EFFICIENTLY INDEXING SPARSE WIDE TABLES IN COMMUNITY SYSTEMS

HUI MEI

(B.Eng), XJTU, China

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2010

Acknowledgement

I would like to express my gratitude to all who have made it possible for me to complete this thesis. The supervisor of this work is Professor Ooi Beng Chin; I am grateful for his invaluable support. I would also like to thank Associate Professor Anthony K. H. TUNG, Associate Professor Chan Chee Yong and Dr Panagiotis Karras for their advice.

I wish to thank my co-workers in the Database Lab who deserve my warmest thanks for our many discussions and their friendship. They are Chen Yueguo, Jiang Dawei, Zhang Zhenjie, Yang Xiaoyan, Chen Su, Wu Sai, Tam Vohoang, Zhou Yuan, Wu Ji, Wang Nan, Dai Bintian, Zhang Dongxiang, Cao Yu and Wang Tao.

I am very grateful for the love and support of my parents and my parents-in-law.

I would like to give my special thanks to my husband Guo Chen, whose patient love has enabled me to complete this work.

CONTENTS

LIST OF FIGURES

Summary

The increasing popularity of Community Web Management Systems(CWMSs) calls for tailor-made data management approaches for them. In CWMSs, 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 not well defined in each tuple. Low-dimensional structured similarity search and general complex query on a combination of numerical and text attributes is common operations. However, many properties of wide tables and their associated Web 2.0 services render most multi-dimensional indexing structures ineffective. Recent studies in this area have mainly focused on improving the efficiency of storage management and the deployment of inverted indices; so far no new data structure 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 attribute information and the content of attribute values. Furthermore, it is not sufficient for complex queries. In this thesis, we examine this problem and propose iVA-file indexing structure for structured similarity query and CW2I indexing scheme for complex query respectively.

The iVA-file works on the basis of approximate contents and guarantees 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 present an efficient query processing strategy for the iVA-file, which is different from strategies used for existing scan-based indices. To enable the usage of different metrics of distance between a query and a tuple varying 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, while at the same time keeps a good update speed.

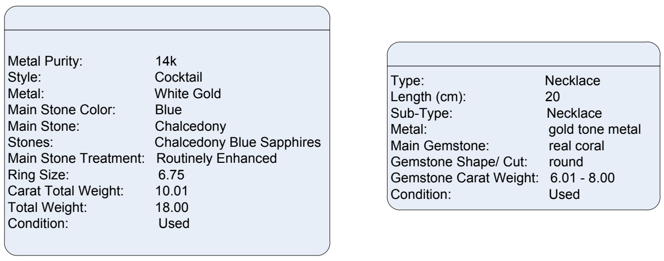
CW2I combines two effective indexing methods: inverted index and direct index for each attribute. Inverted index gathers a list of tuples which are sorted by tuple ID for each attribute value; the inverted index is sorted by value itself. Separate direct index for each attribute provides fast access to those tuples for which the given attribute is defined. The direct index is sorted by tuple ID following a column-oriented architecture. Comparative experiments demonstrate that our proposed scheme outperforms other approaches for answering complex queries on community web data.

In summary, this thesis proposes indexing techniques for efficient structured similarity query and complex query over sparse wide table in community systems. Extensive performance studies show that these proposed indices significantly improve the query performance.

CHAPTER 1

Introduction

We have witnessed the increasing popularity of Web 2.0 systems such as blogs [6], Wikipedia [5], Facebook [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 (CWMSs) have been successfully applied in an extensive range of communities because of their effectiveness in collecting the information and organizing the wisdom of crowds. The increasing popularity of CWMSs calls for tailor-made data management approaches for them. It drives the design of new storage platforms that impose requirements unlike those of conventional database systems and it needs effective and efficient query schemes. Due to it, humongous volume of data has also led to the proposal of new cluster based systems for large data analysis such as Map Reduce and Hadoop.



Metal Purity:	14k
Style:	Cocktail
Metal:	White Gold
Main Stone Color:	Blue
Main Stone:	Chalcedony
Stones:	Chalcedony Blue Sapphires
Main Stone Treatment:	Routinely Enhanced
Ring Size:	6.75
Carat Total Weight:	10.01
Total Weight:	18.00
Condition:	Used

Type:	Necklace
Length (cm):	20
Sub-Type:	Necklace
Metal:	gold tone metal
Main Gemstone:	real coral
Gemstone Shape/ Cut:	round
Gemstone Carat Weight:	6.01 - 8.00
Condition:	Used

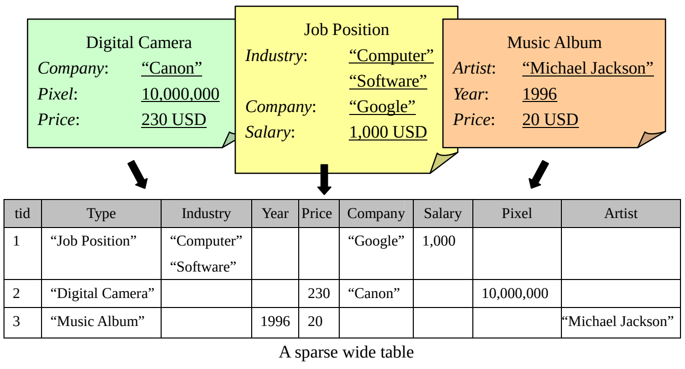
Figure 1.1: Data Items in eBay

1.1 Data in CWMS

Community Web Management Systems (CWMSs) provide a platform in which users of a community can interact and collaborate. Users can contribute to, and take ownership of, the content and display their collective knowledge. In general, a CWMS database stores information on a wide-ranging set of entities, such as products, commercial offers, or persons. Due to diverse product specifications, user expectations, or personal interests, the data set, when rendered as a table, can be very sparse and comprises a good mix of alphanumeric and string-based attributes. For example, there are millions of collectibles, decor, appliance, computers, cars, equipment, furnishings and other miscellaneous items are listed, bought or sold on e-commerce system eBay [1] every day. Each item is described by a set of attributes specified as shown in Figure 1.1. The first item is a ring, and it is described by eleven attributes such as metal purity, style and ring size etc. The second item is a necklace, and it has five different attributes. Both the ring and the necklace fall into category jewelry. As the items are being submitted into the system, the new attributes are added to the current categories and new categories are added to the catalog. As a result, there will be thousands attributes in the system. However, each item is described by a small subset of the attributes only. For another example, the dataset of the CNET e-commerce system examined by Chu et al. [26] comprises a total of $2,984$ attributes and $233,304$ products; still, on average a product is described by only ten attributes. Likewise, most community-based data publishing systems, such as Google Base [4], allow users to define their own meta data and store as much information as they wish, as shown in Figure 1.2. Users may submit different types of items as shown in Figure 1.2, such as digital camera, job position and music album, and describe these data items using different attributes. 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 [4], where $779,019$ items define $1,147$ attributes and the average number of attributes defined in each item is 16. The characteristics of the dataset in CWMSs are summarized as follows:

- The dataset consist of a large number of attributes, due to the diverse product specifications.
- The dataset is very sparse. The dataset when rendered as a horizontal table will have thousands of columns, but each data item is described by only ten or so attributes. Each data item has NULL values for most of the attributes. As a result, the dataset is very sparse.
- The schema is evolving as new data items are added, the new attributes are also introcuced. Therefore, the schema of the dataset is not fixed, but it is evolving all the time.

To facilitate fast and easy storage and efficient retrieval, the wide table storage structure has been proposed in [17, 26, 51, 25]. The wide table can be physically implemented as vertical tables and file-based storage [26, 51]. In this thesis, the dataset in CWMSs is referred as sparse wide table(SWT).



tid	Type	Industry	Year	Price	Company	Salary	Pixel	Artist
1	"Job Position"	"Computer" "Software"			"Google"	1,000		
2	"Digital Camera"			230	"Canon"		10,000,000	
3	"Music Album"		1996	20				"Michael Jackson"

Figure 1.2: Users submit freely defined meta data to the sparse wide table.

1.2 Queries in CWMS

The fast development and popularity of CWMSs calls for flexible and efficient way to search the data items and information shared in CWMSs. Recent research [44] 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 [28], the SWT, which is the only table maintained for each application does not impose strict relational constraint on the 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 introduce problems to text-centric IR-based query processing.

In this thesis, we investigate and propose efficient query processing techniques for two types of queries as follows:

1. Structured Similarity Query

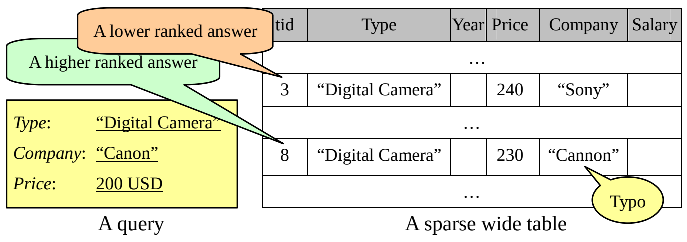


Figure 1.3: A structured similarity query in CWMSs.

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 Figure 1.3. CWMS ranks 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 Figure 1.3 should be "Canon". To facilitate the ranking, edit distance [30, 40, 41], a widely used typo-tolerant metric, is adopted to evaluate the similarity between two strings.

2. General Complex Query

To our knowledge, there is no existing CWMS provides SQL equivalent selection queries such as "retrieve a set of objects that have the same value for a given single attribute", or "find all products sold in Jakarta". However such a way of querying CWMSs data is not only relevant to the data at hand, but also attainable. Thus, it is essential to identify a reasonable indexing scheme for efficiently and scalably processing complex and general queries.

1.3 Motivation

Recent studies on SWTs, such as the interpreted schema [17, 26, 51], mainly focus on optimizing the storage scheme of datasets. To the best of our knowledge, no new indexing techniques have been proposed, and so far only the inverted index has been evaluated for SWTs in [51]. For each attribute, a list of identifiers of the tuples that are well defined 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.

In addition, the existing multi-dimensional indices that have been designed for multi-dimensional and spatial databases are not supposed to be suitable and efficient for SWTs, due to 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.

To the best of our knowledge, none of the existing approaches for Community Web Data Management provides a satisfactory solution for neither structured similarity query processing nor complex query processing. Indeed, existing SWT management schemes are not designed with such queries in mind. Instead, they aim at providing easy access to attribute-value pairs, to the set of values defined for a given object, or to a range of objects.

In this thesis, we propose an indexing structure that stores approximation vectors as the approximate representation of data values, and supports efficient partial scan and similarity search. In addition, we espouse an architecture that puts binary vertical representation and inverted index together and allows them to interact with each other to support efficient complex query processing.

1.4 Contribution

The main contribution of this thesis are summarized as follows:

- We conduct an in-depth investigation on storing and indexing wide sparse tables.

- We propose iVA-file as an indexing structure that stores approximation vectors as the rough representation of data values, and supports efficient partial scan and similarity search. It is the first content-conscious indexing mechanism designed to support structured similarity queries over SWTs prevalent in Web 2.0 applications. We have conducted extensive experiments using real CWMS datasets and the results show that the iVA-file is much more efficient than the existing approaches.

- We combine inverted index and direct index for each attribute to improve the performance of complex query processing. The *inverted* index for each attribute gathers a list of tuples which are sorted by tuple ID; the inverted index is sorted by the attribute value itself. The separate *direct* index for each attribute provides fast access to those tuples for which the given attribute is defined. The separate *direct* index is sorted by tuple ID, following a column-oriented architecture inspired by [20, 21, 56]. We conduct a performance evaluation using the GoogleBase dataset and compare our proposed method to existing ones. The results confirm that the proposed indexing scheme we propose outperforms the systems based on a monolithic vertical-oriented or horizontal-oriented representation. Our proposed scheme can efficiently handle complex queries over community data.

1.5 Organization of Thesis

The rest of the thesis is organized as follows:

- Chapter 2 introduces related work about SWTs storage and indexing structure.

- In Chapter 3, the iVA-file structure is introduced. We describe the encoding scheme of both strings and numerical values. In order to reduce cost of scanning the index file we propose four types of iVA-file structures suitable for different conditions. Based on the iVA-file structure we discuss its query processing and update. We describe the experimental study conducted on the iVA-File, inverted index and directly scanning of the table file scheme.

- In Chapter 4, we propose the CW2I index structure for complex query in CWMSs. We describe the index structure and the experimental study CW2I, horizontal storage scheme, vertical storage scheme and iVA-file scheme.

- Chapter 5 concludes the work in this thesis with a summary of our main findings. We also discuss some limitations and indicate directions for future work.

CHAPTER 2

Related Work

It has been long observed that the relational database representations are not suited for emerging applications with sparsely populated and rapidly evolving data schemas. In this chapter we present an overview of existing approaches for both storage and index of sparse wide tables.

2.1 Storage Format 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 based on the schema of the relational table. However, for sparse wide tables (SWT), a horizontal storage scheme is not efficient due to the large amount of undefined values (*ndf*). A cursory study of the storage problem of the sparse table may suggest the following approaches such as binary vertical representation [29], ternary vertical representation [11], and interpreted storage format [17]. These approaches have the

Oid	Attr1	Attr2	Attr3	Attr4
1	a1	a2	a3	--
2	b1	--	--	b4
3	--	c2	c3	c4
4	d1	--	--	--
5	e1	e2	--	--

Figure 2.1: A sparse dataset in horizontal schema.

possibility to alleviate the problem of ndfs and the number of attributes.

2.1.1 Binary Vertical Representation

A natural approach to handling sparse relational data is to split a sparse horizontal table into as many binary (2-ary) tables as the number of attributes (columns) in the sparse table. This idea was first suggested in the context of database machines [47] and was brought up again with the decomposition storage model [29]. In DSM[29], the authors proposed to fully decompose the table into multiple binary tables, the values of different attributes are stored in different tables. Figure 2.1 shows a sparse table stored in horizontal storage schema, In Figure 2.2 the horizontal table is decomposed into 4 tables one for each column in the horizontal table. In decomposed storage schema, each table has two columns; one is Oid which ties different fields of the horizontal table across these binary tables. The second column stores the value of the corresponding attribute. Using DSM only non-null values are stored, but any operation requesting multiple attributes requires the reconstruction of the tuple of the original horizontal table. This type of column-store model has been followed by MonetDB, along with an algebra to hide the decomposition [20, 21], as well as C-Store [56], gaining the benefits of compressibility [8] and performance [10]. Furthermore, in [7], Abadi suggested that, apart from data warehouses and OLAP workloads, column-stores may also be well suited for storing extremely sparse and wide tables.

Attr1

Oid	Val
1	a1
2	b1
4	d1
5	e1

Attr2

Oid	Val
1	a2
3	c2
5	52

Attr3

Oid	Val
1	a3
3	c3

Attr4

Oid	Val
2	b4
3	c4

Figure 2.2: A sparse dataset in decomposed storage format.

2.1.2 Ternary Vertical Representation

Agrawal et al. [11] discerned that a ternary (3-ary) vertical representation offers a hybrid design point between the n-ary horizontal representation of conventional RDBMSs for non-sparse data and the binary vertical representation outlined above. They found that this vertical representation does uniformly outperform the horizontal representation for sparse data, yet the binary representation performs better. This approach has been employed by many commercial software systems for storing objects in a sparse table, hence [11] investigated how to best support it, by creating a logical horizontal view of the vertical representation and transforming queries on this view to the ternary vertical table. Like the conventional horizontal representation, the ternary vertical representation requires only one giant table to store all the data; it does not split the table into as many tables as the number of attributes. Figure 2.3 shows the same sparse table stored in vertical schema. A tuple in horizontal schema is decomposed into several tuples in vertical schema. A ternary vertical table contains entries of the scheme $<Oid$ (object identifier), Aid (attribute identifier), Val (attribute value)$>$. Thus, it contains tuples for only those attributes that are present for an object. Different attributes of an object are linked together using the same *Oid*. Thus, the arguments in favor of the ternary vertical representation focuses around its flexibility in supporting schema evolution and manageability, as it maintains a single table instead of as many tables as the number of attributes in the binary scheme. In response, [11] suggested the use of multiple, *partial indexes*, i.e., one index on each of the three columns of the ternary vertical table, along the line of [55]. A premonition of a multiple-indexing approach is also contained in this suggestion.

Still, a similar approach to non-relational data representation has been followed in the context of RDF data storage for Semantic Web applications. In this context, RDF triples of the schema <*Sid* (subject identifier), *Pid* (property identifier), *Oid* (object identifier)> have been stored in a giant *triples table*, analogous to the ternary storage system for sparse tables [13, 14, 16, 22, 31, 32, 52, 61, 45]. Indeed, [11] also suggested that, among others, a potential application of the work it reported includes stores for RDF.

Hence, the limitations faced by the ternary architecture for sparse data are analogous to those faced by triples stores for RDF data. Indeed, simple similarity, lookup, or statement-based queries can be efficiently answered by such systems. However, such queries do not constitute the most challenging way of querying sparse data. More complex queries, involving multiple steps like unions and joins, call for a more sophisticated approach.

2.1.3 Interpreted Storage Format

Beckmann et al. [17] argued that, in order to efficiently scale to applications that require hundreds or even thousands of sparse attributes, RDBMSs should provide an alternative storage format that would be independent of the schema width. The suggestion for such a format introduced in [17] is the *interpreted storage format*. Figure 2.4 shows the first tuple in horizontal table in Figure 2.1 stored in interpreted attribute storage format, the first three fields constitute the header, the following fields are the attribute-value pairs. In this format, only the non-null values are stored and the fields of a single tuple are stored together unlike the vertical schema or DSM the value of the single tuple are stored independent of each other. In particular, it stores a list of attribute-value pairs for each tuple. In other words, the interpreted storage format gathers together the attribute-identifier and attribute-value entries of a single object-identifier that would appear separately in ternary vertical representation, and creates a single tuple for them, without explicitly storing null values for the undefined attributes. Unfortunately, as observed in [17], the interpreted format renders the retrieval of values from attributes in tuples significantly more complex. As the name of this format suggests, the system must discover the attributes and values of a tuple at tuple-access time, rather than using pre-compiled position information from the catalog. To ameliorate this problem, [17] suggested an *extract* operator that returns the offsets to the referenced interpreted attribute values. Still, as also noted in [7, 9, 64], handling sparse tables by this format incurs a significant performance overhead.

Chu et al. [26] argued that the option of collecting the sparse data set into a very wide, very sparse table, could actually be an attractive alternative. They did observe the lack of indexability as one of the major reasons why this approach would appear as unappealing, and suggested building and maintaining a sparse B-tree index over *each* attribute, as well as materialized views over an automatically discovered hidden schema, to ameliorate this problem. Thus, following the idea of using one partial index over each of the three columns of the ternary vertical table as in [11], [26] suggested the use of many sparse indexes, which are a special case of partial indexes [55]. Such indexes are effective for avoiding whole-table scans when answering range and aggregate queries. However, it is of little help for more complex queries involving unions and joins. Besides, the usage of a sparse index over each attribute imposes additional storage requirements, while, as noted in [49],

Oid	Key	Val
1	Attr1	a1
1	Attr2	a2
1	Attr3	a3
2	Attr1	b1
2	Attr4	b4
3	Attr2	c2
3	Attr3	c3
3	Attr4	c4
4	Attr1	d1
5	Attr1	e1
5	Attr2	e2

Figure 2.3: A sparse dataset represented in the vertical schema.

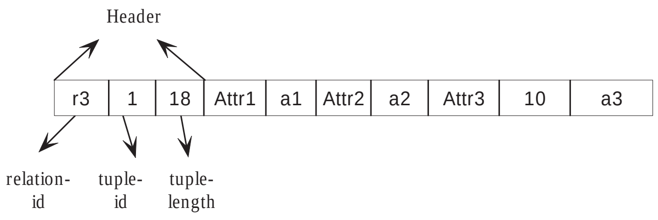


Figure 2.4: Interpreted attribute storage format.

it does not effectively address the resulting issues of efficient query optimization and processing.

These studies merely focus on enhancing the query efficiency through diverse organization of data storage. [26] 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 a sparse B-tree index on all attributes is recommended in [26], 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 [51]. 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. It is possible to bin and map attribute values into a smaller set of ranges and use a bitmap index [24] to index the dataset. However, the transformation may cause loss of information and similarity search on the index has not shown to be efficient.

The SWT in our context is different from the Universal Relation [46], which has also been discussed in [26, 51]. 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.

2.2 Indexing Schemes

2.2.1 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 such as the iDistance [68] 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 [19, 54] due to the curse of dimensionality. Weber et al. [63] 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 file to the data file. The vector approximation file (VA-file) divides the data space into 2^b rectangular cells and each cell is represented by a bit string of length b. The Data which falls into the cell is approximated by the bit string of the cell. The VA-file is much smaller than the original file and it supports 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 [23]. 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 as other indices [54]. 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 [17, 26, 51] 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.

Another multi-dimensional index based on sequential scan is the bitmap index [65, 66, 15]. As a bit-wise index approach, the bitmap index is efficiently supported by hardware at the cost of inefficient update performance. Compression techniques [66, 15] have been proposed to manage the size of the index. The bitmap index is an efficient way to process complex multidimensional select queries for read-mostly or append-only data, and is not known to be able to support similarity queries efficiently. It does not support text data although many encoding schemes have been proposed [65, 24].

2.2.2 Text Indices

The inverted index and the signature file [36, 69] 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 [33, 43] treat a record as a text document ignoring the attributes.

Many non-keyword similarity measures of strings have been proposed [39], among which edit distance could be most widely adopted [60, 30, 40, 41]. 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 [30]. The inverted index on n-grams [41] 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 [67]. A multi-dimensional index for unlimited-length strings was proposed in [35] 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.

2.3 String Similarity Matching

In CWMSs, most of the attributes are short string values, and typos are very common because of the participation of large groups of people. In this section, we introduce the background and the related work of string similarity matching.

2.3.1 Approximate String Metrics

There are a variety of approximate string metrics, including edit distance, cosine similarity and Jaccard similarity. Edit distance is a widely used typo-tolerant metric to evaluate the similarity between tow strings, due to its applicability in many scenarios. Edit distance 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 [30]. For example, the edit distance between hello and hallo is 1. Particularly, we can transform the first string to the second string by substituting the second character of the first string with character 'a'. Many recent works [59, 57, 30, 42] on string similarity matching adopt edit distance as the approximate string metric.

2.3.2 n-Gram Based Indices and Algorithms

n-gram[1] is widely used for estimating the edit distance between two strings [59, 57, 30, 40, 42]. 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 [40]. For example, to obtain all the 3-grams of "yes", we first extend it to "##yes\$\$". So "##y", "#ye", "yes", "es\$" and "s\$\$" are the 3-grams of "yes".

[59, 57, 30, 40, 38, 42, 18] proposed algorithms based on n-grams of strings to answer string similarity queries. These algorithms rely on the following observation: if the edit distance between strings are within a threshold *theta*, then they should share a certain number of common grams, and this lower bound is related to the gram-length n and the threshold *theta*. In [59], the authors argued that the edit distance leads to dynamic programming that is often relatively slow. The approximate string-matching problem could be solved faster for n-gram distance than for edit distance. A linear algorithm is proposed to evaluate the n-gram distance between two strings. However, the relationship between n-gram distance and edit distance is not examined and no index structured is designed. Therefore this algorithm won't scale well when the string dataset is very large. [57] introduced an algorithm based on sampling which utilize the fact that the preserved q-grams have to be approximately at the same location both in the pattern and in its approximate match. But location information of the n-gram will introduce additional

[1] Also called non-positional n-gram in some literatures.

space cost and sampling creates false negatives. In [30], a technique for building approximate string join capabilities on top of commercial databases by exploiting facilities already available in them. The properties of n-gram are adopted to filter the results. In particular the filters are count filter, position filter and length filter which can be implemented easily using SQL expressions. [40] proposed framework based on extending n-gram with wildcards to estimate selectivity of string matching with low edit distance. It is based on string hierarchy and combinatorial analysis but not applicable for string similarity query processing. [42, 18, 38] proposed the adoption of inverted-list index structure of the grams in strings to support approximate string queries. [42] improves the approximate string query performance and reduces the index size by proposing variable-length grams, but it can only support edit distance. [38] proposed the two level n-gram inverted index to reduce the size of the index and improve the query performance while preserving the advantages of the n-gram inverted index. [18] improved the performance of [42] by introducing cost-based quantitative approach to deciding good grams for approximate string queries. Compared to these studies, our work focuses on structural similarity queries, which contain information about the different attributes.

2.4 Summary

In this chapter, we have reviewed the current work on storage format and indexing schemes on wide sparse table. We also have discussed the approximate string metrics, n-gram based indices and algorithms.

CHAPTER 3

Community Data Indexing for Structured Similarity Query

3.1 Introduction

Structured similarity query is an easy-to-use way for users to express demand of data. In this chapter, we design the iVA-file, an indexing structure works on the basis of approximate contents and keeps scanning efficiency within a bounded range[1]. We introduce the nG-signature to encode both of the numerical values and strings which guarantees no false negative. 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 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.

The rest of this chapter is organized as follows. Section 3.2 introduces the formal definition of the problem. In Section 3.3, we describe the encoding schemes for both

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

string values and numerical values. Section 3.4 introduces the index structure–iVA-file structure. Query processing algorithm and update strategy are introduced in Section 3.5 and Section 3.6 respectively. Experimental study is explained in Section 3.7. We conclude in Section 3.8.

3.2 Problem Description

The wide table does not conform to the relational data model, and it aims to provide fast insertion of tuples with a subset of 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 the 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 Figure 1.2.

In this chapter, 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 $K = \min\{k, |\mathcal{T}|\}$.

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 chapter, 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 at-

Table 3.1: Table of notations

Notation	Explaination
$\mathcal{A}$	set of all attributes in the large table
A	an attribute
$\mathcal{T}$	set of all tuples in the table
$\|\mathcal{T}\|$	the number of the tuples
Q	a query
$v(T, A)$	the value in tuple T on attribute A
$D(T, Q)$	similarity distance between query Q and tuple T
$ed(s_1, s_2)$	edit distance between s_1 and s_2
$est(s_1, s_2)$	estimated edit distance between s_1 and s_2
$d[A](T, Q)$	
$c(s)$	nG-signature of String s
$g(s)$	n-gram set of string s
$cg(s_1, s_2)$	*common n-gram set* of two strings s_1 and s_2
$\|cg(s_1, s_2)\|$	size of common n-gram set of s_1 and s_2
$hg(s_q, c(s_d))$	n-gram set of s_q which is a hit on the nG-signature of s_d

tribute 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, states 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 thesis 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 metrics and attribute weight settings through experiments over real datasets. Table 3.1 summarize the notation used in this chapter.

We design a new index method named the inverted vector approximation file (iVA-file). 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 (Section. 3.3.1) or a numerical value (Section. 3.3.2) 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 (Section. 3.4).

3.3 Encoding Schemes

We propose encoding schemes to encode the string values and numerical values in the table to improve the efficiency of measuring similarity between two values.

3.3.1 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 Section. 3.5. We confine ourselves to introducing the encoding scheme and the calculation of $est(s_q, c(s_d))$ here.

A. nG-Signature

The nG-signature $c(s)$ of a string s is a bit vector that consists of two parts. The higher bits denoted by $c_{\mathrm{H}}[l, t](s)$ $(0 < t < l)$ and the lower bits denoted by $c_{\mathrm{L}}(s)$. The lower bits record the length of s. The higher bits are generated in the following steps as shown in Figure 3.1, first, we generate all the n-grams of the

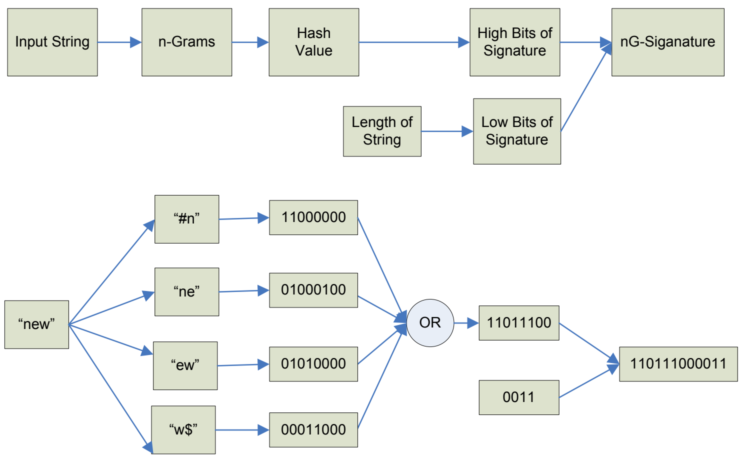


Figure 3.1: An example of generating a string's nG-signature

string; second, we use a has 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. Third, we execute log OR of all $h[l,t](\omega_i)$, where ω_i is an n-gram of s. In the last step we append the lower bits to the higher bits to generate the nG-signature of string s.

Example 3.1: [n**G-Signature**] Suppose a string is "new". The 2-grams are "#n", "ne", "ew" and "w$". $l = 8$, $t = 2$ and use 4 bits to record the string length. The process of encoding the c("new") is shown in Figure 3.1. □

B. Edit Distance Estimation with nG-Signature

We calculate $est(s_q, c(s_d))$ based on the method proposed in [30]. Let $g(s)$ denote the n-gram set of string s. For the purpose of estimating edit distance, the same n-grams starting at different positions in s should not be merged in the n-gram set [30]. 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 3.2: [n-**Gram Set**] The 2-gram set of string "aaaa" is {(1,"#a"), (3,"aa"), (1,"a$")}. It has the size of 5. □

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:

$$est'(s_q, s_d) = \frac{\max\{|s_q|, |s_d|\} - |cg(s_q, s_d)| - 1}{n} + 1 \tag{3.1}$$

According to [30]:

$$est'(s_q, s_d) \leq ed(s_q, s_d) \tag{3.2}$$

[30] 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 $c_{\mathrm{L}}(s_d)$, 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 $c_{\mathrm{H}}[l, t](s_d)$ 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. ∎

Consequently, we have the following property:

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 says 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 provide 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 but ω is not an n-gram of s_d.* ∎

An example of False Hit is shown in Figure 3.2, "ow" is a hit in the nG-signature of "new", but "ow" is not a n-gram of "new".

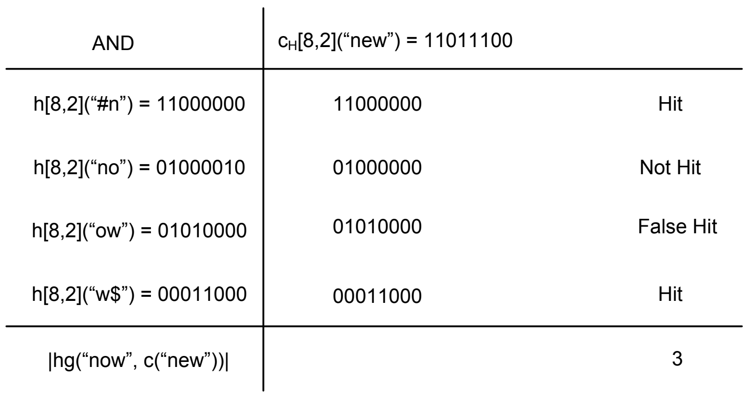
We define the hit gram set $hg(s_q, c(s_d))$ as follows:

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 to estimate $|cg(s_q, s_d)|$ in Equation 3.1 with $|hg(s_q, c(s_d))|$. There-



AND	c_H[8,2](“new”) = 11011100	
h[8,2](“#n”) = 11000000	11000000	Hit
h[8,2](“no”) = 01000010	01000000	Not Hit
h[8,2](“ow”) = 01010000	01010000	False Hit
h[8,2](“w$”) = 00011000	00011000	Hit
|hg(“now”, c(“new”))|		3

Figure 3.2: An example of estimating edit distance with nG-signature

fore the edit distance estimation function for the iVA-file is:

$$est(s_q, c(s_d)) = \frac{\max\{|s_q|, |s_d|\} - |hg(s_q, c(s_d))| - 1}{n} + 1 \tag{3.3}$$

Example 3.3: [**Edit Distance Estimation**]Suppose that the data string is “new” and the query string is “now”. As in Example 3.1, $l = 8$, $s = 2$, and we adopt the same hash function. So the higher bits of the nG-signature of “new” is 11010101. The 2-grams of “now” are “#n”, “no”, “ow” and “w$”. The process of calculating $|hg(s_q, c(s_d))|$ is shown in Figure 3.2. According to Equation 3.3, the edit distance is estimated as 0.5. We can safely loosen it to 1. □

We prove that the lower-bounding estimation causes no false negatives by the following proposition.

Propositon 3.1 *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.

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 3.1 and 3.3, we have:

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

According to Equation 3.2, we obtain:

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

■

C. nG-Signature Parameters

Proposition 3.1 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:

$$e = \frac{est'(s_q, s_d) - est(s_q, c(s_d))}{est'(s_q, s_d)} \tag{3.4}$$

Let $\overline{e}$ denote the expectation of e.

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 $c_{\mathrm{H}}[l,t](s_d)$ 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:

$$p=\left(1-\left(1-\frac{t}{l}\right)^{|s_d|+n-1}\right)^{t} \tag{3.5}$$

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 3.1, we have:

$$est'(s_q,s_d)\approx\frac{M}{n} \tag{3.6}$$

$|hg(s_q,c(s_d))|-|cg(s_q,s_d)|=i$ $(i=0,1,\cdots,M)$ implies that there are i false hits. 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:

$$\begin{aligned}&\overline{|hg(s_q,c(s_d))|-|cg(s_q,s_d)|}\\=&\sum_{i=0}^{M} i\cdot\binom{M}{i}\cdot p^i\cdot(1-p)^{M-i}=\sum_{i=1}^{M} i\cdot\binom{M}{i}\cdot p^i\cdot(1-p)^{M-i}\\=&pM\sum_{i-1=0}^{M-1}\binom{M-1}{i-1}\cdot p^{i-1}\cdot(1-p)^{(M-1)-(i-1)}\end{aligned} \tag{3.7}$$

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

$$\begin{aligned}\overline{|hg(s_q, c(s_d))| - |cg(s_q, s_d)|} = pM \sum_{j=0}^{N} \binom{N}{j} \cdot p^j \cdot (1-p)^{N-j} \\ = pM\,(p + (1-p))^N = pM\end{aligned} \tag{3.8}$$

According to Equation 3.4, 3.3 and 3.1, 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 3.6, we have:

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

$$\overline{e} \approx \left(1 - \left(1 - \frac{t}{l}\right)^{|s_d|+n-1}\right)^t \tag{3.9}$$

We can see that it is easy to determine t. When l is set, we can just choose a value $\widehat{t}$ from all integers from 1 to $l-1$ that makes $\overline{e}$ the smallest, as we always want $\overline{e}$ 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 $\overline{e}$ according to Equation 3.9, 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 chapter verify this point.

3.3.2 Encoding of Numerical Values

Quantization was proposed in the VA-file [63, 54] to encode a numerical alue, where the approximation code is generated by 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 prevented. However, this method is too simple to fulfill 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 are usually within a much smaller range and fall in very few slices, which lowers the filtering efficiency.

We propose encoding numerical values by using *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.

3.4 iVA-File Structure

After having introduced the our encoding scheme where we use nG-signature as the approximate vector for string values and the code on relative domain as the approximate vector for numerical values. We will introduce the iVA-File to organize these vectors. The iVA-File is very compatible and supports correct mapping between a vector and the value it represents in the table. The encoding vector lists

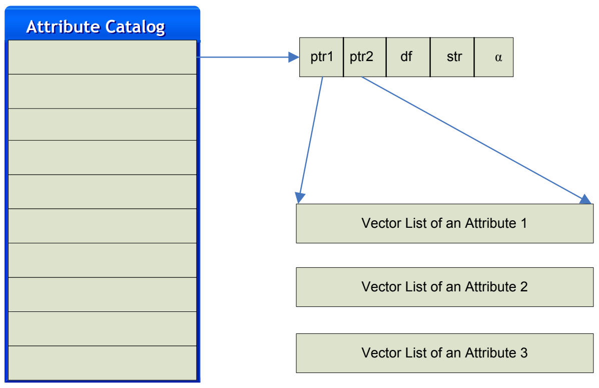

Attribute Catalog
ptr1
ptr2
df
str
α
Vector List of an Attribute 1
Vector List of an Attribute 2
Vector List of an Attribute 3

Figure 3.3: Structure of the iVA-file

of all the attributes and the attribute catalog are maintained in iVA-File as how in Figure 3.3. The list is organized as a sequence of list elements.

We store the data items in a vector list, referred to as tuple 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 [17]. 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.

In the iVA-file, we have an attribute catalog, which holds elements corresponding to each attribute A_i in the table. An entry in the attribute catalog 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 catalog.

Each attribute has a corresponding vector list where approximation vectors are organized in increasing order of tuple *id*s. Partial scan is possible as any vector list can be scanned separately. The organization of approximation 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.

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 *ndf*s. All elements are sorted in increasing order of tuple *id*s. 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 *id*s.

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*

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

Figure 3.4: An example of vector lists

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 3.4: [**Vector Lists**] As shown in Figure 3.4, 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 Figure 3.4, where 1111 is reserved as the approximation vector for *ndf* numerical value, and an underlined consecutive 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:

$$L_{\mathrm{I}} = l_{\mathrm{tid}} \cdot str + L$$

$$L_{\mathrm{II}} = (l_{\mathrm{tid}} + l_{\mathrm{num}}) \cdot df + L$$

$$L_{\mathrm{III}} = l_{\mathrm{num}} \cdot |\mathcal{T}| + L$$

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.

$$L_{\mathrm{I}} = (l_{\mathrm{tid}} + \lceil \alpha \cdot r \rceil) \cdot df$$

$$L_{\mathrm{IV}} = \lceil \alpha \cdot r \rceil \cdot |\mathcal{T}|$$

3.5 Query Processing

As in most Filter-and-refine processing strategies, query processing based on the iVA-File consists of two steps: filtering by scanning the index and refining through random accesses to the data file. The existing process proposed in the VA-file [63] 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. Consequently, we propose the *parallel plan*, where refining happens from time to time during the filtering process.

The algorithm flow is shown in Figure 3.5. When processing a query with the

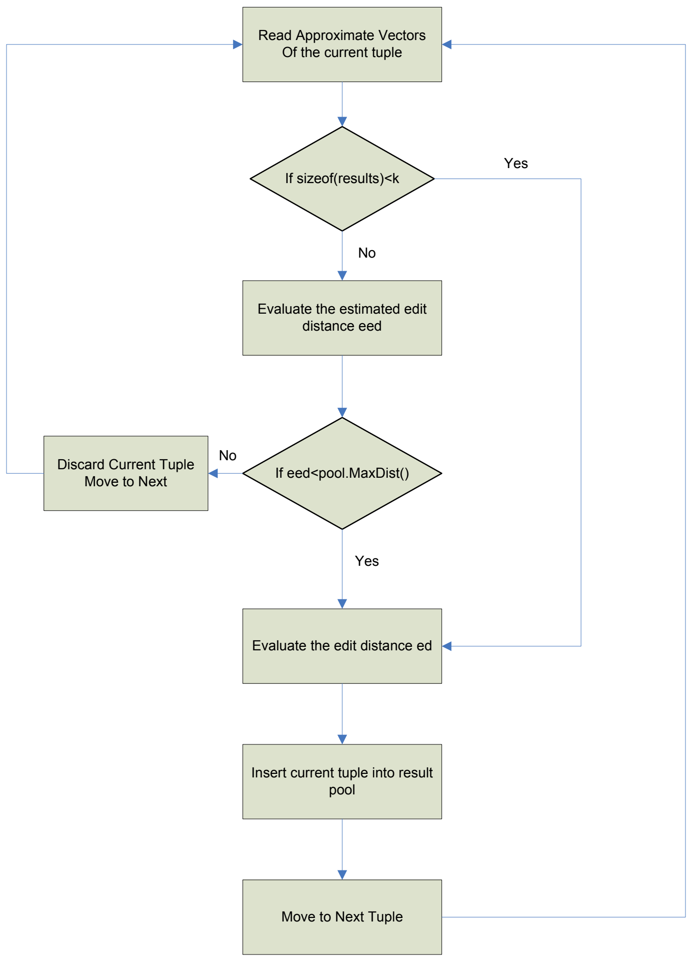


Figure 3.5: The Query Processing Algorithm Flow Chart

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 temporal result set, initialized to be empty. *currentTuple* is a result candidate if and only if, the tuples in the temporal result set is less than k , or the maximum actual distance of the tuples in the temporal 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 temporal 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 temporal 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

Algorithm 1 Query Processing with iVA-file

```
Require: query $Q$, attribute list $aList[]$, tuple list $tList[]$
Ensure: temporal 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: end for
 5: for $i = 0$ to $|\mathcal{T}|$-1 do
 6:     $currentTuple \leftarrow tList[i].tid$
 7:     for all $A$ where $v(Q, A) \neq ndf$ do
 8:         $scanPtr[A]$.MoveTo($currentTuple$)
 9:         $diff[A] \leftarrow$ estimate difference on $A$
10:     end for
11:     $dist \leftarrow$ calculate estimated distance from $diff[]$
12:     if $pool$.Size()$< k$ or $dist < pool$.MaxDist() then
13:         read $currentTuple$ from table file
14:         $dist \leftarrow$ calculate actual distance
15:         if $dist < pool$.MaxDist() then
16:             $pool$.Insert($currentTuple$, $dist$)
17:         end if
18:     end if
19: end for
20: return $pool$
```

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 2-4, the scanning pointers are set to the start addresses of the corresponding 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 5-19. Line 6 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 7-10. In line 11, we estimate the distance between the query and the ith filtered tuple. Line 12 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 13-17.

Example 3.5: [**Query Processing**] Suppose we have a query defined on two attributes over the table and index in Figure 3.4, 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 Figure 3.6. 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 unfreezed 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. □

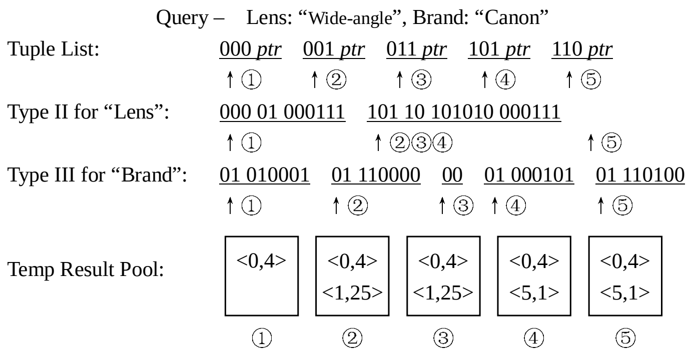


Figure 3.6: An example of processing a query

3.6 Update

Insertion is straightforward. We simply append the new elements to the end 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 assume 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.

3.7 Experimental Study

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 [51]. 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.

3.7.1 Experiment Setup

We set up our experimental evaluation over a subset of Google Base dataset [4] 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 [17] 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 follow 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.

Table 3.2: Default settings of experiment parameters

Parameter	Default Setting
Defined values per query	3
k	10
Distance metric	Euclidean
Attribute weight	Equal
α	20%
n	2

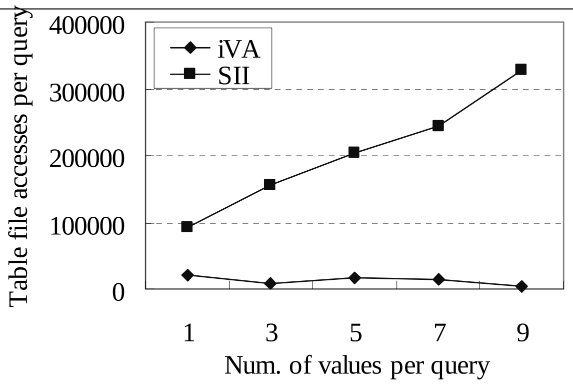


Figure 3.7: Effect of the number of defined values per query on the data file access times per query.

3.7.2 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. 3.4. 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 default values of the parameters are listed in Table 3.2 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

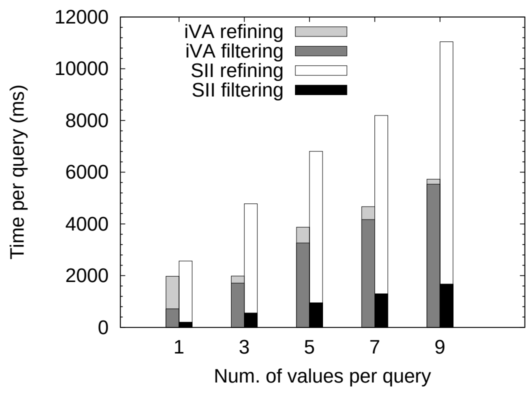


Figure 3.8: Effect of the number of defined values per query on filtering and refining time per query.

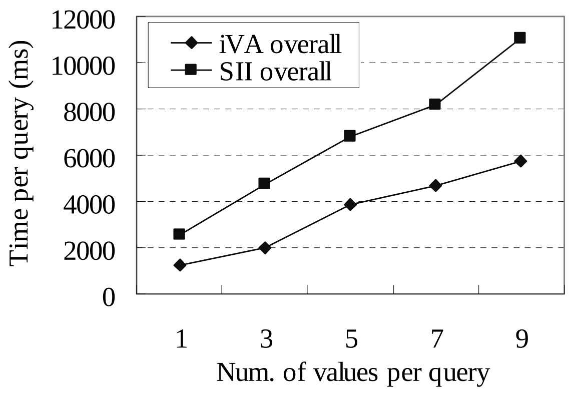


Figure 3.9: Effect of the number of defined values per query on the overall query time per query.

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.

A. 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. Figure 3.7 exhibits the average times of accessing the

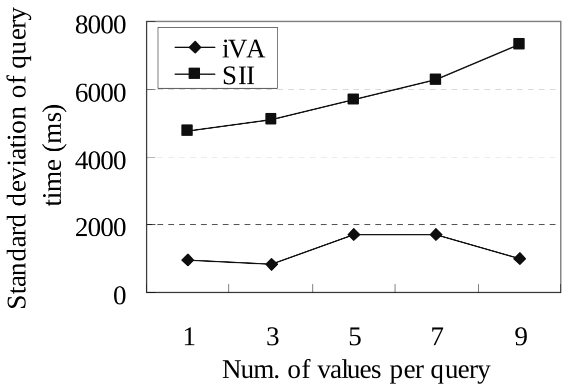


Figure 3.10: Effect of the number of defined values per query on filtering and refining time per query.

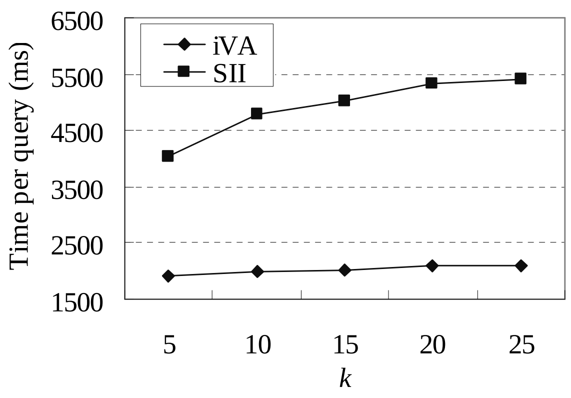


Figure 3.11: Effect of k of the top-k query on the query time.

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. Figure 3.8 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. Figure 3.9 gives the average query

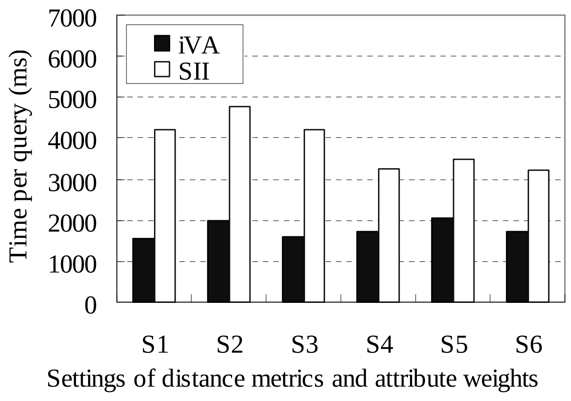


Figure 3.12: Effect of different settings of distance metrics and attribute weights.

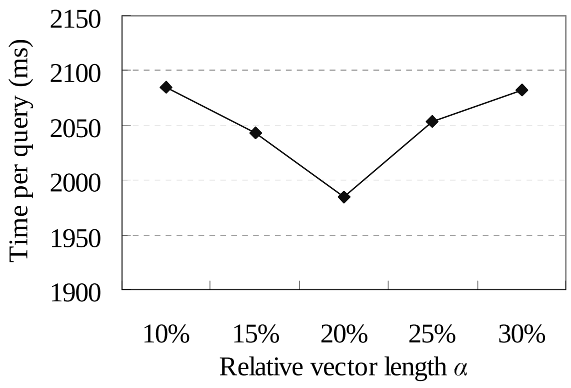


Figure 3.13: Effect of the relative vector length α on the iVA-file query time.

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 Figure 3.10, where we depict the standard deviation of query time with different number of values in each query.

B. Effects of k

Under the scenario of the top-k query, k affects 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 scalability

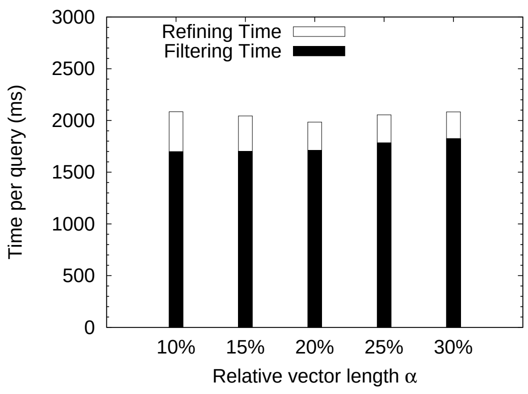


Figure 3.14: Effect of the relative vector length α on iVA-file filtering and refining time per query.

of the iVA-file and SII. The result is shown in Figure 3.11. Thanks to the tight lower bound of iVA-file querying processing scheme, the iVA-file surpasses the SII in query efficiency for all k. And the slope of the iVA-file line is smaller, which indicates that although the processing time per query inevitably increases as the value of k does, the iVA-file is still acceptable when k is big.

C. 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 Figure 3.12.

D. 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 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 Figure 3.13 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. Figure 3.14 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 Figure 3.15, the average time of processing one query keeps growing as n grows. So $n = 2$ is a good choice for short text.

3.7.3 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,

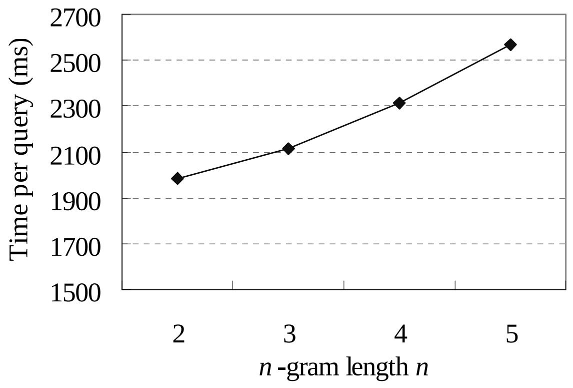


Figure 3.15: Effect of the length of n-grams n on iVA-file query time.

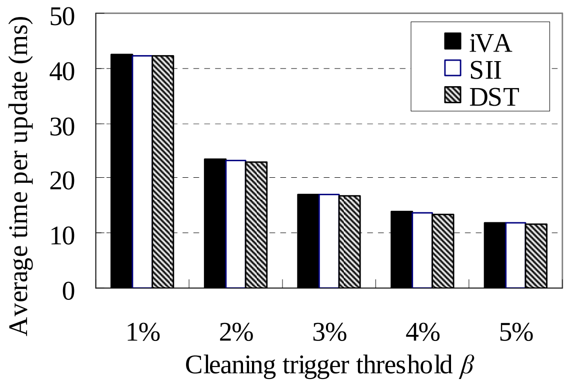


Figure 3.16: Comparison of iVA, SII and DST's average update time under different cleaning trigger threshold β.

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 $t_{\mathrm{r}}/|\mathcal{T}|$ where $|\mathcal{T}|$ is the total number of tuples in the table. As we mentioned in Sec. 3.6, 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:

$$t_{\mathrm{d}} + \frac{t_{\mathrm{r}}}{\beta \cdot |\mathcal{T}|},\ t_{\mathrm{i}} + \frac{t_{\mathrm{r}}}{\beta \cdot |\mathcal{T}|},\ t_{\mathrm{d}} + t_{\mathrm{i}} + \frac{t_{\mathrm{r}}}{\beta \cdot |\mathcal{T}|}$$

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 Figure 3.16 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.

3.8 Summary

In this chapter, we have presented a new approach to answer structured similarity query over SWTs in CWMSs. The proposed solution includes a content-conscious and scan-efficient index structure and a novel query processing algorithm which is suitable for any rational similarity metrics and guarantee no false negatives. Experimental results clearly show that iVA-file outperforms the existing proposals in query efficiency significantly and scales well with respect to data and query sizes. At the same time, it maintains a good update speed.

CHAPTER 4

Community Data Indexing for Complex Queries

4.1 Introduction

Most existing CWMSs either are disadvantaged by a lack of scalability to large data sets, or offer good performance only for specialized kinds of queries. None of the solutions has provided to date a scheme that can handle ever-increasing amounts of data as well as allow efficient processing of general-purpose complex queries. In this chapter, we propose a two-way indexing scheme that facilitates efficient and scalable retrieval and complex query processing with community data.The unique features and contributions of the proposed approach are:

- We combine two effective indexing methods: First, an inverted index for each attribute gathers a list of tuples, sorted by tuple ID, for each attribute value; the inverted index is sorted by value itself. Second, a separate direct index for each attribute provides fast access to those tuples for which the given attribute is defined, sorted by tuple ID, following a column-oriented architecture.

- We propose that, for the sake of both storage efficiency and functionality, less frequent queried attributes should receive a tailor-made treatment.
- We identify four different kinds of complex queries and extend CW2I scheme to handle these queries.

The reminder of this chapter is structured as follows. In Section 4.2 we introduce two-way indexing solution for SWT data management. The indexing construction and query processing steps are explained with examples. In Section 4.3, we introduce four types of complex query and extend CW2I to handle all of them. Section 4.4 presents our experimental results using real world data sets. In Section 4.5 we provide the summary of this chapter.

4.2 CW2I: Two-Way Indexing of Community Web Data

We propose the combination of *two* design approaches for complex query processing on SWTs. *First*, we espouse a binary vertical representation for each attribute *Aid* defined in the data at hand, which collects a *sorted* vector of *Tid* (tuple identifier) entries. Each of these entries is appended with an associated sorted list of attribute values *Val*. We call this binary representation *direct* index. The binary vertical representation of [29, 37, 20, 7] can be seen as a manifestation of our *direct* index. *Second*, we argue for an inverted index built over each *frequent* queried attribute. An attribute identifier *Aid* is linked to an *inverted* index, consisting of a sorted vector of *Val* (attribute value) entries, appended with their associated sorted lists of *Tids* $<Tid_1, Tid_2, \ldots, Tid_n>$ that match the given value for the *Aid* attribute. In case an attribute value consists of several short strings, these are independently indexed by our inverted index. String separators prevalent during data entry are used for this purpose. This design amounts to double indexing scheme for Community Web data, which we call CW2I. It is capable to address both lookup and aggregation queries, as well as more complex queries involving several join operations. In effect, it robustly handles all ways of querying the data.

4.2.1 The Unified Inverted Index

The main problem arising out of queried attribute skewness is that, if we are supposed to build both a *direct* and an *inverted* index for each attribute, then we raise unreasonable storage requirements and most of attributes are never required in the queries. After all, if an attribute is defined for only a few tuples by just a few users or in the worst case by just only one user, then not much stands to be gained by indexing the few values it assumes over the whole data set in both a *direct* and an *inverted* manner, because few users or no user is expected to express queries using the names of such attributes. By analyzing the query log, we pick the attributes which have 0.9 possibility to appear in a query as *frequent queried attributes*. Therefore, we suggest that *less frequent queried attributes* may best be seen as repositories of *unstructured* information about tuples. As far as *text* attributes are concerned, this information can be appropriately considered as a collection of keywords related to the tuples in question. Thus, we propose to handle lesser queried *text* attributes in a *unified* manner, gathering all of them together in a *unified inverted index*. This index provides a powerful keyword-search-like functionality over these attributes, while it increases both the storage-efficiency and the user-friendliness of the system. In particular, the unified inverted index gathers a sorted vector of attribute value entries, regardless of the attribute they correspond to; each entry is associated to a list $<Tid$ (tuple identifier), Aid (attribute

identifier)> pairs.

Numerical attributes are excluded from the unified inverted index; this discussion pertains to *text* attributes only. However, numerical attributes that fall below the frequency threshold to receive their own inverted index, are not indexed in this manner at all. Given the sparsity of such attributes, an inverted index is redundant for them. A lookup to their direct index is sufficient to detect any tuples that match a given value-based predicate. Besides, such numerical values do not offer anything in terms of keyword-search. Users are expected to refer to such attributes by name. The same reasoning applies to the case when a user needs to refer to a specific lesser-used text attribute by name; again, the low density of the attribute itself renders a lookup for values matches in the direct index practicable enough.

The usage of this unified inverted index addresses a problem that is particular to the community web data we examine. Moreover, it confers the following advantages to our CW2I indexing scheme:

- Storage efficiency, as it is not efficient from a storage point of view to have a separate inverted index on each of the myriads of less-frequent attributes.
- Facilitation of keyword-search queries, in which a given string is to be found in *any* less frequent attribute. Resorting to a unified index of all less-frequent attributes for keyword-search is more efficient than checking many small indexes. Besides, lesser-used text attributes can be validly seen as collecting keywords related to the specific domain where an entry belongs.
- User-friendliness, users are not expected, or required, to know obscure attribute by name. Users are mostly familiar with the names of the most commonly-used attributes, but, naturally, they cannot easily figure out by what name the others are entered. These lesser-used attributes are usu-

ally domain-specific and their appearance depends on the values of other attributes.

- Functionality and practical sense: for rarely used attributes, the direct index is already good enough for a lookup, hence the inverted index can be spared. Thus, while the said benefits are gained by unifying of what could be several smaller indexes, not much is significantly lost.
- Safeguard against user inconsistency. Different users are likely to define the same lesser-used concept with differently-named attributes. Thus, it makes sense to collect the values they provide under one unified index. A keyword search on the values of such attributes is guaranteed to return the tuples related to them (i.e., there are no false negatives).

4.2.2 Examples

We proceed to illustrate the CW2I system with examples of indexing and query processing.

A. Indexing

We offer an example that illustrates the CW2I indexing mechanism in relation to the sample data in Figure 1.2. The *direct* index for the Price attribute should contain the <*Tid*, *Val*> pairs $< 2, 230 >$ and $< 3, 20 >$. Likewise, the *inverted* index for the same attribute should contain an entry for *Val* 20, appended with a list of matching tuples, containing, in this case, the tuple $\{3\}$, as well as another entry for *Val* 230, appended with a tuple list containing the tuple $\{2\}$. The same pattern is replicated for all tuples and attributes in the table.

Furthermore, if we assume that Industry and Artist are less frequent attributes than the top quartile of attributes in terms of frequency, then the *unified inverted* index should contain entries for *Computer*, *Software*, and *Michael Jackson*. Both of the *Computer* and *Software* entries are to be amended with a list containing the $<$*Tid*, *Aid*$>$ pair $< 1, 2 >$, while the lists amended to the *Michael Jackson* entry should contain the $<$*Tid*, *Aid*$>$ pair $< 3, 8 >$. The same pattern is repeated for all other less frequent attributes in the table.

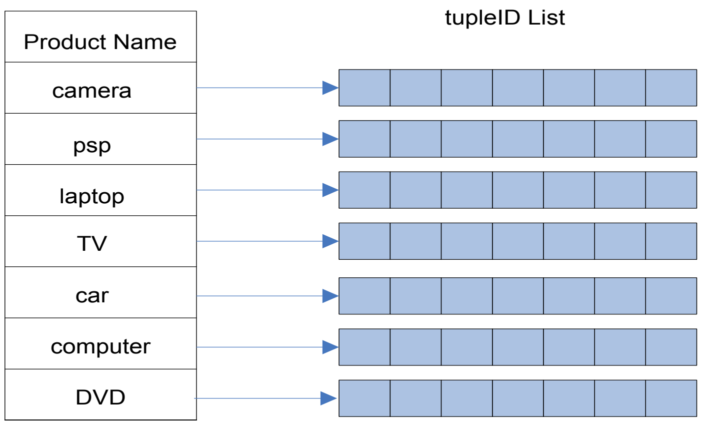


Figure 4.1: Example Query: First Step

B. Query Processing

As an example of query processing using a CW2I system, assume that, in an e-commerce system like googlebase [4], we wish to find all the products provided by companies that also provide laptops. Processing of this query starts out by accessing the inverted index of the Product attribute, to retrieve the (sorted) list A of all *Tids* for which product is *Laptop* (see Figure 4.1).

In the next step, the retrieved *Tid*-list A is merge-joined with the direct index of the Company attribute, to derive the list B of Company Names that offer laptops (see Figure 4.2).

Having derived list B, we can now resort to the inverted index of the Company attribute, to collect a *Tid* list L_i for each Company Name value v_i in B, and

Attribute: Company Name

Tuple ID	Values
1	John's home
2	ABC.ltd
3	pp.com
4	TaoBao
5	Amazon
6	IBM
7	Microsoft

Figure 4.2: Example Query: Second Step

construct the union of all L_i lists to get the list C of all *Tids* associated with Companies that have laptops on offer (see Figure 4.3).

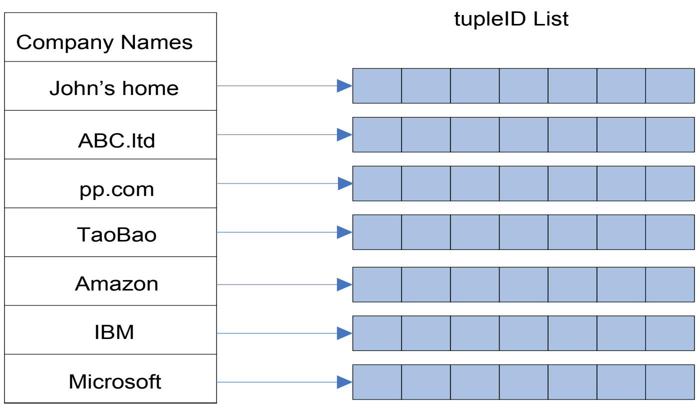


Figure 4.3: Example Query: Third Step

Lastly, we extract all products on offer by companies that also have laptops on offer by merge-joining the direct index of the Product attribute with list C (see Figure 4.4).

4.2.3 Argumentation

The main advantages of our two-way indexing scheme in relation to earlier CWMSs are be outlined as follows:

Attribute: Product Name

Tuple ID	Values
1	PSP
2	camera
3	TV
4	DVD
5	computer
6	laptop
7	car

Figure 4.4: Example Query: Fourth Step

- **Devoted Attention to Text Attributes.** Community web data contain myriads of text attributes, which cannot be efficiently handled by existing systems. CW2I separates text attribute values to the words they are composed from and indexes each word separately, either in the inverted index for that attribute per se, or in the unified inverted index. Moreover, CW2I defines a fuzzy join operator over the indexed short strings. Thus, it facilitates complex query processing and keyword-search operations over these values.

- **Concise and efficient handling of multi-valued attributes.** An attribute that gathers more than one value is naturally accommodated in CW2I, indexed in both a *direct* and *inverted* manner. The direct index gathers all attribute values in a list. Likewise, the inverted index collects all tuple IDs that share the same value (or short-string element).

- **Avoidance of NULLs.** Only those attributes that are relevant to a particular tuple need to be stored in a particular index. Thus, storage space is saved. Other systems, such as [11, 17] have tried to tackle the nullity problem in multifarious ways, but were always entangled within a sparse-table representation, hence some storage space was always devoted for them, either for representing them per se, or, as in [17], for specifying the attribute names of non-null attributes. By eschewing this representation, we provide an efficient handling of sparsity.

- **Prevalence of merge-joins.** Due to the indexing of attribute in both a direct and an inverted manner, sorted lists of tuple IDs both of those tuples defined for a given (more frequent) attribute, as well as of those tuples that a share a particular value for a given attribute, are readily available. Thus, most equi-join operations are bound to be fast merge-joins of such lists. By contrast, other sparse-and-wide data management system would need to perform multiple whole-table scans in order to execute complex join operations, seriously undermining their performance.

- **I/O efficiency.** CW2I minimizes the information that needs to be accessed for query processing, while avoiding the proliferation of whole-table scans that other Community Web data management systems suffer from. Depending on attribute value by which a query is bound, CW2I retrieves a list of tuples related to that value via an inverted index, without redundant accesses. Thus, CW2I eliminates redundant data accesses thanks to its two-way indexing architecture.

4.3 Query Typology

Although CW2I is designed to answer complex query based on exact matching efficiently, it can be easily extended to fuzzy matching. As we have discussed, several attribute values in Community Web systems usually appear as collections of short strings. Such strings are usually distinguished by separators. Each user may employ diverse separators (such as '>', '/', ':', ';',) to delimit short strings.

Our inverted index distinguishes these short strings and creates a separate entry for each of them. A string match during query processing can be satisfied either in an *exact* or a *fuzzy* manner. Exact string matching is straightforward. For the case of fuzzy matching, we still wish to take advantage of lexicographic order for fast query processing. Thus, we say that a query string s_q of length L and an indexed string s_i satisfy a *fuzzy string match* when there is an *exact* match of the first half of the query string and a similarity between their other parts. Thus, if $\mathsf{prefix}(K, s)$ is the length-K prefix of string s, we define a fuzzy mach as:

$$\mathsf{prefix}\left(\left\lceil \frac{L}{2} \right\rceil, s_q\right) = \mathsf{prefix}\left(\left\lceil \frac{L}{2} \right\rceil, s_i\right) \wedge \;\; \mathsf{suffix}\left(\left\lfloor \frac{L}{2} \right\rfloor, s_q\right) \approx \mathsf{suffix}\left(\left\lfloor \frac{L}{2} \right\rfloor, s_i\right)$$

Where $\approx$ denotes an approximate string similarity measure. According to this kind of fuzzy matching, short strings like 'accessories' would match with 'accessorize' and 'accessory', but not with 'access, windows version'. In the case of text match, we define the fuzzy match score between two text values s_i and s_j as:

$$\text{Score}(s_i, s_j) = N/min(len(s_i), len(s_j))$$

where N is the number of matched words in s_i and s_j, $len(s_i)$ is the number of words in text value s_i and $len(s_j)$ is the number of words in text value s_j. We set a threshold τ, when $\text{Score}(s_i, s_j) \geq \tau$, we say s_i matches s_j. This rule of thumb operates well in practice, allowing for the identification of related strings with tolerance to orthographic and terminological variations.

We distinguish four different types of queries that CW2I can process, based on their exploitation of inverted indexes, unified inverted index, and fuzzy string matching, as follows. A general *complex query*, covering all four types, is defined as follows.

$$Q\{p_1(G_1), p_2(G_2), \ldots, p_n(G_n), r_1(G_{r_1}, G'_{r_1}), \quad r_2(G_{r_2}, G'_{r_2}), \ldots, r_m(G_{r_m}, G'_{r_m})\}$$

where $p_i(G_i)$ is set of (select) predicates that define a group of tuples G_i and $r_i(G_{r_i}, G'_{r_i})$ is a (join) relation among the tuples in groups G_{r_i}, G'_{r_i} which satisfy their respective predicates. For instance, consider the query “find the black-color jewelery and purple-color jewelery such that their price difference is less than 10$”. Then G_1 is ‘black color jewelery’, hence the predicate that defines this group is

$$\mathsf{Product} = \textit{jewelery} \wedge \mathsf{Color} = \textit{black}$$

Likewise, G_2 is ‘purple color jewelry’, hence the predicate that defines it is

$$\mathsf{Product} = \textit{jewelery} \wedge \mathsf{Color} = \textit{purple}$$

The relation that should be satisfied among any item $t_1 \in G_1$ and any item $t_2 \in G_2$ is

$$|t_1.\mathit{price} - t_2.\mathit{price}| \leq 10\$$$

We can then classify the basic join queries that satisfy this general-purpose definition in to four distinct classes as follows.

1. ***Exact* join query *without* keyword.** In such a query, the select predicates are all on most-frequent, *indexed* (i.e., having their own inverted index) attributes. The join relation operates on a *simple* (i.e., numerical or single-string) attributes (e.g., price, name, but not multiple-short-string or text attributes, on which no exact match can be done).

2. ***Exact* join query *with* keyword.** In this type of query, the select predicates may be defined on published attributes or may be just keyword-specified, referring to less-frequent attributes, lacking their own *inverted index*. The join relation is defined on a simple attributes, as in Type 1.

3. ***Fuzzy join* query *without* keyword.** In this case, the select predicates are on indexed attributes. However, the join operation is a *fuzzy join* on *text* or multiple-short-string attributes. For example, consider the query "Find a cellphone and a laptop of the same brand.". Then G1 is "cellphone", G2 is "laptop", the predicate defines G1 is:

   $$\mathsf{Product} = \textit{cellphone}$$

   Similarly the predicate defines G2 is:

   $$\mathsf{Product} = \textit{laptop}$$

   The join operation on G1 and G2 is:

   $$\textit{t1.brand} = \textit{t2.brand}$$

   Given that *brand* is a text attribute, the value of *brand* consists of several short strings. Thus the join operation on this attribute should not be done with exact match. We call this type of query *fuzzy join* query.

4. ***Fuzzy* join query *with* keyword.** A query of this type may have both a select predicate defined by a non-indexed attribute, as well as a *fuzzy* join operation on a *text* attribute.

A Type-1 query can be handled by our scheme as well as by other architectures suggested in previous research (e.g., [11, 26]). However, the last three types are not straightforwardly handled by other architectures, but can be managed by our CW2I system.

Due to the fact that CW2I separates text attributes into single words, and indexes these words, it can perform (fuzzy) select and join operation defined by predicates over such attributes. Thus, for example, a select operation may require that the word 'shirt' appears in the Product attribute value, or that the word 'linen' appears in a less-frequent attribute value. The set of tuples satisfying such a predicate can be determined using an appropriate inverted index of an appropriate attribute, or the unified inverted index.

Likewise, a *fuzzy join* condition may be defined as a *fuzzy string match* between the attribute values of two tuples. This can be processed by extracting all words in the text value of one tuple and performing, for each of these words, a *fuzzy lookup* on the inverted index of a specified attribute, if there be such, or, otherwise, on the unified inverted index.

4.4 Experimental Study

In this section we discuss experimental studies of the scalability and performance of CW2I scheme. We compare the I/O cost of CW2I in answering Type-1 queries with the straightforward horizontal storage scheme (HORIZ), vertical storage scheme (VERTI) and iVA-file Scheme(iVA). iVA-file scheme is designed to handle structured similarity query, here we set the threshold of estimated edit distance to 0 to answer exact matching query. The query processing time is recorded for Type-2, Type-3 and Type-4 queries which are not evaluated in existing systems. We study the performance and the scalability of CW2I on them.

4.4.1 Experiment Setup

In VERTI, we store each attribute as a separate table and build index on the tuple_id column. This is same as the direct index in CW2I. The difference of implementation between CW2I and VERTI is that we create one inverted index for each of the indexed attributes and a unified index for each of the other attributes. We build a B^+ tree index on the keyword column of the inverted index and the unified index. In HORIZ, we store the indexed attributes in one relational table and the other un-indexed attributes in the vertical storage format. We build a B^+ tree index on TID (tuple id) column. We set a 4 KB file page in memory for the index and the table file operations. Our experimental environment is a Intel Core2 Duo 1.8GHz machine with 2GB memory, 160GB hard disk, running Windows XP Professional with SP2.

4.4.2 Description of Data

We downloaded published data items from GoogleBase, and set up our experimental evaluation on this dataset. It consists of 30 thousands tuples which are described by 1319 attributes out of which 1217 are text attributes and others are numerical attributes. According to our statistics, the average number of attributes per tuple is 16. To test the storage cost of the four methods, we initially insert 10k tuples. We measure the additional disk space cost by incrementally inserting 5K tuples each time, as shown in Figure 4.5. We observe that the storage space linearly increases with the number of tuples inserted. CW2I consumes about 25% more disk space than VERTI and HORIZ. Since iVA has encoding vector list for each attribute and a full data table using interpreted storage schema, it introduces much more disk

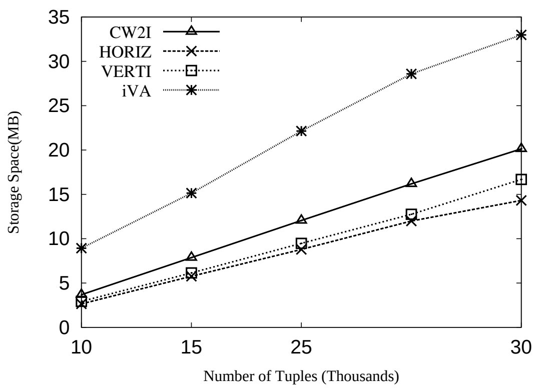


Figure 4.5: Disk Space Cost of the Three Methods.

space cost.

4.4.3 Description of Queries

We have discussed the four types of queries in Section 4.3. In our experiment we generate several queries for each query type. For Type-1 queries, i.e. exact join query without keyword, we compare the I/O cost of CW2I to VERTI , HORIZ and iVA. We describe the queries and the implementation details below.

Type-1 Queries

We first outline the Type-1 queries we have included in our experimental study.

- **Query 1.** Find the number of products that belong to the mp3 type. To process this query, CW2I system retrieves the result straightforwardly from product_type inverted index. In contrast, both of the horizontal and vertical storage scheme have to make a full table scan, either of a the complete horizontal table or of the table of product_type, and count the number of the qualifying tuples thereby. The iVA-file scheme need to scan the encoding vector list of product_type and the whole data table to answer this query.

- **Query 2.** Find the stores which sale both mp3 and computer products. Thus, this query defines two groups of items, group A being the group of tuples pertaining to mp3 products and group B being the group of tuples having to do with a computer product. The required join relation between these two groups is that the store_id of a tuple in A has to be the same as the store_id of a tuple in B.

  In the horizontal storage scheme, this query requires a self-join operation on attribute store_id. In the vertical storage scheme, we need to first retrieve the tuple_id of each mp3 product to create group A, as well as the tuple_id of every computer product to create group B, by accessing the in vertical table of the product_type attribute. Then we have to retrieve the corresponding store_id for each tupe_id in group A and B, using the vertical table of the store_id attribute. In the last step, we merge the two store_id lists to get the final results. In iVA-file Scheme, the encoding list of attribute product_type and the table are scanned to create group A and group B. Then a self join on attribute store_id is required to fetch the results. In contrast, in the CW2I scheme, we directly retrieve the two tuple_id lists A and B in the first step due to the availability of the inverted index of product_type attribute. Then we proceed as for the vertical storage scheme.

- **Query 3.** Find a black jewelry item and a purple jewelry item such that the difference of their price is less than 20$. Again, this query is similar to Query 2. However, now the selection predicates are on two attributes, namely on product_type, as well as on color. Through these two predicates, two groups of tuples are defined.

  CW2I derives these two groups via a fast merge-join of the respective tuple_id lists for product_type = *jewelry* and color = *black* or color = *purple*, respectively. The derivation of these groups is a more costly affair for HORIZ, VERTI and iVA. For the HORIZ scheme, it involves a full table scan whereby tuples that qualify on both attributes are identified. For the VERTI scheme, it requires the separate derivation of two tuple_id lists via full scan of the table for the product_type and color attributes. For the iVA scheme, it requires full scan of the encoding vector list for the product_type and color attributes and a full table scan to do the filtering step. Then the selection of price values and the join operation over them proceeds as it does for Query-2, with the difference that the join condition is now the inequality $|t_1.price - t_2.price| < 20\$$.

Type-2 Queries

We now define the Type-2 queries we have used for our experimental evaluation.

- **Query 1.** Find the stores which sell both an mp3 and a computer with 250GB disk. It defines a group A of mp3 items and a group of B computer items having the requested property. Again, it uses predicates on the product_type attribute, which are processed using the inverted index of that attribute.

  However, now the second group is also defined by the keyword-predicate specified by keyword '250GB'. The tuples satisfying this predicate are extracted by CW2I using the unified inverted index; the derived tuple_id list is then merge-joined with the tuple_id list of tuples satisfying the predicate product_type = *computer* to derive group B. These two groups of items are then joined on store_id, so that the stores selling both kinds of products are derived, in the same fashion as when processing a Type-1 query. The processing of this query is supported by our CW2I scheme, but not by other schemes for Community Web data management.

- **Query 2.** Find pairs of two 'ipods', such that one's price is less than 200$, the other's price is more than 200$, and the difference of their prices is less than 100$. Each of the two groups this query operates on are defined by a predicate on the price attribute as well as a keyword predicate, using the keyword 'ipod' on the unified inverted index. The join condition is on the price attribute as well. The processing steps of this query are the same as Type-1.

- **Query 3.** Find the stores that sale both Mahal's CD and Dorina's CD. It defines two groups and both of the two groups have selecting predicate on attribute product_type. Each of them has a keyword predicate and they are joined on attribute store_id.

- **Query 4.** Find thinkpad T41s and thinkpad T20s that the difference of their prices is less than 100$. This query has two keyword predicates (T41 and T20) which define two groups of items and a keyword predicate (Thinkpad) on both of the two groups. The items are joined on attribute price. The processing steps of this query are the same as Type-1 queries.

Type-3 Queries

We have also defined two Type-3 queries, as follows.

- **Query 1.** Find brands that make both cellphones and laptops. The select predicates are on the product_type attribute, which has its own inverted index, hence they are processed straightforwardly. However, the join operation is defined on attribute brand. The values of this attribute are short strings, so we have to conduct the matching based on string similarity. We retrieve the tuple_id lists of cellphone and laptop using the inverted index on product_type and obtain two item_id lists, A and B, of the two groups.

We measure the number of keywords defined for all items in each group. The items in the cellphone group (A) turn out to collectively contain less keywords. Thus, we first retrieve the string value of brand for each item_id α_i in group A. We extract all keywords in such a string value and obtain the item_id list for each of these keywords using the inverted index on brand; we merge-join these lists to get list $\mathcal{L}_i$. Finally, we merge-join the item_id list $\mathcal{L}_i$ so obtained (i.e., the list of all items whose brands match with the cellphone brand of α_i) with the item_id list of the 'laptop' group B, and desired results are so obtained.

- **Query 2.** Find types of products such that there exists both at least one *red* item and at least one *blue* item of that product_type. Thus, this query is defined by two select predicates on attribute color and a join operation is on the attribute product_type. Given that each value of the product_type attribute contains several short strings, the equi-join match between different values of this attribute is based on their similarity; therefore this is also a Type-3 Query. The processing proceeds as in the preceding discussion for Query 1.

Type-4 Queries

Lastly, we have also included two queries of the most complex type in our typology, Type-4, in our study, which are outlined below.

- **Query1.** Find *hardcover* books and Mahal's CDs with the same allowed form of payment. Thus, the two groups it joins are defined by two select predicates on the product_type attribute, *as well as* two *keyword* predicates, with keyword 'hardcover' and 'Mahal' respectively. The join operation is on the attribute payment, and also has to be processed in a keyword-oriented fashion, as for Type-3 queries discussed above.

- **Query2.** Find products of material ‘stone’ and products of material ‘silver’, having the same product_type. Thus, it has to define two groups of items using keyword-only predicates on keywords ‘stone’ and ‘silver’ respectively. Then, these two groups have to joined on the product_type attribute, in a keyword-oriented fashion again.

4.4.4 Results

In this section we report the results of our experimental study with the queries described in Section 4.4.3. We use progressively larger prefixes of the experimental data set while measuring the number of I/Os of each Type-1 query and recording the execution time for each Type-2, Type-3 and Type-4 query. We use logarithmic y-axes for the execution time in each case. Typically, the performance of our CW2I scheme is 10 or more times better than HORIZ and VERTI and is 1000 more times better than iVA scheme in terms of I/Os.

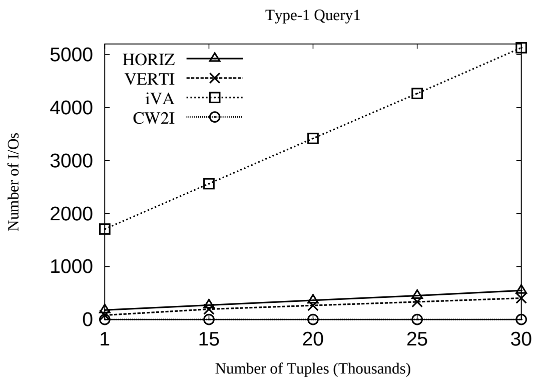


Figure 4.6: I/O Cost, Type-1 Query 1

Figure 4.6 shows the I/O cost for Type-1 Query 1. In this case, CW2I largely outperforms the prototype implementation of HORIZ, VERTI and iVA. Thanks to the availability of an inverted index in CW2I, only one B^+-tree search is required in order to find the number of qualifying items. In contrast, a whole table scan is necessitated by both HORIZ and VERTI, while an encoding vector list scan together with a table scan are required by iVA.

Given that the size of the vertical table is smaller than that of the horizontal table, the number of I/Os of VERTI is noticeably smaller than that of HORIZ especially when the number of tuples become larger. The number of I/Os of iVA is significantly larger than that of the other three, since besides the scan of encoding vector list a full table scan is needed to filter the false positive results. Besides, the growth of I/Os with the size of the data set is perceptible for both HORIZ ,VERTI and iVA; on the other hand, the I/Os remains relatively stable for CW2I, as new relevant items are inserted in the idlist of the inverted index and can still be easily retrieved without substantial overhead.

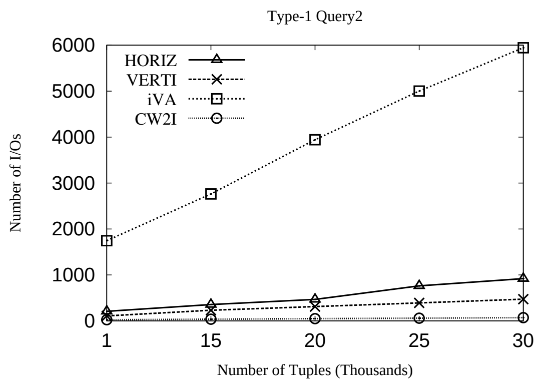


Figure 4.7: I/O Cost, Type-1 Query 2

Our results for Query 2 of Type-1 is shown in Figure 4.7. We observe that CW2I outperform HORIZ, VERTI and iVA, while VERTI is slightly better than HORIZ and iVA is much worse than VERTI and HORIZ. Thanks to the inverted index built on attribute product_type, CW2I gets the tuple_ids of mp3 and computer with just serval I/O. On the other hand, VERTI has to scan the vertical table of attribute product_type and random I/Os are required for fetching store_id in the vertical table of attribute store_id. As far as HORIZ is concerned, a whole table scan is needed to do the selection and a self join on attribute store_id is conducted on the intermediate results in this case. In terms of iVA, an encoding vector list scan and a whole table scan is required for the selection and a self join on attribute store_id is execute on the selection results.

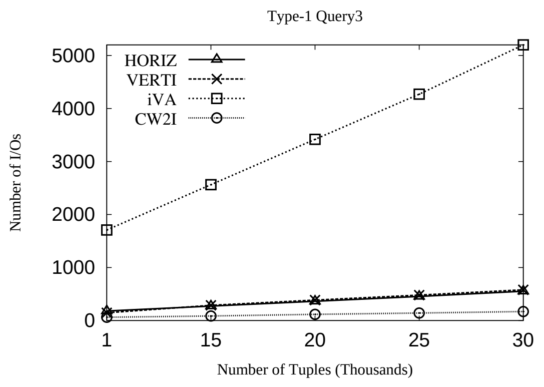


Figure 4.8: I/O Cost, Type-1 Query 3

Figure 4.8 depicts our results on Type-1 Query3. Observably, CW2I gains a significant advantage over both HORIZ, VERTI and iVA. The underlying cause of this efficiency advantage is the same as in our preceding analysis for Query 2. However, now this advantage is more perceptible within the examined data set sizes.

Figure 4.9 illustrates the results of the Type-2 queries. It is indicated that CW2I scales well for the queries of Type-2 with the increase of the dataset size. Q1 and Q3 are almost identical and they both join on attribute store_id, while Q2 and Q4 join on attribute price. The efficiency gap is due to the fact that the cardinality of attribute price is smaller than attribute store_id, so the second group gets better scalability. Moreover, the difference between the results of Q2 and Q4 is due to the

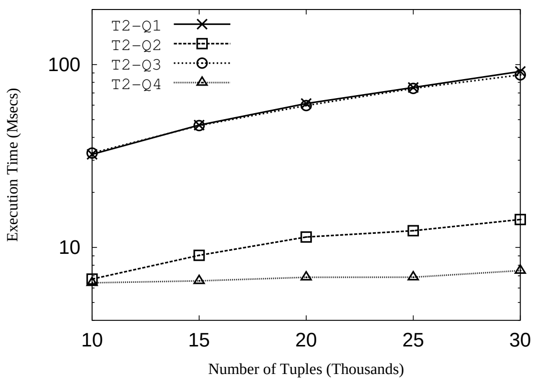


Figure 4.9: Execution time, Type-2.

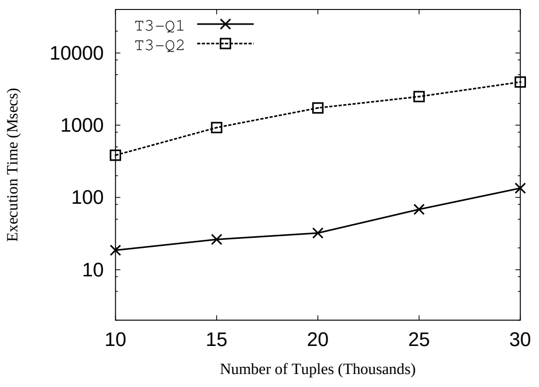


Figure 4.10: Execution time, Type-3.

difference of the sizes of the intermediate results.

Figure 4.10 and Figure 4.11 shows our results for queries of Type-3 and Type-4 respectively. CW2I scales well for the queries of these two types. Still, compared to Type-2 queries (Figure 4.9), those of Type-3 and Type-4 are less scalable. This difference is due to the fact that the join operations of Type-3 and Type-4 are fuzzy joins, which cost more than the exact join operations of Type-2. Besides, we observe differences among the queries of Type-3 and Type-4 themselves, which are due to variations in result set sizes.

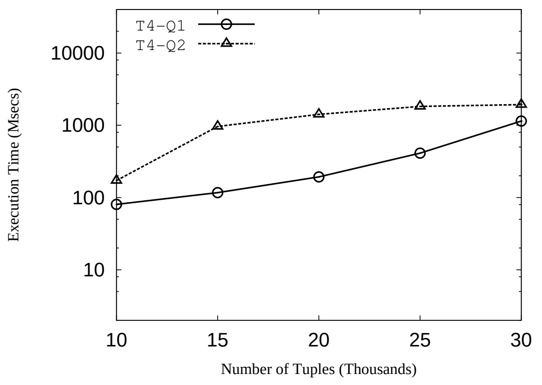


Figure 4.11: Execution time, Type-4.

4.5 Summary

In this chapter we have proposed an architecture for the management of sparse and wide data in Community Web systems that can efficiently handle complex queries. Our approach combines the benefits of an inverted indexing scheme with those of the direct-access feasibility provided by SWTs. A thorough experimental comparison, based on real-word data and practical queries, illustrates the advantages of our scheme compared to other approaches for community web data management, while our CW2I indexing scheme provides an attractive solution to the storage problem as well.

CHAPTER 5

Conclusion

The growing popularity of 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 neither structured similarity query nor complex query efficiency.

In CWMSs, past research has proposed SWT as a platform for storage of community data. Yet, such tables fail to provide an efficient storage architecture, as they typically include thousands of attributes, with most of them being undefined for each tuple. To enhance the query interfaces in such CWMSs, structured similarity query processing and complex query processing call for well designed index structures.

5.1 Summary of Main Findings

This section summarizes the main findings of the thesis. We discuss the contributions to structured similarity queries and complex queries in CWMSs.

5.1.1 Structured Similarity Query Processing

Existing studies on community based applications mainly focus on the efficient storage of the sparse table which is used as the basic structure to capture diverse datasets entered by users, and so far only one index method, namely the inverted index, has been evaluated for enhancing the query efficiency. In this thesis, we have proposed 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, comparing the performance of iVA-file and other implementations confirms that the iVA-file is an 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.

5.1.2 Complex Query Processing

While there has been long stream of research on keyword based retrieval, little attention has been paid to complex query processing. In this thesis, we have proposed an architecture for the management of sparse and wide data in Community Web systems that can efficiently handle complex queries. Our proposal combines two hitherto distinct approaches. On the one hand, we employ a vertical representation scheme, whereby each attribute, no matter how frequently defined, obtains its own column-oriented *direct* index. On the other hand, we utilize an *inverted* indexing scheme, whereby the tuples that define a *more frequent* attributes are indexed *by value*. Furthermore, we propose a *unified inverted* indexing scheme that gathers together all *less frequent* in a single keyword-oriented index. This additional index facilitates schema-agnostic keyword search that fits the nature of such less frequent attributes. We have defined four distinct types of join queries that our CW2I system can naturally process. Our experimental study using real dataset, comparing the performance of iVA-file and other implementations confirms that CW2I enables fast and scalable processing of complex queries over Community Data more efficiently than systems based on a monolithic vertical-oriented or horizontal-oriented representation, and gains an advantage of several orders of magnitude over them in our prototype implementation.

5.2 Future Work

While this thesis has presented efficient approaches to structured similarity query processing and complex query processing, a number of issues need to be further investigated:

- First, iVA-file proposed to indexing community data for structured similarity query processing provides an approximation of the data file and it has to be scanned during query processing. It is plausible to structure iVA-File as a tree structure to avoid full scanning. Further, optimization algorithms could be designed for more efficient pruning based on some constraints.

- Second, we need to clearly define conditions on when to apply inverted indexing for a certain attribute in CW2I. This decision can be made based on both the frequency of the said attribute as well as the query workload. The system can fine tune and decide on the indexing based on available statistics and trend.

- Third, the iVA-file index structure and CW2I index structure are isolated. iVA-file index and query processing algorithms are invoked for structured similarity query while CW2I is for complex query. The issue of query optimization should be examined to take full advantages of both index structures to maximize the benefits. Further, it would be good to design an index that serves both purposes.

BIBLIOGRAPHY

[1] ebay. Online at: http://www.ebay.com/.

[2] Facebook. Online at: http://www.facebook.com/.

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

[4] Google base. Online at: http://base.google.com/.

[5] Wikipedia. Online at:http://www.wikipedia.org/.

[6] Windows Live Spaces. Online at:http://spaces.live.com/.

[7] Daniel J. Abadi. Column stores for wide and sparse data. In *CIDR*, 2007.

[8] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[9] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[10] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. Materialization strategies in a column-oriented dbms. In *ICDE*, pages 466–475, 2007.

[11] Rakesh Agrawal, Amit Somani, and Yirong Xu. Storage and querying of e-commerce data. In *VLDB*, pages 149–158, 2001.

[12] Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, pages 359–370, 2004.

[13] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, and Dimitris Plexousakis. On storing voluminous RDF descriptions: The case of web portal catalogs. In *WebDB*, 2001.

[14] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. The ICS-FORTH RDFSuite: managing voluminous RDF description bases. In *SemWeb*, 2001.

[15] Tan Apaydin, Guadalupe Canahuate, Hakan Ferhatosmanoglu, and Ali Saman Tosun. Approximate encoding for direct access and query processing over compressed bitmaps. In *VLDB*, pages 846–857, 2006.

[16] Dave J. Beckett. The design and implementation of the redland RDF application framework. In *WWW*, pages 449–456, 2001.

[17] Jennifer L. Beckmann, Alan Halverson, Rajasekar Krishnamurthy, and Jeffrey F. Naughton. Extending RDBMSs to support sparse datasets using an interpreted attribute storage format. In *ICDE*, page 58, 2006.

[18] Alexander Behm, Shengyue Ji, Chen Li, and Jiaheng Lu. Space-constrained gram-based indexing for efficient approximate string search. In *ICDE*, pages 604–615, 2009.

[19] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *ICDT*, pages 217–235, 1999.

[20] Peter A. Boncz and Martin L. Kersten. MIL primitives for querying a fragmented world. *VLDB J.*, 8(2):101–119, 1999.

[21] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: hyper-pipelining query execution. In *CIDR*, 2005.

[22] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *ISWC*, 2002.

[23] Guadalupe Canahuate, Michael Gibas, and Hakan Ferhatosmanoglu. Indexing incomplete databases. In *EDBT*, pages 884–901, 2006.

[24] Chee Yong Chan and Yannis E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *SIGMOD*, pages 215–226, 1999.

[25] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.

[26] Eric Chu, Jennifer L. Beckmann, and Jeffrey F. Naughton. The case for a wide-table approach to manage sparse relational data sets. In *SIGMOD*, pages 821–832, 2007.

[27] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435, 1997.

[28] William W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *SIGMOD*, pages 201–212, 1998.

[29] George P. Copeland and Setrag Khoshafian. A decomposition storage model. In *SIGMOD*, pages 268–279, 1985.

[30] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[31] Stephen Harris and Nicholas Gibbins. 3store: Efficient bulk RDF storage. In *PSSS*, 2003.

[32] Stephen Harris and Nigel Shadbolt. SPARQL query processing with conventional relational database systems. In *SSWS*, 2005.

[33] Hao He, Haixun Wang, Jun Yang 0001, and Philip S. Yu. BLINKS: ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.

[34] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

[35] H. V. Jagadish, Nick Koudas, and Divesh Srivastava. On effective multi-dimensional indexing for strings. In *SIGMOD*, pages 403–414, 2000.

[36] Alan J. Kent, Ron Sacks-Davis, and Kotagiri Ramamohanarao. A signature file scheme based on multiple organizations for indexing very large text databases. *JASIS*, 41(7):508–534, 1990.

[37] Setrag Khoshafian, George P. Copeland, Thomas Jagodis, Haran Boral, and Patrick Valduriez. A query processing strategy for the decomposed storage model. In *ICDE*, pages 636–643, 1987.

[38] Min-Soo Kim, Kyu-Young Whang, Jae-Gil Lee, and Min-Jae Lee. n-Gram/2L: A space and time efficient two-level n-gram inverted index structure. In *VLDB*, pages 325–336, 2005.

[39] Grzegorz Kondrak. N-Gram similarity and distance. In *SPIRE*, pages 115–126, 2005.

[40] Hongrae Lee, Raymond T. Ng, and Kyuseok Shim. Extending q-grams to estimate selectivity of string matching with low edit distance. In *VLDB*, pages 195–206, 2007.

[41] Chen Li, Jiaheng Lu, and Yiming Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[42] Chen Li, Bin Wang, and Xiaochun Yang. VGRAM: improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314, 2007.

[43] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, pages 903–914, 2008.

[44] Fang Liu, Clement T. Yu, Weiyi Meng, and Abdur Chowdhury. Effective keyword search in relational databases. In *SIGMOD*, pages 563–574, 2006.

[45] Li Ma, Zhong Su, Yue Pan, Li Zhang, and Tao Liu. RStar: an RDF storage and query system for enterprise resource management. In *CIKM*, pages 484–491, 2004.

[46] David Maier and Jeffrey D. Ullman. Maximal objects and the semantics of universal relation databases. *ACM Trans. Database Syst.*, 8(1):1–14, 1983.

[47] Michele Missikoff. A domain based internal schema for relational database machines. In *SIGMOD*, pages 215–224, 1982.

[48] Shamkant B. Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical partitioning algorithms for database design. *ACM Trans. Database Syst.*, 9(4):680–710, 1984.

[49] Thomas Neumann and Gerhard Weikum. RDF-3X: A RISC-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.

[50] Beng Chin Ooi, Cheng Hian Goh, and Kian-Lee Tan. Fast high-dimensional data search in incomplete databases. In *VLDB*, pages 357–367, 1998.

[51] Beng Chin Ooi, Bei Yu, and Guoliang Li. One table stores all: Enabling painless free-and-easy data publishing and sharing. In *CIDR*, pages 142–153, 2007.

[52] Zhengxiang Pan and Jeff Heflin. DLDB: extending relational databases to support semantic web queries. In *PSSS*, 2003.

[53] Rajesh Raman, Miron Livny, and Marvin H. Solomon. Matchmaking: distributed resource management for high throughput computing. In *HPDC*, page 140, 1998.

[54] Uri Shaft and Raghu Ramakrishnan. Theory of nearest neighbors indexability. *TODS*, 31(3):814–838, 2006.

[55] Michael Stonebraker. The case for partial indexes. *SIGMOD Rec.*, 18(4):4–11, 1989.

[56] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-Store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[57] Erkki Sutinen and Jorma Tarhio. On using q-Gram locations in approximate string matching. In *ESA*, pages 327–340, 1995.

[58] Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Commun. ACM*, 22(11):606–611, 1979.

[59] Esko Ukkonen. Approximate string matching with q-grams and maximal matches. *Theor. Comput. Sci.*, 92(1):191–211, 1992.

[60] Julian R. Ullmann. A binary n-Gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. *The Computer Journal*, 20(2):141–147, 1977.

[61] Raphael Volz, Daniel Oberle, Steffen Staab, and Boris Motik. KAON SERVER: A semantic web management system. In *WWW (Alternate Paper Tracks)*, 2003.

[62] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[63] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, pages 194–205, 1998.

[64] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[65] Harry K. T. Wong, Jianzhong Li, Frank Olken, Doron Rotem, and Linda Wong. Bit transposition for very large scientific and statistical databases. *Algorithmica*, 1(3):289–309, 1986.

[66] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006.

[67] Xiaochun Yang, Bin Wang, and Chen Li. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *SIGMOD*, pages 353–364, 2008.

[68] Cui Yu, Beng Chin Ooi, Kian-Lee Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB*, pages 421–430, 2001.

[69] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23(4):453–490, 1998.