

D-Tree: A Multi-Dimensional Indexing Structure for Constructing Document Warehouses*

FRANK S. C. TSENG AND WEN-PING LIN

*Department of Information Management
National Kaohsiung First University of Science and Technology
Kaohsiung, 811 Taiwan
E-mail: imfrank@ccms.nkfust.edu.tw*

Document warehouses, unlike traditional document management systems, contain extensive semantic information about documents, cross-document feature relations, and document grouping or clustering, thus providing an accurate and efficient access to business intelligence information. Since documents are multi-dimensional in nature, we claim that traditional indexing methods are not really suitable for document warehousing. In this paper, we propose an indexing structure, called the D-tree, which can facilitate the construction of document cubes. We formally present the related definitions, the design of its storage structure and related algorithms for D-trees. The above are essential for establishing an infrastructure for combining text processing methods with numeric OLAP processing technologies. Hopefully, the proposed combination of data warehousing and document warehousing will be an important kernel for knowledge management and customer relationship management applications.

Keywords: data warehousing, document warehousing, knowledge management, multi-dimensional access method, OLAP

1. INTRODUCTION

Data warehousing [20] and data mining techniques [15] are gaining in popularity as organizations come to realize the benefits of being able to perform multi-dimensional analyses of accumulated historical business data to help contemporary administrative decision-making [14, 15, 21, 29]. This is inspiring enterprises to obtain useful business intelligence (BI) from both internal and external data. Business intelligence is supposed to provide decision-makers with the tactical and strategic information they need to understand, manage, and coordinate operations and processes in their organizations.

However, many of these efforts have only touched the tip of the information iceberg. While techniques for data warehouses, multi-dimensional models, on-line analytical processing (OLAP), or even ad hoc reports have served enterprises well, they have not completely addressed the full scope of business intelligence. It is believed that only about 20% of business intelligence can currently be extracted from formatted data stored in

Received March 30, 2004; revised September 22 & December 22, 2004; accepted December 30, 2004.

Communicated by Suh-Ying Lee.

* A shorter version of this paper was presented at the 20th Workshop on Combinatorial Mathematics and Computation Theory, Taiwan, 2003. ([42] in Chinese)

* This research was partially supported by the National Science Council, Taiwan, R.O.C., under Contract No. NSC-91-2416-H-327-005.

relational databases [17]. The remaining 80% is hidden in unstructured or semi-structured documents. This includes market survey reports, project status reports, meeting records, customer complaints, e-mails, patent application sheets, and advertisements of competitors, which are all recorded in documents.

Therefore, knowledge workers, managers, and executives still have to spend much time reading dozens, if not hundreds, of various types of electronic documents spread over the Internet. There is just too much text to digest in daily life without the use of powerful tools. As a result, some relevant documents may be ignored, and some irrelevant documents may be considered based on faulty intuition in the course of decision-making.

To alleviate these problems, Grigsby [12], McCabe *et al.* [26] and Sullivan [33] have proposed that documents be properly *warehoused* according to some well-defined concepts so as to expand the scope of business intelligence to include textual information. Although many research works on text mining have been conducted (readers are referred to [19, 22-24, 28, 34]), issues regarding document warehouses have rarely been addressed. With document warehouses, the documents of enterprises can be well organized for effective analysis or feature extraction aimed at creating distilled and fruitful business intelligence.

Since documents usually contain diverse concepts, they are multi-dimensional in nature. Nevertheless, we claim that traditional indexing methods are not really suitable for document warehousing. This is because the index of a document cube is not only helpful for indexing, but can also serve as the *metadata* of the document cube itself. Although a multi-dimensional array can be employed to represent the index of a document cube, it is usually too costly as document cubes are usually sparse. In addition, a multi-dimensional array cannot support roll-up and drill-down operations directly. It must have a nested structure to support these functions.

In this paper, we will formally define and expand on the important concepts of document warehousing. Then, we will propose an indexing structure, called the *D-tree*, to facilitate the organizing of documents into document cubes, which together constitute a document warehouse. We hope that this infrastructure combined with text processing technologies can make data warehousing and document warehousing together an important kernel of knowledge management and customer relationship management applications.

Document warehousing also provides an important platform for on-line analytical processing (OLAP) at the text level for the interactive analysis of multidimensional documents of varying granularity, which facilitates effective text mining, integrates documents into a business intelligence infrastructure and provides a means of searching for and targeting specific information in the same way we now do numeric data.

Our paper is organized as follows. Section 2 will survey current multi-dimensional indexing structures. The design criteria obtained from these structures can be applied in our D-tree. In section 3, the important concepts of document warehousing will be presented, and then, based on these concepts, a formal definition of the D-tree will be given. In section 4, the storage structure and processes for constructing D-trees will be explored. We will also address query processing and discuss experiments which we have conducted. Finally, we will draw conclusions and outline future works in section 6.

2. RELATED WORKS

2.1 Indexing Structures for Multi-Dimensional Representations

Based on various types of data, indexing structures proposed for multidimensional representations can be classified as follows:

1. *Representing point data on multidimensional spaces.* The K-D-B-tree [30], LSD-tree [16] and hB-tree [25], for example, fall into this category.
2. *Representing spatial data on multidimensional spaces.* The R-tree [13], R⁺-tree [31] and SKD-tree [27], for example, are included in this category.
3. *Representing textual documents on multidimensional spaces.* So far, we have not found such indexing structure proposed in the literature. That was the motivation for developing the D-tree.

From the multidimensional point of view, since the relative positions of different points (or objects) are important information for spatial data management and need to be properly recorded, the first two categories have the same properties in terms of representing spatial data. However, for document processing and management, keywords are the most important objects. But the relative positions of the keywords in a document depend on the context, which is hard to capture through current natural language understanding technologies. For the on-line analytical processing over documents, the traditional structures for indexing spatial data may be too complex and unsuitable for textual documents. Therefore, we think that multidimensional structures for indexing textual documents should be re-explored.

2.2 Requirements for Multi-Dimensional Indexing Structures

Based on the properties of spatial data and their applications, a variety of requirements for multidimensional access methods have been proposed [10, 25, 30], and we follow these requirements in designing our D-tree. The requirements are as follows:

1. *Dynamics.* As data objects are inserted and deleted from the database in any given order, access methods should continuously keep track of the changes.
2. *Secondary/tertiary storage management.* Despite the growing size main memory, it is often not possible to hold a complete database in main memory. Therefore, access methods need to integrate secondary and tertiary storage in a seamless manner.
3. *Broad range of supported operations.* Access methods should not support just one particular type of operation (such as retrieval) at the expense of other tasks (such as deletion).
4. *Independence of the input data.* Access methods should remain efficient even when the input data are highly skewed. This point is especially important for data that are distributed differently along various dimensions.
5. *Simplicity.* Intricate access methods with special cases are often error-prone and, thus, not sufficiently robust for use in large-scale applications.

6. *Scalability*. Access methods should adapt well to the growth of the underlying database.
7. *Time efficiency*. Access methods should guarantee a logarithmic worst-case search performance for all possible input data distributions regardless of the insertion sequence. This worst-case performance should hold for any combination of attributes.
8. *Space efficiency*. An index should be small in size compared with the data to be addressed and, therefore, guarantee a certain degree of storage utilization.
9. *Concurrency and recovery*. In modern databases, where multiple users concurrently update, retrieve and insert data, access methods should provide robust techniques for transaction management without incurring a significant performance penalty.
10. *Minimum impact*. The integration of an access method into a database system should have a minimal impact on the existing parts of the system.

The rationale behind the D-tree is similar to that behind the R-tree [13], in the sense that the R-tree models spatial objects with overlapping subspaces, whereas the D-tree indexes textual documents with overlapping keywords, the structure of the D-tree is simpler and completely different from that of the R-tree.

3. D-TREE: A MULTI-DIMENSIONAL INDEXING STRUCTURE

3.1 Definitions

Although issues concerning document warehousing have been addressed in [12, 26, 33], there are still no formal definitions. In the following, we will define a *document*, *dimension*, *document tuple* and *document cube*. Then, based on these definitions, we will define the D-tree.

Definition 1 A *document* $T = \{k_1, k_2, \dots, k_i\}$ is a logical unit of text characterized by a set of keywords $\{k_1, k_2, \dots, k_i\}$.

Definition 2 A *dimension* D is a tree structure of m levels, $m \geq 1$, which is used to represent hierarchical relationships among the keywords in a set. A node in a dimension D is called a *member*, and each internal node contains a special child, called *summary member*, indicated by an ‘*’, which is used to define the other children of the internal node.

The summary member of the dimension root is used to represent the whole dimension. It is used in contemporary data warehousing systems to provide flexible query processing when the dimension member is not explicitly specified in user queries. The other summary members are used to represent the other children of the internal node.

The keywords in a dimension are not limited to those contained in document contents. Any property or metadata of a document file (e.g., such as those defined in the Dublin Core Metadata Element Set [8]) can also be regarded as a keyword in a dimension for constructing document cubes. Furthermore, if documents are organized into predefined categories, the category hierarchy to which a document belongs can also be regarded as a dimension. That is, text is not unstructured as is often assumed and as

pointed out by Sullivan [33]. The concept of a dimension can be employed to model the structure inherently hidden in text.

To simplify our discussion, we will use only ordinary dimensions in the following examples.

Definition 3 For a *dimension* D , the i th-level member set, denoted $D(i)$, is defined as $D(i) = \{a \mid a \text{ is a member in the } i\text{th level of } D, \text{ but } a \text{ is not a summary member}\}$. In addition, we use $D(0)$ to denote the union of all non-summary members in D , which is the union of all i -th level member sets in D . That is, $D(0) = \cup_{1 \leq i \leq h} D(i)$, where h is the height of D .

In practice, a dimension can be represented by a relational table, where each level corresponds to an attribute in the relation and the attribute names are usually regarded as corresponding level names. To illustrate the above definitions, we will give an example below.

Example 1: Suppose there is a relation *Region* representing the regions of Taiwan as shown in Table 1 (a). Another alternative is shown in Table 1 (b). This relation can be used to construct a dimension, denoted R as depicted in Fig. 1, where the first level corresponds to the dimension itself, which is commonly denoted “(All Region),” and the second and third levels are derived from the attributes *Location*, and *City*, respectively. All starred nodes in Fig. 1 are summary members. That is, the summary member in the second level has the same meaning as *all regions in Taiwan*, which represents $\{\textit{South}, \textit{North}\}$. Also, the summary members under *South* and *North* mean the same as *South* and *North*, which denote $\{\textit{Tainan}, \textit{Kaohsiung}, \textit{Pingtung}\}$ and $\{\textit{Taipei}, \textit{Taoyun}, \textit{Hsinchu}\}$, respectively. By omitting all the summary members, we can redraw Fig. 1 as shown in Fig. 2. According to the illustration of dimension R , we know that $R(1) = \{(\textit{All Region})\}$, $R(2) = \{\textit{South}, \textit{North}\}$, and $R(3) = \{\textit{Tainan}, \textit{Kaohsiung}, \textit{Pingtung}, \textit{Taipei}, \textit{Taoyun}, \textit{Hsinchu}\}$, and $R(0) = \{(\textit{All Region}), \textit{South}, \textit{North}, \textit{Tainan}, \textit{Kaohsiung}, \textit{Pingtung}, \textit{Taipei}, \textit{Taoyun}, \textit{Hsinchu}\}$.

In Fig. 3, another dimension denoted P and representing the products of a company that manufactures consumer electronics, is concisely depicted. Both dimensions will be used in the following examples. \square

Table 1. A relation *Region* and its alternative for constructing dimension R .

(a)		(b)			
<i>Location</i>	<i>City</i>	<i>Tag</i>	<i>Parent</i>	<i>Level</i>	<i>Keyword</i>
South	Tainan	1	1	1	(All Region)
South	Kaohsiung	2	1	2	South
South	Pingtong	3	1	2	North
North	Taipei	4	2	3	Tainan
North	Taoyun	5	2	3	Kaohsiung
North	Hsinchu	6	2	3	Pingtong
		7	3	3	Taipei
		8	3	3	Taoyun
		9	3	3	Hsinchu

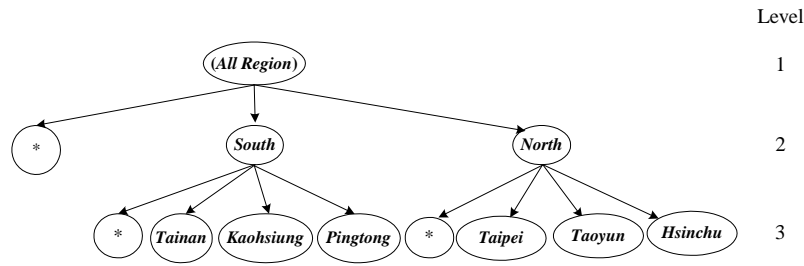


Fig. 1. An illustration of dimension R .

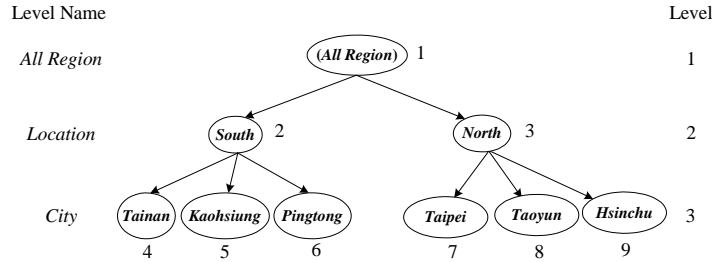


Fig. 2. A concise illustration of dimension R .

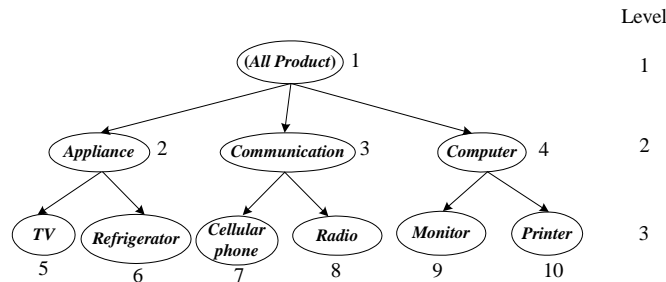


Fig. 3. A concise illustration of dimension P .

For a dimension D , there are two basic operations, called *drill-down* and *roll-up*. The *drill-down* operation involves the expanding of an internal node to obtain all of its children, and the *roll-up* operation involves the shrinking of a set of children to obtain their common parent. This can be further clarified by the following definitions.

Definition 4 For any two n -tuple of keywords $A = (a_1, a_2, \dots, a_i, \dots, a_n)$ and $B = (b_1, b_2, \dots, b_i, \dots, b_n)$ defined on n dimensions $(D_1, D_2, \dots, D_i, \dots, D_n)$, where a_i and $b_i \in D_i(0)$, we define B as a member of drilling down A along dimension D_i (or A as a member of rolling up B along dimension D_i), denoted $A \prec_i B$, if and only if there exists exactly one i , $1 \leq i \leq n$, such that b_i is a child of a_i in D_i , and $b_j = a_j$, for all $j \neq i$.

Definition 5 For a document T with a unique identifier id_T , a *document index* of T defined on n dimensions (D_1, D_2, \dots, D_n) is denoted $x = (id_T, K_T)$, where $K_T = (K_1, K_2, \dots, K_i, \dots, K_n)$ is an n -tuple of keyword sets such that each K_i contains a set of keywords, and for all keywords $k_{ij} \in K_i$, where $k_{ij} \in T$, and $k_{ij} \in D_i(0)$, for all $1 \leq i \leq n$.

For simplicity, the first and second components of a document index $x = (id_T, K_T)$ will be denoted x^1 and x^2 (i.e., $x^1 = id_T$ and $x^2 = K_T$), respectively. When all $|K_i| = 1$, the document index is also called a *base document index*, and each K_i can also be denoted by its only element for the sake of convenience. (That is, in such cases, a $K_T = (\{k_1\}, \{k_2\}, \dots, \{k_i\}, \dots, \{k_n\})$ can be abbreviated as $K_T = (k_1, k_2, \dots, k_i, \dots, k_n)$.) If there is at least one K_i , such that $|K_i| > 1$, and if the sizes of the other K_j 's are all equal to 1, then the document index is also called a *composite document index*. Finally, if there are some K_i , such that $|K_i| = 0$, then the document index is also called a *degenerate document index*. In the following, a degenerate document index with some $|K_i| = 0$ will be generalized by using the top level member set of the corresponding dimension, i.e., $D_i(1)$ or ‘*’, to replace the missing keyword set K_i .

Example 2: Following Example 1, suppose there is a complaint e-mail T issued by a customer as shown in Fig. 4. Then, a base document index of T defined on the above two dimensions (R, P) can be obtained as $x = (A0001, (\{Kaohsiung\}, \{TV\}))$, where A0001 is the unique identifier of T . □

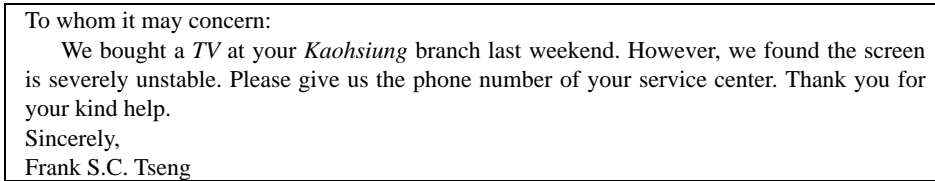


Fig. 4. A complaint e-mail issued by a customer (A0001).

The basic component of a document cube is called a *cell*, which is defined as follows.

Definition 6 A *cell* defined on n dimensions (D_1, D_2, \dots, D_n) is denoted $c = (t_c, X_c)$, where $t_c = (c_1, c_2, \dots, c_i, \dots, c_n)$, $c_i \in D_i(0) \cup \{*\}$, $1 \leq i \leq n$, and $X_c = \{x_1, x_2, \dots, x_j, \dots, x_m\}$ is a set of document indices of the form $x_j = (id_{T_j}, (K_1, K_2, \dots, K_n))$, where id_{T_j} is the unique identifier of some document T_j and $K_i \cap D_i(0) \neq \emptyset$, $1 \leq i \leq n$. The set of all such document unique identifiers id_{T_j} involved in the cell $c = (t_c, X_c)$ is denoted $ID(c) = \{x_j^1 \mid \forall x_j \in X_c\}$. That is, a document with a unique identifier in $ID(c)$ can be directly accessed from cell c .

Definition 7 A *cell* $c = (t_c, X_c)$, where $t_c = (c_1, c_2, \dots, c_i, \dots, c_n)$, defined on n dimensions (D_1, D_2, \dots, D_n) is called an m -d *cell*, $0 \leq m \leq n$, if and only if there are exactly m non-summary members c_i (i.e., $c_i \neq *$). If $m = n$ and $c_i \in D_i(h_i)$, where h_i is the height of D_i , for all $1 \leq i \leq n$, then c is also called a *base cell*; otherwise, c is called a *non-base cell*.

Definition 8 An n -dimensional i -d cell $a = ((a_1, a_2, \dots, a_n), X_a)$ is a *parent* of another n -dimensional i -d cell $b = ((b_1, b_2, \dots, b_n), X_b)$, if and only if the following conditions hold:

1. There exists exactly one k such that a_k is the parent of b_k in D_k and $a_l = b_l$ for all $l \neq k$, $1 \leq l \leq n$.

2. $ID(b) \subseteq ID(a)$, where $ID(a)$ and $ID(b)$ are the sets of all document unique identifiers involved in cells a and b , respectively.

Definition 9 A document cube $DC = (S, (D_1, D_2, \dots, D_n))$, where S is a set of documents defined on n dimensions (D_1, D_2, \dots, D_n) , is a cube composed of all cells $c_i = (t_{c_i}, X_{c_i})$ with $t_{c_i} \in \times_{1 \leq j \leq n} D_j(0)$ and $ID(c_i) \subseteq S$.

A sample illustration of a document cube $DC = (S, (R, P, T))$ is shown in Fig. 5, where R and P represent the aforementioned dimensions Region and Product, respectively. Here, we assume that T is a dimension representing Time.

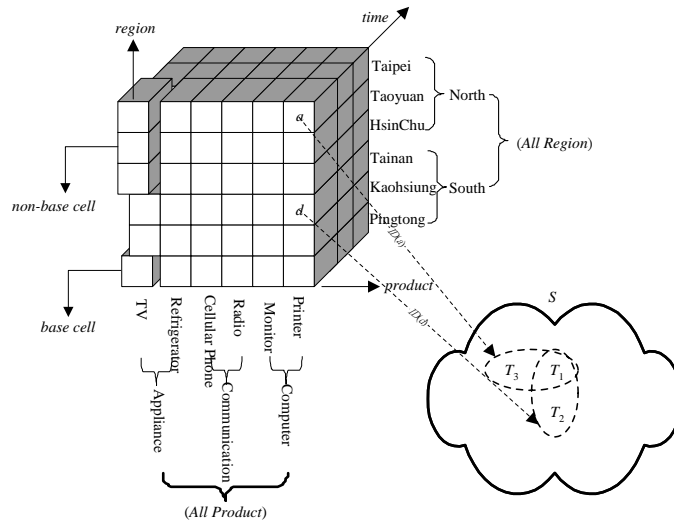


Fig. 5. A sample illustration of a document cube.

A document cube can be implemented based on the concept of the D -tree, which will be formally defined in Definition 10. In a complete D -tree, leaf nodes and non-leaf nodes are used to represent the base cells and non-base cells of a document cube, respectively. These nodes are linked according to the parent-child relationships defined in Definition 8, such that all leaf nodes are located in the lowest level.

Definition 10 A D -tree is a tree structure for implementing a document cube $DC = (S, (D_1, D_2, \dots, D_n))$. The basic components of a D -tree are as follows:

1. A set of nodes, including *leaf nodes* and *internal nodes*, which, respectively, represent *base cells* and *non-base cells* in the document cube. A non-base cell $n = (t_n, X_n)$ is represented by an internal node $I = (t_n, X_n, l_n)$ in a D -tree, where l_n is a set of links (defined below) linking to its children, and a base cell $b = (t_b, X_b)$ is represented by a leaf node $L = (t_b, X_b, l_b)$, where l_b is an empty set. Since a leaf node has no children, it can be abbreviated as $L = (t_b, X_b)$.

2. A set of links, which are defined as follows:

- ◆ A node $P = (t_P, X_P, l_P)$ has a child node $C = (t_C, X_C, l_C)$ (i.e., $C \in l_P$) if and only if t_C is a drilling down member t_P along some dimension D_i (i.e., $t_P \prec_i t_C$). Thus, if there exists exactly one c_i such that $t_C = (p_1, p_2, \dots, c_i, \dots, p_n)$, $t_P = (p_1, p_2, \dots, p_i, \dots, p_n)$, and c_i is a child of p_i in D_i , then P is a parent of C .

Accordingly, the set of all the document unique identifiers involved in a node $N = (t_N, X_N, l_N)$ in a D-tree is also denoted $ID(N) = \{x^1 \mid \forall x \in X_N\}$, where N can be an internal node or a leaf node. That is, a document with a unique identifier in $ID(N)$ can be directly accessed from node N .

Based on the above definitions, we have studied some properties of a complete D-tree. However, due to space limitations, readers are referred to [35].

3.2 Supported Query Types of D-trees

The query type for a document warehouse is most likely the same as that for a data warehouse. That is, for a $DC = (S, (D_1, D_2, \dots, D_n))$, users can provide a base or non-base cell $c = (c_1, c_2, \dots, c_i, \dots, c_n)$, $c_i \in D_i(0) \cup \{ '*' \}$, $1 \leq i \leq n$, and the system will then respond with document sets $S_c \subseteq S$ indexed by c . Based on the current query results, users can further issue roll-up or drill-down operations along some dimensions as well. For example, for the document cube shown in Fig. 5, managers can issue a base cell $c_b = ('TV', 'Taipei', 'Q4, 2004')$ to obtain a subset of documents containing information regarding *Taipei* customers who purchased *TVs* in *Q4, 2004*, or a non-base cell $c_n = ('TV', '*', 'Q4, 2004')$ to obtain a subset of documents containing information regarding *all* customers who purchased *TVs* in *Q4, 2004*.

To support various types of queries in data warehouses, MDX (Multi-Dimensional eXpression) [32] has been proposed to provide multi-dimensional OLAP (On-Line Analytical Processing) capabilities. MDX has been widely adopted and supported by Applix, Microsoft, Microstrategy, Whitelight, SAS and SAP [32]. In [36], we extended the constructs of MDX to MD²X (Multi-Dimensional Document eXpression) in order to include more features for querying document warehouses.

4. IMPLEMENTATION OF A D-TREE

According to Definitions 9 and 10, since the size of $\times_{1 \leq j \leq n} D_j(0)$ is factorial, a complete D-tree for a document cube generally consists of a huge number of nodes, which grows exponentially and is not practical for business applications. However, as we pointed out in section 1, document cubes are usually sparse, and it is not necessary to generate all nodes for a D-tree. We will now present a feasible implementation of a D-tree in the following.

4.1 The Storage Structure of a D-Tree

In fact, the best storage structure for a D-tree is based on the use of relational or object-relational databases. In Table 1, we show two representations of the dimension Re-

gion. Both representations can be easily obtained from each other through conversion. For example, to convert Table 1 (b) to Table 1 (a), the SQL statement can be employed:

```
select R1.keyword as Level2_name, R2.keyword as Level3_name
from Di R1, Di R2
where R1.Level = 2 and R1.Tag = R2.Parent
```

The structure of Table 1 (a) is easier for people to understand, and that of Table 1 (b) is more efficient for computer processing. In the following, we assume that each dimension D_i is stored in the relation $D_i(\underline{Tag}, Parent, Level, Keyword)$, conforming to the structure of Table 1 (b), such that the underlined attribute Tag represents the primary key, and $D_i.Parent$ is a foreign key referencing $D_i.Tag$. For example, the dimensions Product and Time in Fig. 5 can be represented as shown in Fig. 6.

<u>Tag</u>	Parent	Level	Keyword
1	1	1	(All Product)
2	1	2	Appliance
3	1	2	Communication
4	1	2	Computer
5	2	3	TV
6	2	3	Refrigerator
7	3	3	Cellular Phone
8	3	3	Radio
9	4	3	Monitor
10	4	3	Printer

<u>Tag</u>	Parent	Level	Keyword
1	1	1	(All Time)
2	1	2	2003
3	1	2	2004
4	2	3	Q1, 2003
5	2	3	Q2, 2003
6	2	3	Q3, 2003
7	2	3	Q4, 2003
8	3	3	Q1, 2004
9	3	3	Q2, 2004
10	3	3	Q3, 2004
11	3	3	Q4, 2004

Fig. 6. Dimensions product and time represented by relations.

To construct a D-tree for a document cube $DC = (S, (D_1, D_2, \dots, D_n))$ with the best possible utilization of storage, we only have to take into account all the document indices obtained from all the document instances in S and convert them into *initial* D-tree nodes, which will be used to generate all *effective* D-tree nodes. This is because *non-effective* D-tree nodes are of no use for indexing the documents in S and can be neglected. This may save an extremely large amount of storage space, as the ratio $|S|/|\times_{1 \leq j \leq n} D_j(0)|$ is very small in practical applications.

A document index $x = (id_T, K_T)$, where $K_T = (K_1, K_2, \dots, K_i, \dots, K_n)$, can be used to generate $\{id_T\} \times (\times_{1 \leq i \leq n} K_i)$ as the set of initial D-tree nodes, which can be stored in the relation $DTree(Doc_id, D1_tag, D2_Tag, \dots, Di_Tag, \dots, Dn_Tag)$, where Doc_id is a foreign key that references the document id in S and each Di_Tag is a foreign key matching the primary key $D_i.Tag$ of dimension D_i . Here, we also regard S as being stored in the relation $S(Doc_id, file_path)$, where Doc_id is the primary key and $file_path$ is used to store the file paths or URIs of documents. We call this approach *dimensional modeling* for document warehousing and illustrate it in Fig. 7. This can be regarded as a counterpart of the approach proposed by Kimball [21] for traditional data warehousing. Through

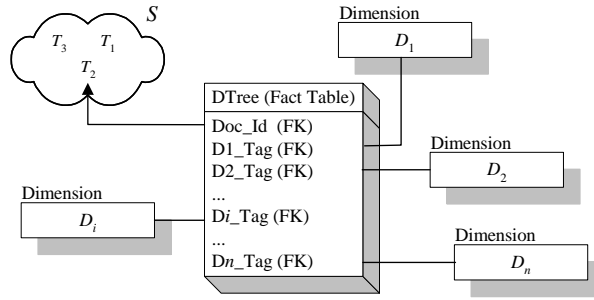


Fig. 7. Dimensional modeling of a document cube $DC = (S, (D_1, D_2, \dots, D_n))$.

dimensional modeling, all effective D-Tree nodes can be generated through the steps discussed in section 4.2.

Example 3: For the document cube $DC = (S, (R, P, T))$ illustrated in Fig. 5, a document index $x = (A0001, (\{North, Kaohsiung\}, \{TV, Radio\}, \emptyset))$ will be used to generate the initial D-tree nodes, which will be stored as tuples in the relation *DTree* shown in Fig. 8. Notice that, since x is a degenerate document index, the empty keyword set corresponding to the third dimension T will be replaced by $D_3(1) = \{(All\ Time)\}$ before generation proceeds. That is, x will be transformed into $(A0001, (\{North, Kaohsiung\}, \{TV, Radio\}, \{(All\ Time)\}))$ to generate the following set of tuples: $\{(A001, North, TV, (All\ Time)), (A001, North, Radio, (All\ Time)), (A001, Kaohsiung, TV, (All\ Time)), (A001, Kaohsiung, Radio, (All\ Time))\}$, which can be encoded into $\{(A001, 2, 5, 1), (A001, 2, 8, 1), (A001, 8, 5, 1), (A001, 8, 8, 1)\}$ based on the relationship between the foreign keys Di_Tag and their matching primary keys Di_Tag of each dimension, $1 \leq i \leq n$. \square

<i>Doc_id</i>	<i>D1_tag</i>	<i>D2_tag</i>	<i>D3_tag</i>
A001	2	5	1
A001	2	8	1
A001	8	5	1
A001	8	8	1

Fig. 8. Initial relation *DTree* generated for $DC = (S, (R, P, T))$.

4.2 The Generation of Effective D-Tree Nodes

Based on the above storage structure, according to Definition 10, the set of links of a D-tree can be inherently maintained based on the relationships between the foreign keys Di_Tag and their matching primary keys Di_Tag of each dimension, $1 \leq i \leq n$. Therefore, the parent nodes of all initial D-tree nodes (i.e., the tuples in the initial relation *DTree*) can be derived by joining $DTree.Di_tag$ with the attribute $Di.Tag$ of dimension Di , for all $1 \leq i \leq n$, and then projecting $Di.Parent$ to replace $DTree.Di_Tag$, together with the other $DTree.Dj_Tag$, for all $1 \leq j \leq n, j \neq i$. In general, the SQL statement for this task can be formulated as follows:

```

select distinct Doc_id, D1.Parent, DT.D2_Tag, ..., DT.Di_Tag, ..., DT.Dn_Tag
from D1, DTree DT
where D1.Tag = DT.D1_Tag
union
select distinct Doc_id, DT.D1_Tag, D2.Parent, ..., DT.Di_Tag, ..., DT.Dn_Tag
from D2, DTree DT
where D2.Tag = DT.D2_Tag
union
...
select distinct Doc_id, DT.D1_Tag, DT.D2_Tag, ..., Di.Parent, ..., DT.Dn_Tag
from Di, DTree DT
where Di.Tag = DT.Di_Tag
...
union
select distinct Doc_id, DT.D1_Tag, DT.D2_Tag, ..., DT.Di_Tag, ..., Dn.Parent
from Dn, DTree DT
where Dn.Tag = DT.Dn_Tag

```

That is, the following SQL statement can be used to generate the parent nodes of the tuples shown in Fig. 8:

```

select distinct Doc_id, R.Parent, DT.D2_Tag, DT.D3_Tag
from Region R, DTree DT
where R.Tag = DT.D1_Tag
union
select distinct Doc_id, DT.D1_Tag, P.Parent, DT.D3_Tag
from Product P, DTree DT
where P.Tag = DT.D2_Tag
union
select distinct Doc_id, DT.D1_Tag, DT.D2_Tag, T.Parent
from Time T, DTree DT
where T.Tag = DT.D3_Tag

```

As the dimensions are all finite, when this statement is iteratively applied to the newly obtained nodes, there exists a *least fixed point* [37], such that at that point, no more new nodes can be generated. When the least fixed point is reached, all the effective nodes of the target D-tree can finally be obtained. The complete SQL commands for repeatedly generating the effective nodes can be packed into the following Transact-SQL stored procedure **Build_Dtree**.

```

CREATE PROCEDURE build_Dtree AS
declare @Initial_Num_of_Nodes int
declare @Generated_Num_of_Nodes int
select @Generated_Num_of_Nodes = 0

```

```

if exists(select * from sysobjects where id = object_id('TempDTree') and type = 'U')
    drop table TempDTree
select distinct * into TempDTree from Dtree
select @Initial_Num_of_Nodes = count(*) from TempDtree
while (@Initial_Num_of_Nodes <> @Generated_Num_of_Nodes) /* Is the least fixed point
    reached? */
Begin
    select @Initial_Num_of_Nodes = count(*) from TempDtree
    if exists(select * from sysobjects where id = object_id('New_Effective_Tuples') and type =
'U')
        drop table New_Effective_Tuples
    select distinct *
    into New_Effective_Tuples
    from (
        select distinct Doc_id, D1.Parent, DT.D2_Tag, ..., DT.Di_Tag, ..., DT.Dn_Tag
        from D1, DTree DT
        where D1.Tag = DT.D1_Tag
        union
        select distinct Doc_id, DT.D1_Tag, D2.Parent, ..., DT.Di_Tag, ..., DT.Dn_Tag
        from D2, DTree DT
        where D2.Tag = DT.D2_Tag
        union
        ...
        select distinct Doc_id, DT.D1_Tag, DT.D2_Tag, ..., Di.Parent, ..., DT.Dn_Tag
        from Di, DTree DT
        where Di.Tag = DT.Di_Tag
        ...
        union
        select distinct Doc_id, DT.D1_Tag, DT.D2_Tag, ..., DT.Di_Tag, ..., Dn.Parent
        from Dn, DTree DT
        where Dn.Tag = DT.Dn_Tag
    ) Tmp
    Truncate table TempDTree
    insert into TempDTree select distinct * from New_Effective_Tuples
    insert into DTree select * from TempDTree
    select @Generated_Num_of_Nodes = count(*) from New_Effective_Tuples
end
/* Remove temporary relations TempDtree and New_Effective_Tuples */
drop table TempDtree, New_Effective_Tuples
/* The following statements delete duplicate tuples */
select distinct * into TmpDtree from Dtree
drop table Dtree
exec sp_rename 'TmpDtree', 'Dtree'

```

4.3 Query Processing in a D-Tree

To support the query types addressed in section 3.2, the following SQL statement can be employed to generate the query result (i.e., the document set) indexed by a specific cell $c = (c_1, c_2, \dots, c_i, \dots, c_n)$:

```

select distinct Doc_id
from DTree
where D1_Tag = c1 and D2_Tag = c2 and Di_Tag = ci ... and Dn_Tag = cn

```

To support the drill-down operation, i.e., to generate a query result indexed by drilling down a specific cell $c = (c_1, c_2, \dots, c_i, \dots, c_n)$ along dimension D_i to obtain a set of cells $C = \{(c_1, c_2, \dots, b_i, \dots, c_n) \mid b_i \text{ is a child of } c_i \text{ in } D_i\}$, the following SQL statement can be used to find set C :

```

select distinct DT.D1_Tag as C1, DT.D2_Tag as C2, ..., Di.Tag as Bi, ..., DT.Dn_Tag as Cn
from Di, DTree DT
where Di.Parent = DT.Di_Tag and DT.D1_Tag = c1 and DT.D2_Tag = c2 and ...
DT.Di_Tag = ci ... and DT.Dn_Tag = cn

```

Then, we can obtain the drill-down result from the union of all the query results of cells in C .

To support the roll-up operation, i.e., to generate a query result indexed by rolling up a specific cell $c = (c_1, c_2, \dots, c_i, \dots, c_n)$ along dimension D_i to obtain the cell $p = (c_1, c_2, \dots, p_i, \dots, c_n)$, such that p_i is the parent of c_i in D_i , the following SQL statement can be issued to find cell p :

```

select distinct DT.D1_Tag as C1, DT.D2_Tag as C2, ..., Di.Parent as Pi, ..., DT.Dn_Tag as Cn
from Di, DTree DT
where Di.Tag = DT.Di_Tag and DT.D1_Tag = c1 and DT.D2_Tag = c2 and ...
DT.Di_Tag = ci ... and DT.Dn_Tag = cn

```

Then, we can obtain the roll-up result from the query result of cell p .

Example 4: Based on the situations discussed in Example 1, we will demonstrate the process of generating all effective nodes for the target D-tree of the document cube $DC = (S, (R, P))$, where R and P are as depicted in Figs. 2 and 3, respectively. To compare the storage utilization of the D-tree constructed with our approach, which contains effective nodes only, and that of the complete D-tree, we list the nodes of the complete D-tree in Table 2.

Now, suppose S contains three documents: T_1 (with $id_{T_1} = A001$), T_2 (with $id_{T_2} = A002$) and T_3 (with $id_{T_3} = A003$), from which three document indices can be generated as follows:

1. $x_1 = (A0001, (\{Kaohsiung\}, \{TV\}))$. This generates $(A001, Kaohsiung, TV)$ and can be stored in the relation $DTree$ as $(A001, 5, 5)$.
2. $x_2 = (A0002, (\{Tainan\}, \{Monitor, Printer\}))$. This generates $(A002, Tainan, Monitor)$ and $(A002, Tainan, Printer)$ and can be stored in the relation $DTree$ as $(A002, 4, 9)$ and $(A002, 4, 10)$, respectively.

Table 2. All n -tuples of keywords corresponding to dimensions R and P .

$t_1 = ((\text{All Region}), (\text{All Product}))$	$t_2 = ((\text{All Region}), (\text{Appliance}))$	$t_3 = ((\text{All Region}), (\text{Communication}))$
$t_4 = ((\text{All Region}), (\text{Computer}))$	$t_5 = (\text{South}, (\text{All Product}))$	$t_6 = (\text{North}, (\text{All Product}))$
$t_7 = ((\text{All Region}), (\text{TV}))$	$t_8 = ((\text{All Region}), (\text{Refrigerator}))$	$t_9 = ((\text{All Region}), (\text{Cellular Phone}))$
$t_{10} = ((\text{All Region}), (\text{Radio}))$	$t_{11} = ((\text{All Region}), (\text{Monitor}))$	$t_{12} = ((\text{All Region}), (\text{Printer}))$
$t_{13} = (\text{South}, (\text{Appliance}))$	$t_{14} = (\text{South}, (\text{Communication}))$	$t_{15} = (\text{South}, (\text{Computer}))$
$t_{16} = (\text{North}, (\text{Appliance}))$	$t_{17} = (\text{North}, (\text{Communication}))$	$t_{18} = (\text{North}, (\text{Computer}))$
$t_{19} = (\text{Tainan}, (\text{All Product}))$	$t_{20} = (\text{Kaohsiung}, (\text{All Product}))$	$t_{21} = (\text{Pingtung}, (\text{All Product}))$
$t_{22} = (\text{Taipei}, (\text{All Product}))$	$t_{23} = (\text{Taoyun}, (\text{All Product}))$	$t_{24} = (\text{Hsinchu}, (\text{All Product}))$
$t_{25} = (\text{South}, (\text{TV}))$	$t_{26} = (\text{South}, (\text{Refrigerator}))$	$t_{27} = (\text{South}, (\text{Cellular Phone}))$
$t_{28} = (\text{South}, (\text{Radio}))$	$t_{29} = (\text{South}, (\text{Monitor}))$	$t_{30} = (\text{South}, (\text{Printer}))$
$t_{31} = (\text{North}, (\text{TV}))$	$t_{32} = (\text{North}, (\text{Refrigerator}))$	$t_{33} = (\text{North}, (\text{Cellular Phone}))$
$t_{34} = (\text{North}, (\text{Radio}))$	$t_{35} = (\text{North}, (\text{Monitor}))$	$t_{36} = (\text{North}, (\text{Printer}))$
$t_{37} = (\text{Tainan}, (\text{Appliance}))$	$t_{38} = (\text{Tainan}, (\text{Communication}))$	$t_{39} = (\text{Tainan}, (\text{Computer}))$
$t_{40} = (\text{Kaohsiung}, (\text{Appliance}))$	$t_{41} = (\text{Kaohsiung}, (\text{Communication}))$	$t_{42} = (\text{Kaohsiung}, (\text{Computer}))$
$t_{43} = (\text{Pingtung}, (\text{Appliance}))$	$t_{44} = (\text{Pingtung}, (\text{Communication}))$	$t_{45} = (\text{Pingtung}, (\text{Computer}))$
$t_{46} = (\text{Taipei}, (\text{Appliance}))$	$t_{47} = (\text{Taipei}, (\text{Communication}))$	$t_{48} = (\text{Taipei}, (\text{Computer}))$
$t_{49} = (\text{Taoyun}, (\text{Appliance}))$	$t_{50} = (\text{Taoyun}, (\text{Communication}))$	$t_{51} = (\text{Taoyun}, (\text{Computer}))$
$t_{52} = (\text{Hsinchu}, (\text{Appliance}))$	$t_{53} = (\text{Hsinchu}, (\text{Communication}))$	$t_{54} = (\text{Hsinchu}, (\text{Computer}))$
$t_{55} = (\text{Tainan}, (\text{TV}))$	$t_{56} = (\text{Tainan}, (\text{Refrigerator}))$	$t_{57} = (\text{Tainan}, (\text{Cellular Phone}))$
$t_{58} = (\text{Tainan}, (\text{Radio}))$	$t_{59} = (\text{Tainan}, (\text{Monitor}))$	$t_{60} = (\text{Tainan}, (\text{Printer}))$
$t_{61} = (\text{Kaohsiung}, (\text{TV}))$	$t_{62} = (\text{Kaohsiung}, (\text{Refrigerator}))$	$t_{63} = (\text{Kaohsiung}, (\text{Cellular Phone}))$
$t_{64} = (\text{Kaohsiung}, (\text{Radio}))$	$t_{65} = (\text{Kaohsiung}, (\text{Monitor}))$	$t_{66} = (\text{Kaohsiung}, (\text{Printer}))$
$t_{67} = (\text{Pingtung}, (\text{TV}))$	$t_{68} = (\text{Pingtung}, (\text{Refrigerator}))$	$t_{69} = (\text{Pingtung}, (\text{Cellular Phone}))$
$t_{70} = (\text{Pingtung}, (\text{Radio}))$	$t_{71} = (\text{Pingtung}, (\text{Monitor}))$	$t_{72} = (\text{Pingtung}, (\text{Printer}))$
$t_{73} = (\text{Taipei}, (\text{TV}))$	$t_{74} = (\text{Taipei}, (\text{Refrigerator}))$	$t_{75} = (\text{Taipei}, (\text{Cellular Phone}))$
$t_{76} = (\text{Taipei}, (\text{Radio}))$	$t_{77} = (\text{Taipei}, (\text{Monitor}))$	$t_{78} = (\text{Taipei}, (\text{Printer}))$
$t_{79} = (\text{Taoyun}, (\text{TV}))$	$t_{80} = (\text{Taoyun}, (\text{Refrigerator}))$	$t_{81} = (\text{Taoyun}, (\text{Cellular Phone}))$
$t_{82} = (\text{Taoyun}, (\text{Radio}))$	$t_{83} = (\text{Taoyun}, (\text{Monitor}))$	$t_{84} = (\text{Taoyun}, (\text{Printer}))$
$t_{85} = (\text{Hsinchu}, (\text{TV}))$	$t_{86} = (\text{Hsinchu}, (\text{Refrigerator}))$	$t_{87} = (\text{Hsinchu}, (\text{Cellular Phone}))$
$t_{88} = (\text{Hsinchu}, (\text{Radio}))$	$t_{89} = (\text{Hsinchu}, (\text{Monitor}))$	$t_{90} = (\text{Hsinchu}, (\text{Printer}))$

3. $x_3 = (A0003, (\{Hsinchu, Taoyuan\}, \{Radio\}))$. This generates $(A003, Hsinchu, Radio)$ and $(A003, Taoyuan, Radio)$, and can be stored in the relation $DTree$ as $(A003, 9, 8)$ and $(A003, 8, 8)$, respectively.

That is, the initial relation $Dtree$ is as shown in Fig. 9.

Doc_id	$D1_tag$	$D2_tag$
A001	5	5
A002	4	9
A002	4	10
A003	9	8
A003	8	8

Fig. 9. Initial relation $DTree$ generated for $DC = (S, (R, P))$.

After the procedure **Build_Dtree** is applied, all effective tuples can be generated, and we can finally obtain the target relation *DTree* as shown in Fig. 10, where nodes $t_1 = (1, 1) = (\textit{All Region}, \textit{All Product})$ and $t_5 = (2, 1) = (\textit{South}, \textit{All Product})$ contain more than one tuple (the shaded areas in Fig. 10). This means that the document sets {A001, A002, A003} and {A001, A002} can be retrieved by issuing t_1 and t_5 as queries, respectively. If we adopt object-relational databases to store the relation *DTrees*, then this phenomenon can be alleviated, since the three tuples and two tuples of t_1 and t_5 , respectively, can be grouped into two single tuples and stored in an object-relational database. Also, object-relational databases can manipulate D-trees more efficiently by applying the *implicit join* mechanism to each foreign key reference.

	<i>Doc_id</i>	<i>D1_tag</i>	<i>D2_tag</i>
t_1	A001	1	1
t_1	A002	1	1
t_1	A003	1	1
t_2	A001	1	2
t_3	A003	1	3
t_4	A002	1	4
t_7	A001	1	5
t_{10}	A003	1	8
t_{11}	A002	1	9
t_{12}	A002	1	10
t_5	A002	2	1

	<i>Doc_id</i>	<i>D1_tag</i>	<i>D2_tag</i>
t_5	A001	2	1
t_{13}	A001	2	2
t_{15}	A002	2	4
t_{25}	A001	2	5
t_{29}	A002	2	9
t_{30}	A002	2	10
t_6	A003	3	1
t_{17}	A003	3	3
t_{34}	A003	3	8
t_{19}	A002	4	1
t_{39}	A002	4	4

	<i>Doc_id</i>	<i>D1_tag</i>	<i>D2_tag</i>
t_{59}	A002	4	9
t_{60}	A002	4	10
t_{20}	A001	5	1
t_{40}	A001	5	2
t_{61}	A001	5	5
t_{23}	A003	8	1
t_{50}	A003	8	3
t_{82}	A003	8	8
t_{24}	A003	9	1
t_{53}	A003	9	3
t_{88}	A003	9	8

Fig. 10. The target relation *Dtree* with effective D-tree nodes for Example 4.

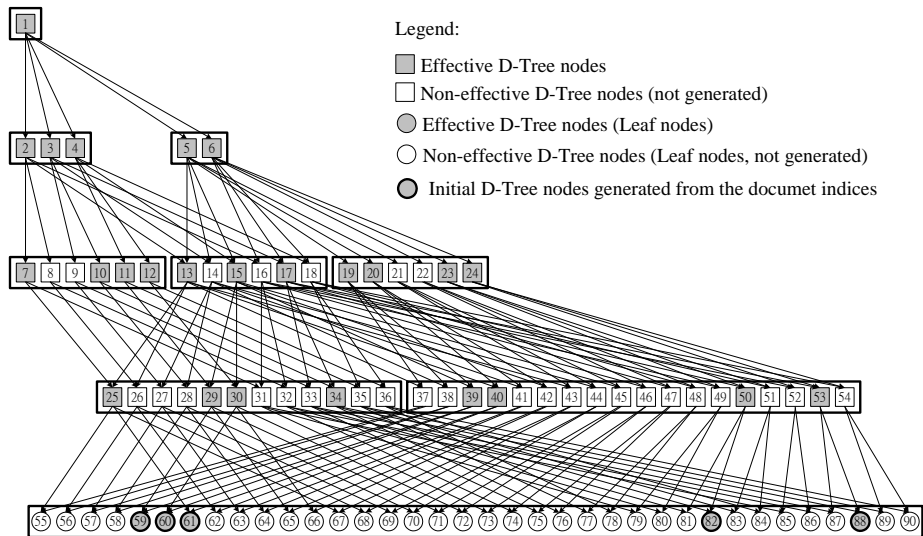


Fig. 11. The complete D-tree vs. the effective nodes.

To show the storage utilization of our approach, in Fig. 11, we depict the effective nodes (i.e., the shaded nodes) in the corresponding complete D-tree, which contains all the nodes listed in Table 2. Compared with the complete D-tree structure, only 1/3 number of the nodes are stored in our target D-tree. \square

4.4 Performance Evaluation Experiments

Based on the above storage structure, if all dimensions consist of no more than 65,536 elements, then the attribute Di_Tag in the relation $DTree$ can be encoded in only two bytes. Also, if the number of documents is no greater than 65,536, then the attribute Doc_id can be represented by a two-byte short integer. Under these circumstances, a tuple in $DTree$ only occupies $(2 + 2n)$ bytes, where n is the number of dimensions. That is, the largest $DTree$ in our experiments, which contained less than one million effective nodes, only occupied $(2 + 2 * 5) * 1,000,000 = 12,000,000$ bytes = 12MB, which is relatively small and can be easily stored in contemporary relational database management systems (RDBMSs).

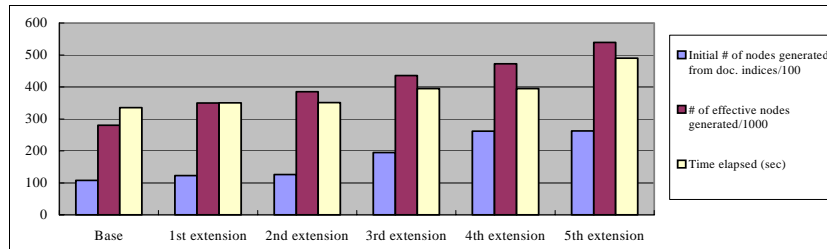
According to our dimensional modeling approach, the SQL statements discussed in sections 4.2 and 4.3 only join two relations. Therefore, the time complexity of all the statements is $O(n \log n)$, where n is the number of tuples in $DTree$. To measure the execution performance, we conducted experiments in which we generated the relation $Dtree$. The experiments were designed to illustrate the superiority of our approach and the feasibility of the D-tree. The experiments were also helpful for studying the accuracy of our method to see if the aforementioned least fixed point can be reached efficiently. In fact, the maximum number of iterations needed to reach the least fixed point is just the same as the height of the target D-tree.

In our experiments, the following data sets were used or generated based on our dimensional modeling approach:

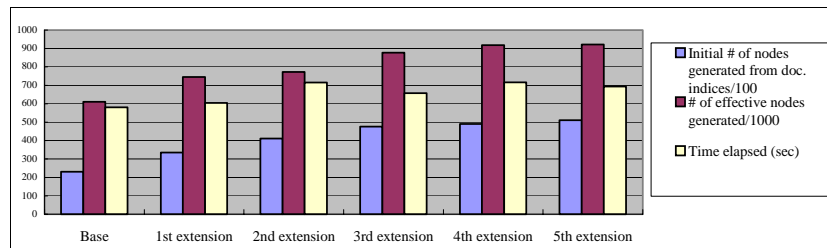
1. *Dimensions used*: As documents usually center around some events concerned with certain objects, people, places, and time periods, five dimensions were used in our experiments to represent the events, products, people, locations, and time periods involved in documents, each having 18, 10, 16, 16 and 52 nodes, respectively. The time dimension included dates from three years, with the levels *year*, *quarter*, and *month*.
2. *Document Index generation*: We generated factitious document sets consisting of the keywords in the five dimensions by means of random perturbations. There were three document sets, composed of 50, 100 and 200 documents, respectively. To test the scalability, the documents in each set were gradually extended with new keywords in each dimension to generate new document indices. In our experiment, we extended each document with new keywords in each dimension for five times. Then, based on the obtained document indices, the initial D-tree nodes could be quickly derived and stored in the relation $DTree$ as Fig. 9 depicts (in our experiments, such actions could be performed in 1 or 2 seconds).
3. *Effective D-tree node generation*: Based on the previously obtained relation $DTree$, all of the effective D-tree nodes were iteratively generated and stored in $DTree$ until the least fixed point was reached.

We implemented our method by means of Transact-SQL in Microsoft SQL Server 2000, and ran the experiments on a dedicated PC with a single Intel Pentium 4 (2.8GHz) CPU, 512M of RAM, and 120GB of hard disk space. In Fig. 12, we list the experimental results, where the elapsed time was computed from scratch for each entry, not incrementally.

# of Doc	Keyword extension	Initial # of nodes generated from doc. indices	# of effective nodes generated	Elapsed Time (sec)
50	Base	1,168	93,600	112
	1 st extension	2,224	146,600	129
	2 nd extension	3,792	166,920	148
	3 rd extension	4,352	173,280	146
	4 th extension	4,664	191,440	162
	5 th extension	6,400	210,280	189
100	Base	10,760	280,240	335
	1 st extension	12,240	349,320	350
	2 nd extension	12,568	384,880	351
	3 rd extension	19,448	435,120	395
	4 th extension	26,192	472,080	395
	5 th extension	26,240	538,880	490
200	Base	23,056	610,360	580
	1 st extension	33,576	745,560	604
	2 nd extension	41,080	772,520	715
	3 rd extension	47,584	877,760	658
	4 th extension	49,080	918,000	716
	5 th extension	51,072	922,320	693



(a) Number of documents = 100.



(b) Number of documents = 200.

Fig. 12. The experimental results.

Note that, we only show bar charts corresponding to the last two sets of documents (i.e., the sets containing 100 and 200 documents). From this illustration, one can see that the time needed to generate the effective nodes of a D-tree grew almost linearly. Therefore, for larger data sets, the experimental results scaled accordingly.

Although this process seems a bit time-consuming, it is necessary for constructing document warehouses. This is the reason why vendors usually support the flexible storage design methods of MOLAP (Multi-dimensional OLAP), ROLAP (Relational OLAP) and HOLAP (Hybrid OLAP) to achieve a time/space trade-off. Fortunately, this time-consuming process is only performed once during warehouse construction. After construction is completed, queries can be answered very quickly.

5. CONCLUSION AND FUTURE DIRECTIONS

In this paper, we have stressed the importance of constructing document warehouses to provide text-centric business intelligence, and proposed an indexing structure, the D-tree, which can be used as metadata for document warehousing. The concept behind the D-tree is simple, and its structure is easy to implement. As documents are warehoused, users can perform on-line analytical processing (OLAP) over text in a document warehouse.

To prove the effectiveness of the D-tree, we have shown that it meets the requirements [10, 25, 30] listed in section 2 below.

1. *Dynamics*. As relational tables are employed in the storage structure of the D-tree, effective nodes generated from document indices can be inserted and deleted in any given order.
2. *Secondary/tertiary storage management*. The size of a D-tree mainly depends on the number of effective nodes. Since the contents of all document instances are kept in secondary storage, only their unique identifiers (or links) are maintained in D-trees, which enable the document links to integrate secondary and tertiary storage seamlessly.
3. *Broad range of supported operations*. We have devised algorithms for multi-dimensional queries, enabling rolling up and drilling down along dimensions.
4. *Independence of input data*. Since only effective nodes are stored in a D-tree, the efficiency of a D-tree is retained, even when the input documents are highly skewed.
5. *Simplicity*. The concept behind the D-tree is simple, and its structure is easy to implement in large-scale applications.
6. *Scalability*. The D-tree adapts well to the growth in the underlying document set.
7. *Time efficiency*. Multi-dimensional searches in D-trees are fast. This guarantees logarithmic worst-case search performance for drilling down and rolling up under all possible input document distributions, regardless of the insertion sequence.
8. *Space efficiency*. A D-tree is small in size compared with the document sets to be addressed, therefore guaranteeing good storage utilization.
9. *Concurrency and recovery*. When multiple users concurrently update, retrieve and insert documents, mature relational database technology provides robust concur-

rency and recovery mechanisms for use in D-trees for document management without a significant performance penalty.

10. *Minimum impact.* Since a D-tree is kept in a separate place and is not mixed with the document set, the integration of a D-tree into a document management system has no impact on the existing parts of the system.

In our future work, we will devise an architecture for document warehousing. The preliminary components may include the following modules.

1. An XML Schema [18] used to define document metadata.
2. Automatic text summarization [11, 14, 22], key feature extraction [9], or even document classification and categorization [3] techniques for document warehousing.
3. Automatic document metadata decomposition and mechanisms for storing obtained metadata in native XML or XML-enabled databases [5-7]. This will help users manage document warehouses more efficiently.

Finally, since the construction of a document warehouse involves scanning number of documents, which is a time-consuming task, a parallel or grid architecture for this process will be investigated in the future.

ACKNOWLEDGMENTS

The authors wish to thank the journal editor and four anonymous referees, whose valuable comments and suggestions helped to improve the quality of this paper substantially.

REFERENCES

1. S. Anahory and D. Murray, *Data Warehousing in the Real World: A Practical Guide for Building Decision Support Systems*, Addison-Wesley Longman, 1997.
2. G. E. Andrews, *The Theory of Partitions*, Cambridge, England: Cambridge University Press, 1998.
3. A. Appiani, F. Cesarini, A. Colla, M. Diligenti, M. Gori, S. Marinai, and G. Soda, "Automatic document classification and indexing in high-volume applications," *International Journal on Document Analysis and Recognition*, Vol. 4, 2002, pp. 69-83.
4. M. J. A. Berry and G. Linoff, *Data Mining Techniques: For Marketing, Sales, and Customer Support*, John Wiley & Sons, New York, 1997.
5. E. Bertino and B. Catania, "Integrating XML and databases," *IEEE Internet Computing*, Vol. 5, 2001, pp. 84-88.
6. E. Bertino and E. Ferrari, "XML and database integration," *IEEE Internet Computing*, Vol. 5, 2001, pp. 75-76.
7. M. Champion, "Native XML vs. XML-enabled: the difference makes a difference," http://www.softwareag.com/xml/library/champion_nativexml.htm, Software AG: The XML Company.
8. Dublin Core Metadata Initiative, <http://dublincore.org/>.

9. F. F. Feng and W. B. Croft, "Probabilistic techniques for phrase extraction," *Information Processing and Management*, Vol. 37, 2001, pp. 199-220.
10. V. Gaede and O. Günther, "Multidimensional access methods," *ACM Computing Surveys*, Vol. 2, 1998, pp. 170-231.
11. J. Goldstein, M. Kantrowitz, V. Mittal, and J. Carbonell, "Summarizing text documents: sentence selection and evaluation metrics," in *Proceedings of 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1999, pp. 121-128.
12. M. Grigsby, "The internet document warehouse: content management for the back office," Technical Report, IMERGE Consulting, Inc., 2001, <http://www.imergeportal.com/publishedarticles.asp>.
13. A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1984, pp. 47-57.
14. U. Hahn and I. Mani, "The challenges of automatic summarization," *IEEE Computer*, Vol. 33, 2000, pp. 29-36.
15. J. Han and M. Kamber, *Data Mining: Concepts and Techniques*, Morgan Kaufmann Publishers, 2001.
16. A. Henrich, H. W. Six, and P. Widmayer, "The LSD tree: spatial access to multidimensional point and non-point objects," in *Proceedings of 15th International Conference on Very Large Data Bases*, 1989, pp. 45-53.
17. <http://www.survey.com>, "Development snapshot: warehouse data of the future," *Application Development Trends*, 2000.
18. <http://www.w3.org/XML/schema>.
19. IBM Corporation, *Intelligent Miner for Text: Text Analysis Tools version 2.10.0*, <http://www-3.ibm.com/software/data/iminer/fortext/>.
20. W. H. Inmon, *Building the Data Warehouse*, John Wiley and Sons, New York, 1993.
21. R. Kimball, *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*, John Wiley & Sons, Inc., New York, 1996.
22. K. Knight, "Mining online text," *Communications of the ACM*, Vol. 42, 1999, pp. 58-61.
23. S. H. Lin, C. S. Shih, M. C. Chen, J. M. Ho, M. T. Ko, and Y. M. Huang, "Extracting classification knowledge of internet documents with mining term associations: a semantic approach," in *Proceedings of 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1998, pp. 241-249.
24. S. Loh, L. K. Wives, and J. P. de Oliverira, "Concept-based knowledge discovery in texts extracted from the web," *ACM SIGKDD Explorations*, Vol. 2, 2000, pp. 29-40.
25. D. Lomet and B. Salzberg, "The hB-tree: a multiattribute indexing method with good guaranteed performance," *ACM Transactions on Database Systems*, Vol. 15, 1990, pp. 625-658.
26. M. C. McCabe, J. Lee, A. Chowdhury, D. Grossman, and O. Frieder, "On the design and evaluation of a multi-dimensional approach to information retrieval," in *Proceedings of 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2000, pp. 363-365.
27. B. C. Ooi, R. Sacks-Davis, and K. J. McDonnell, "Spatial k-d-tree: an indexing mechanism for spatial databases," in *Proceedings of the IEEE 11th Annual Interna-*

- tional Computer Software and Applications Conference (COMPSAC '87)*, 1987, pp. 433-438.
28. Oracle Corporation, InterMedia Text 8.1.6., http://otn.oracle.com/products/text/x/Tech_Overviews/imt_817.html.
 29. R. Hackathorn, "Data warehousing energizes your enterprise," *Datamation*, 1995, Vol. 1, pp. 38-42.
 30. J. T. Robinson, "The K-D-B-tree: a search structure for large multidimensional dynamic indexes," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1981, pp. 10-18.
 31. T. K. Sellis, N. Roussopoulos, and C. Faloutsos, "The R⁺-tree: a dynamic index for multi-dimensional objects," in *Proceedings of 13th International Conference on Very Large Data Bases (VLDB)*, 1987, pp. 507-518.
 32. G. Spofford, *MDX Solutions – with Microsoft SQL Server Analysis Services*, John Wiley & Sons, Inc., New York, 2001.
 33. D. Sullivan, *Document Warehousing and Text Mining: Techniques for Improving Business Operations, Marketing and Sales*, John Wiley & Son, Inc., New York, 2001.
 34. A. H. Tan, "Text mining: the state of the art and the challenges," in *Proceeding of Workshop on Knowledge Discovery from Advanced Databases (PAKDD)*, 1999, pp. 50-70.
 35. F. S. C. Tseng and W. P. Lin, "A study on indexing structure and its properties for constructing document warehouses," in *Proceedings of 20th Workshop on Combinatorial Mathematics and Computation Theory*, 2003, pp. 18-27.
 36. F. S. C. Tseng, "Design of a multi-dimensional query expression for document warehouses," *Information Sciences: An International Journal*, Vol. 174, 2005, pp. 55-79.
 37. J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, Computer Science Press, Vol. 2, Rockville, MD, 1988.



Frank S. C. Tseng (曾守正) received the B.S., M.S. and Ph.D. degrees, all in Computer Science and Information Engineering, from National Chiao Tung University, Taiwan, R.O.C., in 1986, 1988, and 1992, respectively. Dr. Tseng was one of the winners of the *Acer* Long Term Ph.D. dissertation prize in 1992. He joined the faculty of the Department of Information Management, Yuan Ze University, Taiwan, R.O.C., in August, 1995. From 1996 to 1997, he was the Chairman of the Department. He is currently a professor in the Department of Information Management, National Kaohsiung First University of Science and Technology. His research interests include heterogeneous database systems, XML technologies for Internet computing, data warehousing, data mining, and document warehousing. Dr. Tseng is a member of the IEEE Computer Society and the Association for Computing Machinery. He was listed in *Marquis Who's Who in Medicine and Healthcare* in 2004 and *Marquis Who's Who in Asia* in 2007.



Wen-Ping Lin (林文平) received the MBA degree in Information Management from National Kaohsiung First University of Science and Technology, Taiwan, R.O.C., in 2003. He is now completing his military obligation in the R.O.C. Army. His research interests include database systems, data warehousing, and electronic document management.