empty pool
2: **for all** A where $v(Q, A) \neq ndf$ **do**
3:      $scanPtr[A] \leftarrow aList[A].ptr_1$
4: **for** $i = 0$ to $|\mathcal{T}|$-1 **do**
5:      $currentTuple \leftarrow tList[i].tid$
6:      **for all** A where $v(Q, A) \neq ndf$ **do**
7:          $scanPtr[A].\text{MoveTo}(currentTuple)$
8:          $diff[A] \leftarrow$ estimate difference on A
9:      $dist \leftarrow$ calculate estimated distance from $diff[]$
10:     **if** $pool.\text{Size}() < k$ or $dist < pool.\text{MaxDist}()$ **then**
11:         read $currentTuple$ from table file
12:         $dist \leftarrow$ calculate actual distance
13:         **if** $pool.\text{Size}() < k$ or $dist < pool.\text{MaxDist}()$ **then**
14:             $pool.\text{Insert}(currentTuple, dist)$
15: **return** $pool$

---

part is a list element.   ∎

A text attribute can be indexed in one of the three formats, Type I, II and III. Let l_{tid} denote the space taken by a tuple id, and l_{num} denote the space taken by the value that records the number of strings in a text value. If all the vectors on the text attribute take a total space of L, the size of three list types can be pre-compared by the following equations without actually knowing the value of L where df and str can be found in the corresponding element in the attribute list:

$$L_I = l_{tid} \cdot str + L$$
$$L_{II} = (l_{tid} + l_{num}) \cdot df + L$$
$$L_{III} = l_{num} \cdot |\mathcal{T}| + L$$

A numerical attribute should adopt either Type I or IV. By calculating L_I and L_{IV}, the type with the smallest size should be adopted.

$$L_I = (l_{tid} + \lceil \alpha \cdot r \rceil) \cdot df$$
$$L_{IV} = \lceil \alpha \cdot r \rceil \cdot |\mathcal{T}|$$

IV. QUERY PROCESSING AND UPDATE

A. Query Processing

The query processing with the indices based on filter-and-refine strategy consists of two parts: filtering by scanning the index and refining through random accesses to the data file. The existing process proposed in the VA-file [23] is to scan the whole VA-file to get a set of candidate tuples, and check them all in the data file afterwards (*sequential plan*). This plan requires the approximation vector to be able to provide not only a lower bound of the difference to the query value but also a meaningful upper bound. Otherwise, the filtering step fails as all tuples are in the candidate set. However, a limited length vector cannot indicate any upper bound for unlimited-and-variable length strings as there has to be an infinite number of strings to share the same approximation vector. So we propose the *parallel plan*, where refining happens from time to time during the filtering process.

When running a query with the iVA-File, the tuple list and all vector lists related with the defined attributes in the query are scanned in a synchronized manner. We set a scanning pointer for each list, and initialize them with the start addresses of the lists. The scanning pointer of the tuple list moves forward one element at a time, which determines the current tuple being filtered ($currentTuple$) and guarantees that all tuples in the table will be filtered. The scanning pointer of a related vector list should move forward to point to the element of $currentTuple$. As a special case, in a vector list of Type I or II, there may be no element for $currentTuple$ due to the ndf value. In this case, the scanning pointer will point to an element with tid larger than $currentTuple$ or the tail address of the vector list. Then, this scanning pointer freezes until $currentTuple$ grows to the pointed tid. Assume that the member function of a scanning pointer MoveTo($currentTuple$) can achieve the above synchronization on a vector list.

With the help of scanning pointers, for each defined attribute in a query, we can either load the approximation vector(s) of $currentTuple$ in the corresponding vector list or directly determine that the value of $currentTuple$ on this attribute is ndf. Then, we can calculate the lower-bound of the difference between the data value and the query value on each defined attribute in the query. Using these lower-bounds, we can calculate an estimated similarity distance between $currentTuple$ and the query by the metric function f. According to the monotonous property of f, this distance is a lower bound of the actual distance.

For a query, we set a temporary result set, initialized to be empty. $currentTuple$ is a result candidate if and only if, the tuples in the temporary result set is less than k , or the maximum actual distance of the tuples in the temporary result set is larger than the estimated distance of $currentTuple$. If $currentTuple$ is a result candidate, read ptr of $currentTuple$ in the tuple list, and then load $currentTuple$ from the table file and calculate the actual distance. If the temporary result set has tuples less than k, just put $currentTuple$ in the set and record its actual distance. Otherwise, if the actual distance is smaller than the largest distance of tuples in the set, replace the tuple of the largest distance with $currentTuple$.

For the convenience of describing the algorithm of processing a query with the iVA-file, we assume that we have a temporary result pool maintained in the main memory called $pool$. $pool$ holds at most k pairs such as $< tid, dist >$ as we only need the top-k tuples. tid is a tuple id and $dist$ is tuple tid's actual distance to the query. $pool$.Size() gives the number of pairs stored in $pool$. $pool$.MaxDist() returns the largest $dist$ in $pool$. $pool$.Insert($tid, dist$) inserts the pair $< tid, dist >$ into $pool$: if $pool$ is not full, directly insert; otherwise we insert the new pair first, and then remove the pair with the largest $dist$. We present the query processing with the iVA-file in Algorithm 1.

In the algorithm of query processing with the iVA-file, the result pool is initialized in line 1. In line 2-3, the scanning pointers are set to the start addresses of the corresponding
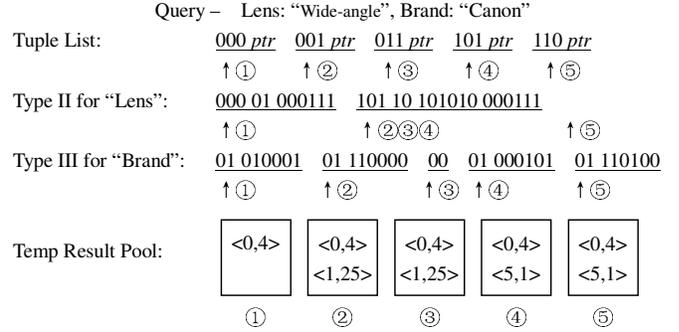
Query – Lens: "Wide-angle", Brand: "Canon"

Tuple List:	000 *ptr*	001 *ptr*	011 *ptr*	101 *ptr*	110 *ptr*
	↑①	↑②	↑③	↑④	↑⑤

Type II for "Lens":	000 01 000111	101 10 101010 000111			
	↑①	↑②③④			↑⑤

Type III for "Brand":	01 010001	01 110000	00	01 000101	01 110100
	↑①	↑②	↑③	↑④	↑⑤

Temp Result Pool:	<0,4>	<0,4>	<0,4>	<0,4>	<0,4>
		<1,25>	<1,25>	<5,1>	<5,1>
	①	②	③	④	⑤

Fig. 7. An example of processing a query

vector lists by reading ptr_1 of the attribute list elements of related attributes in the query. The algorithm filters all the tuples in the table in line 4-14. Line 5 gets the tuple id of the ith filtered tuple from the tuple list. For the ith tuple, the difference between the query value and the data value on all attributes related with the query are estimated in line 6-8. In line 9, we estimate the distance between the query and the ith filtered tuple. Line 10 judges whether the ith filtered tuple is a possible result and, if it is, the tuple is fetched from the table for checking in line 11-14.

Example 4.1: [**Query Processing**] Suppose we have a query defined on two attributes over the table and index in Fig. 6, say (Lens:"Wide-angle", Brand:"Canon"), and we want the top-2 tuples. The tuple list and the vector lists for attribute Lens and Brand are scanned to process the query. Since the table contains five tuples, the processing takes five steps, and the positions of the scanning pointers on each related list in each step are depicted in Fig. 7. Assume the distance function f is $d_{\texttt{Lens}} + d_{\texttt{Brand}}$, and the difference between a query string and ndf is constant 20. We now explain what happens in each step.

Step 1: All scanning pointers are set to the beginning of the lists. The current pointed element (CPE) of the tuple list shows that $currentTuple$ is 0. Since the tid of the CPE of Lens is also 0, the pointer will not freeze. Since the result pool has no tuples, just load tuple 0 from the table file and calculate the actual distance between tuple 0 and the query which is 4. Insert the $< tid, dist >$ pair $< 0, 4 >$ to the result pool.

Step 2: The pointer of the tuple list moves one element forward, and we get $currentTuple = 1$. The pointer of Lens moves forward and finds the tid of CPE is 5, larger than 1. So the pointer of Lens freezes so that it will not move in the next step. The pointer of Brand only need to move one element forward, as it adopts Type III vector list – a counting-way list. Since the result pool has less than 2 tuples, just load tuple 1 from the table file and calculate the actual distance which is 25. Insert $< 1, 25 >$ to the result pool.

Step 3: The pointer of the tuple list moves one element forward, and we get $currentTuple = 3$. The pointer of Lens still freezes as tid of CPE is 5, larger than 3, and we get ndf of tuple 3 on Lens, the difference of which to "Wide-angle" is 20. The pointer of Brand still moves one element forward, and we get the number of strings is 0, which indicates that

it is ndf of tuple 3 on `Brand`, and the difference should be 20. Then the estimated distance between the query and tuple 3 is 40. Since the result pool is full and 40 is larger than any distance in the pool, tuple 3 is impossible to be result.

Step 4: The pointer of the tuple list moves one element forward to get $currentTuple = 5$. The pointer of `Lens` is unfrozen as tid of CPE is 5, and we get two vectors 101010 and 000111. Assume that est("Wide-angle",101010) $= 5$, and est("Wide-angle",000111) $= 0$. So the estimated difference on `Lens` is 0. The pointer of `Brand` just moves one element forward, and we get the only vector 000101. The estimated difference on `Brand` is est("Canon",000101), say 0. Then the estimated distance between the query and tuple 5 is 0. Since there exist distances in the result pool larger than 0, tuple 5 might be a result. So, load tuple 5 from the table file and calculate the actual distance which is 1. Substitute $< 1, 25 >$ with $< 5, 1 >$ in the result pool.

Step 5: The pointer of the tuple list moves one element forward to get $currentTuple = 6$. The pointer of `Lens` moves forward and finds it is at the tail of the vector list. So, it freezes and we get it is ndf of tuple 6 on `Lens`. The estimated difference on `Lens` is 20. The pointer of `Brand` moves one element forward, and we get the vector 110100. Suppose est("Canon",110100) $= 3$. Tuple 6 is impossible to be result as the estimated distance is 23, larger than any distance in the result pool.

So we access the table file three times in steps 1, 2 and 4, and get the final result: tuple 0 with distance 4 and tuple 5 with distance 1. ∎

Although we are switching among different vector lists which seemingly results in a large amount of random accesses to the iVA-file, a small disk cache will avoid this problem as the number of defined attributes per query is very small.

B. Update

Insertion is straightforward. We simply add the new elements to the tail of the tuple list and corresponding vector lists. The tail of vector lists can be directly located by the ptr_2s in the attribute list. Since we assumed that the table file adopts the row-wise storage structure, the new tuple is appended to the end of the table file for an insertion. For a deletion, we just scan the tuple list to find the element of the deleted tuple and rewrite the ptr in the element with a special value to mark the deletion of this tuple, and we do not modify the vector lists and the table file. When querying, just skip the filtering of the deleted tuples. We should periodically clean deleted tuples in the table file and all related elements in the tuple list and vector lists by rebuilding the table file and the

iVA-file. For an update, we break it up into a deletion and an insertion, and we assign a new id to the updated tuple. Since insertions, deletions and updates are not as frequent as queries, periodically cleaning the deleted information will limit the size of the iVA-file and keep the scanning efficient.

V. EXPERIMENTAL STUDIES

In this section, we conduct experimental studies on the efficiency of the iVA-file (iVA), and compare its performance with the inverted index (SII) implementation proposed in [7]. We also recorded the performance of directly scanning of the table file (DST). The query processing time of the methods and the effects of various parameters on the efficiency of the iVA-file were studied. The VA-file is excluded from our evaluations as its size far exceeds that of the table file. The common precision/recall tests were skipped as all these methods provide precise query results.

A. Experiment Setup

We set up our experimental evaluation over a subset of Google Base dataset [5] in which $779,019$ tuples define $1,147$ attributes, where $1,081$ are text attributes and the others are numerical attributes. According to our statistics, 16.3 attributes are defined in each tuple on average and the average string length is 16.8 bytes. We adopt the interpreted schema [6] to store the sparse table, and the table file is 355.7 MB. The size of the SII is 101.5 MB and the sizes of the iVA-files with different parameters range from 82.7 MB to 116.7 MB. We set a 10 MB file cache in memory for the index and the table file operations. The cache is warmed before each experiment. To simulate the actual workload in real applications, we generate several sets of queries by randomly selecting values in the dataset so that the distribution of queries follows the data distribution of the dataset. Each selected value and its attribute id form one value in a structured query. Each query set has 50 queries with the first 10 queries used for warming the file cache and the other 40 for experiment evaluation. The number of defined values per query is fixed in one query set, and the query sets are preloaded into main memory to eliminate unwanted distractions to the results. Our experimental environment is a personal computer with Intel Core2 Duo 1.8GHz CPU, 2GB memory, 160GB hard disk, and Window XP Professional with SP2.

B. Query Efficiency

We first study the effects of the following parameters on the iVA-file, SII and DST to compare them: the number of defined values per query, the value of k for a top-k query, the metric of distance f between a query and a tuple, the setting of the importance weights of attributes. We also tune the relative vector length α and the gram length n to see their impacts on the iVA-file. The type of each vector list is automatically chosen as explained in Sec. III-D. The iVA-files under some settings are even smaller than the SII file, which reflects that the intellectual selection between multi-type vector lists contributes well to lower the index size. The
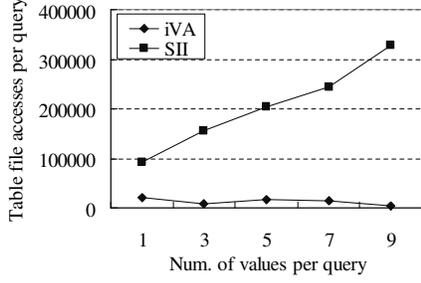
Fig. 8. Effect of the number of defined values per query on the data file access times per query.
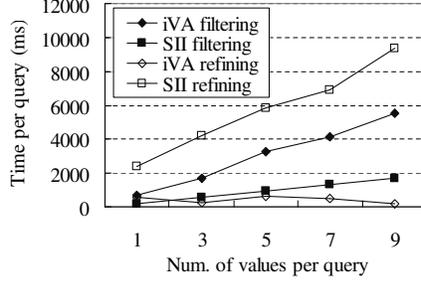

Fig. 9. Effect of the number of defined values per query on filtering and refining time per query.
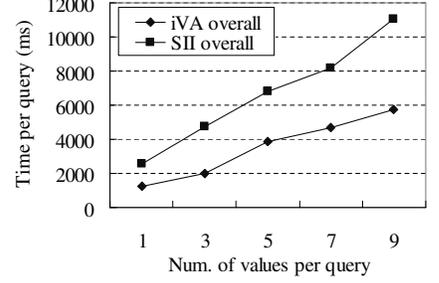

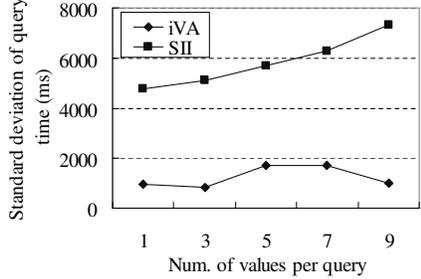Fig. 10. Effect of the number of defined values per query on the overall query time per query.

Fig. 11. Effect of the number of defined values per query on the standard deviation of query time.
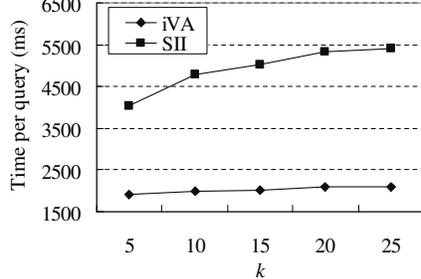

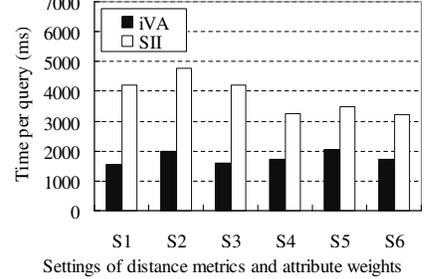Fig. 12. Effect of k of the top-k query on the query time.

Fig. 13. Effect of different settings of distance metrics and attribute weights.

default values of the parameters are listed in Table I and in each experiment we examine only one or two parameters in order to study their effects. The query processing time of DST is very stable under different parameter settings, always around 30 seconds per query. The results of the DST query efficiency were very poor and we left them out from comparisons in all figures.

1) Effects of Defined Values per Query:

In this experiment, we compare the iVA-file and SII by incrementally changing the number of values per query from 1 to 9 in steps of 2 to see their filtering efficiency and query processing time. Fig. 8 exhibits the average times of accessing the table file per query under different number of query values. The iVA-file accesses the table file only about $1.5\% \sim 22\%$ of SII, which means that the approximation vectors in the iVA-file performs very well in the filtering step. Another important fact is that the iVA-file table accesses do not steadily grow with the number of defined values per query. We divide the processing time of one query into two parts: filtering time and refining time, both of which include the corresponding CPU and I/O consumption. Fig. 9 compares the filtering and refining time per query of the iVA-file and SII. We can see that the iVA-file sacrifices on the filtering time while gains lower refining time. Fig. 10 gives the average query time and shows that the iVA-file is usually twice faster than SII. Moreover, the iVA-file also significantly improves the stability of single-query time as shown in Fig. 11, where we depict the standard deviation of query time with different number of values in each query.

2) Effects of k:

Under the scenario of the top-k query, k effects the efficiency of scan-based indices by influencing the rate of accessing the table file. In this experiment, we incrementally

vary the value of k from 5 to 25 in steps of 5 to examine the effects of k on the iVA-file and SII. The result is shown in Fig. 12. The iVA-file surpasses the SII in query efficiency for all ks. And the slope of the iVA-file curve is smaller, which indicates although the processing time per query inevitably increases, the iVA-file is still acceptable when k is big.

3) Effects of Distance Metrics and Attribute Weights:

The efficiency of the iVA-file with respect to different distance metrics and attribute weights is compared with SII. We evaluate the average query processing time per query on three distance metric functions: L_1-metric, L_2-metric and L_∞-metric. We also test it on two settings of the attribute weights: all weights are equal (EQU for short), and inverse tuple frequency (ITF). The ITF weight of an attribute A is

$$\ln \frac{1 + |\mathcal{T}|}{1 + |\mathcal{T}|_A}$$

where $|\mathcal{T}|$ is the total number of tuples and $|\mathcal{T}|_A$ denotes the number of tuples that define A. We set six scenarios of combinations of distance metrics and attribute weights S1~S6, which are EQU+L_1, EQU+L_2, EQU+L_∞, ITF+L_1, ITF+L_2 and ITF+L_∞ respectively. The iVA-file outperforms SII significantly for all these settings. The results are shown in Fig. 13.

4) Effects of nG-signature Parameters:

The key point of the iVA-file is the filter efficiency which depends on the granularity of approximation vectors and influences the rate of random accesses on the table file. Consequently, the settings of the nG-signature affect the query processing efficiency. We first examine the influence of the length of nG-signatures. Longer signatures provide higher precision at the cost of larger vector lists. So the length of nG-signatures influences the trade-off between the I/O of scanning
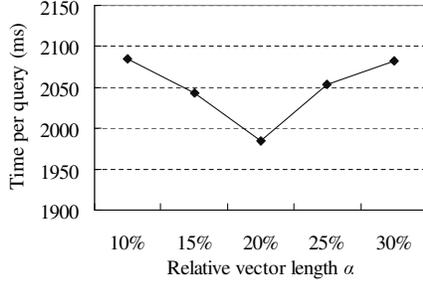
Fig. 14. Effect of the relative vector length α on the iVA-file query time.
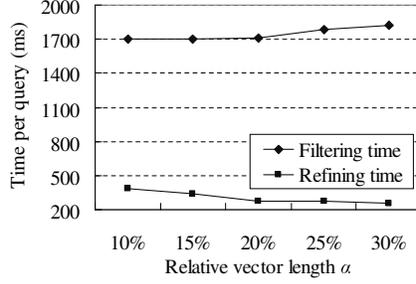

Fig. 15. Effect of the relative vector length α on iVA-file filtering and refining time per query.
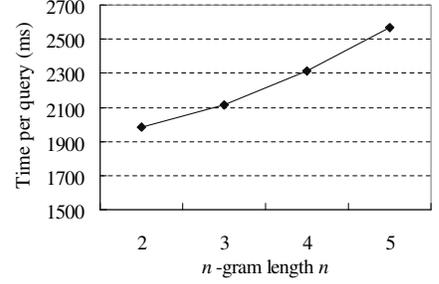

Fig. 16. Effect of the length of n-grams n on iVA-file query time.

the index and the I/O of random access on the table file. We test the average query processing time by incrementally changing the relative vector length α from 10% to 30% in steps of 5%. The query efficiency reaches the best when $\alpha = 20\%$ as shown in Fig. 14 as our expectation of the effects of the length of nG-signatures. We also test the average filtering and refining time per query with different α. Fig. 15 further verifies our point as the filtering time keeps growing with longer vectors, while the refining time drops steadily. We also evaluate the effects of n – the length of n-grams. We test the average query processing time for n equal to 2, 3, 4 and 5. As shown in Fig. 16, the average time of processing one query keeps growing as n grows. So $n = 2$ is a good choice for short text.

C. Update Efficiency

We compare the update efficiency of iVA, SII and DST. We run 10,000 deletions of random tuples, and get the average time per deletion denoted by t_d is 3.89ms, the same for iVA, SII and DST. We run insertions of all 779,019 tuples in the dataset, setting $\alpha = 20\%$: the total time denoted by t_r is the time of rebuilding the table file and the index file, and the average time of one insertion denoted by t_i is $t_r/|\mathcal{T}|$ where $|\mathcal{T}|$ is the total number of tuples in the table. As we mentioned in Sec. IV-B, the table file and the index file should be periodically rebuilt to clean up the deleted data. If we perform the cleaning every time when the amount of deleted tuples reaches a percentage β (*cleaning trigger threshold*) of all tuples in the table, the actual average time cost by one deletion, insertion and update are respectively:

$$t_d + \frac{t_r}{\beta \cdot |\mathcal{T}|}, \; t_i + \frac{t_r}{\beta \cdot |\mathcal{T}|}, \; t_d + t_i + \frac{t_r}{\beta \cdot |\mathcal{T}|}$$
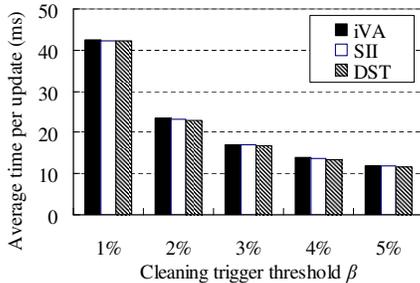

Fig. 17. Comparison of iVA, SII and DST's average update time under different cleaning trigger threshold β.

We compared the average insertion, deletion and update time of iVA, SII and DST for different rebuilding frequency. We only show the average time of an update operation for different β with $\alpha = 20\%$ in Fig. 17 changing β from 1% to 5% in steps of 1%, as the deletion and insertion have the similar property. Compared with the query time, update is around 10^2 faster. The iVA-file's average update time is very close to that of SII and DST. So we can conclude that the iVA-file outperforms SII and DST significantly in query efficiency but sacrifices little in update speed.

VI. CONCLUSIONS

The growing popularity of recent Web 2.0 and community based applications poses the problem of managing the sparse wide tables (SWT). Existing studies in this area mainly focus on the efficient storage of the sparse table, and so far only one index method, namely the inverted index, has been evaluated for enhancing the query efficiency. In this paper, we propose the inverted vector approximation file (iVA-file) as the first content-conscious index designed for similarity search on SWTs, which organizes approximation vectors of values in an efficient manner to support efficient partial scan required for answering top-k similarity queries. To deal with the large amount of short text values in SWTs, we have also proposed a new approximation vector encoding scheme nG-signature efficient in filtering tuples and preventing false negatives at the same time. Extensive evaluation using a large real dataset confirms that the iVA-file is a flexible and efficient indexing structure to support sparse datasets prevalent in Web 2.0 and community web management systems. On one hand, the index outperforms the existing methods significantly and scales well with respect to data and query sizes in query efficiency. On the other hand, the iVA-file sacrifices little in update efficiency. Further, being a non-hierarchical index, the iVA-file is suitable for indexing horizontally or vertically partitioned datasets in a distributed and parallel system architecture which is widely adopted for implementing the community systems.

ACKNOWLEDGEMENTS

References

[1] Windows Live Spaces. [Online]. Available: http://spaces.live.com/

[2] Wikipedia. [Online]. Available: http://www.wikipedia.org/

[3] Flickr. [Online]. Available: http://www.flickr.com/

[4] E. Chu, J. Beckmann, and J. Naughton, "The case for a wide-table approach to manage sparse relational data sets," in *SIGMOD*, 2007.

[5] Google Base. [Online]. Available: http://base.google.com/

[6] J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton, "Extending RDBMSs to support sparse datasets using an interpreted attribute storage format," in *ICDE*, 2006.

[7] B. Yu, G. Li, B. C. Ooi, and L. Zhou, "One table stores all: enabling painless free and easy data publishing and sharing," in *CIDR*, 2007.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *OSDI*, 2006.

[9] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava, "Approximate string joins in a database (almost) for free," in *VLDB*, 2001.

[10] H. Lee, R. T. Ng, and K. Shim, "Extending q-grams to estimate selectivity of string matching with low edit distance," in *VLDB*, 2007.

[11] C. Li, J. Lu, and Y. Lu, "Efficient merging and filtering algorithms for approximate string searches," in *ICDE*, 2008.

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

[13] W. W. Cohen, "Integration of heterogeneous databases without common domains using queries based on textual similarity," in *SIGMOD*, 1998.

[14] R. E. Tarjan, "Storing a sparse table," *Communications of the ACM*, 1979.

[15] B. C. Ooi, C. H. Goh, and K.-L. Tan, "Fast high-dimensional data search in incomplete databases," in *VLDB*, 1998.

[16] G. P. Copeland and S. Koshafian, "A decomposition storage model," in *SIGMOD*, 1985.

[17] R. Agrawal, A. Somani, and Y. Xu, "Storage and querying of e-commerce data," in *VLDB*, 2001.

[18] D. Maier and J. Ullman, "Maximal objects and the semantics of universal relation databases," *TODS*, 1983.

[19] P. Zezula, G. Amato, V. Dohnal, and M. Batko, *Similarity Search: The Metric Space Approach.* Hardcover, 2006.

[20] C. Yu, B. C. Ooi, K.-L. Tan, and H. V. Jagadish, "Indexing the distance: an efficient method to knn processing," in *VLDB*, 2001.

[21] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is "nearest neighbor" meaningful?" in *ICDT*, 1999.

[22] U. Shaft and R. Ramakrishnan, "Theory of nearest neighbors indexability," *TODS*, vol. 31, no. 3, pp. 814–838, September 2006.

[23] R. Weber, H. J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *VLDB*, 1998.

[24] G. Canahuate, M. Gibas, and H. Ferhatosmanoglu, "Indexing incomplete databases," in *EDBT*, 2006.

[25] W. Müller and A. Henrich, "Faster exact histogram intersection on large data collections using inverted VA-files," in *CIVR*, 2004.

[26] H. K. T. Wong, J. Z. Li, F. Olken, D. Rotem, and L. Wong, "Bit transposition for very large scientific and statistical databases," *Algorithmica*, vol. 1, no. 3, pp. 289–309, 1986.

[27] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficien compression," *TODS*, vol. 31, no. 1, pp. 1–38, March 2006.

[28] T. Apaydin, G. Canahuate, H. Ferhatosmanoglu, and A. S. Tosun, "Approximate encoding for direct access and query processing over compressed bitmaps," in *VLDB*, 2006.

[29] C. Chan and Y. E. Ioannidis, "An efficient bitmap encoding scheme for selection queries," in *SIGMOD*, 1999.

[30] A. Kent, R. Sacks-Davis, and K. Ramamohanarao, "A signature file scheme based on multiple organizations for indexing very large text databases," *JASIST*, vol. 41, no. 7, pp. 508–534, November 1988.

[31] J. Zobel, A. Moffat, and K. Ramamohanarao, "Inverted files versus signature files for text indxing," *TODS*, vol. 32, no. 4, pp. 453–490, December 1998.

[32] H. He, H. Wang, J. Yang, and P. S. Yu, "BLINKS: ranked keyword searches on graphs," in *SIGMOD*, 2007.

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

[34] G. Kondrak, "N-gram similarity and distance," in *SPIRE*, 2005.

[35] J. Ullman, "A binary n-gram technique for automatic correction of substitution, deletion, insertion, and reversal errors in words," in *The Computer Journal*, 1977.

[36] X. Yang, B. Wang, and C. Li, "Cost-based variable-length-gram selection for string collections to support approximate queries efficiently," in *SIGMOD*, 2008.

[37] H. V. Jagadish, N. Koudas, and D. Srivastava, "On effective multi-dimensional indexing for strings," in *SIGMOD*, 2000.

Appendix

A. nG-Signature Error Expectation Analysis

The possibility for a bit in $h[l,t](\omega)$ to be 0 is:

$$1 - \frac{t}{l}$$

Since the size of the n-gram set of s_d is $|s_d| + n - 1$, the possibility of a bit in $c_H[l,t](s_d)$ to be 1 is:

$$1 - \left(1 - \frac{t}{l}\right)^{|s_d|+n-1}$$

If ω is not an n-gram of s_d, the possibility that ω is a false hit is:

$$p = \left(1 - \left(1 - \frac{t}{l}\right)^{|s_d|+n-1}\right)^t \tag{6}$$

Let M denote the difference between the size of $g(s_q)$ and $cg(s_q, s_d)$. Then $M = |s_q| + n - 1 - |cg(s_q, s_d)|$. According to Equation 1, we have:

$$est'(s_q, s_d) \approx \frac{M}{n} \tag{7}$$

$|hg(s_q, c(s_d))| - |cg(s_q, s_d)| = i$ $(i = 0, 1, \cdots, M)$ means i false hits happen. So, the possibility of $|hg(s_q, c(s_d))| - |cg(s_q, s_d)| = i$ is:

$$\binom{M}{i} \cdot p^i \cdot (1-p)^{M-i}$$

Thus, the average $|hg(s_q, c(s_d))| - |cg(s_q, s_d)|$ is:

$$\overline{|hg(s_q, c(s_d))| - |cg(s_q, s_d)|}$$
$$= \sum_{i=0}^{M} i \cdot \binom{M}{i} \cdot p^i \cdot (1-p)^{M-i} = \sum_{i=1}^{M} i \cdot \binom{M}{i} \cdot p^i \cdot (1-p)^{M-i}$$
$$= pM \sum_{i-1=0}^{M-1} \binom{M-1}{i-1} \cdot p^{i-1} \cdot (1-p)^{(M-1)-(i-1)} \tag{8}$$

Substitute N for $M-1$, and substitute j for $i-1$.

$$\overline{|hg(s_q, c(s_d))| - |cg(s_q, s_d)|} = pM \sum_{j=0}^{N} \binom{N}{j} \cdot p^j \cdot (1-p)^{N-j}$$
$$= pM \left(p + (1-p)\right)^N = pM \tag{9}$$

According to Equation 4, 3 and 1, the estimation of e is:

$$\overline{e} = \frac{\overline{|hg(s_q, c(s_d))| - |cg(s_q, s_d)|}}{n \cdot est'(s_q, s_d)} = \frac{pM}{n \cdot est'(s_q, s_d)}$$

According to Equation 7, we have:

$$\overline{e} \approx p$$