January 18, 2007

RT0703
Computer Science pages

Research Report

Online Analytical Processing of Text Data

Akihiro Inokuchi, Koichi Takeda

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

1 Introduction

Since many of business intelligence applications started to incorporate unstructured (primarily textual) information for more context-oriented analysis and decision-making [4], database technology has been seriously challenged to ingest, map, store, and access such text-originated information along with the structured information in a way that two types of information can mutually enhance information discovery and analysis capability. The most critical problems is that most of semantics underlying the unstructured information (such as ontological hierarchy, synonymous and antonymous relationship) cannot be effectively managed by conventional database systems. Another significant problem is that a rigid schematic representation (and associated queries and analytic processing) of unstructured information often suffer frequent modifications due to updates to dictionary and ontology for adequately categorize words, phrases, and entities included and described in the unstructured information. Therefore, it is very important to propose a more flexible representation, which reduces the cost and workload of frequent rebuild and re-population of the database schema.

The multidimensional database technology has been considered for the interactive analysis of large amounts of data for decision making purposes [20, 11, 1, 2, 8, 10]. Multidimensional data models categorize data either as facts with associated numerical measures or as dimensions that characterize the facts. In a retail business, for example, a purchase transaction would be a fact and the purchase amount and price would be measures, and the type of purchased product, the purchase time and location would be dimensions. Queries for Online Analytical Processing (OLAP) aggregate measures over a range of dimensional values to provide results such as the total sales per month of a given product, leading to overall trends. An important feature of the multidimensional data model is to use hierarchical dimensions to provide as much context as possible for the facts. Dimensions are used for selecting and aggregating data at the desired level of detail. Most of traditional multidimensional databases assume that the dimensional hierarchies are balanced and non-ragged trees.

The star and snowflake schemas which are representative schemas for the multidimensional data model store data in fact tables and dimension tables. A fact table holds one row for each fact in the database and it has a column for each measure, containing the measured value for the particular fact, as well as a column for each dimension that contains a foreign key referring to a dimension table for the particular dimension.

When analyzing unstructured information in a multidimensional data model, a document would be typically represented as a fact, and categories of keywords, such as protein, gene, or disease in the lifescience domain, would be selected as axis for the interactive analysis as shown in Figure 1. Each cell of the cube in the figure stores the number of the corresponding documents. Operations, such as drill down, roll up, slice, dice, pivoting or drill through, are available for analyzing/aggregating large amounts of documents and their contextual information to obtain insights. It is often very difficult, however, to define a set of dimensions and their hierarchies for a huge set of keywords such as protein name, gene names[1] . To design a hierarchy used in the online analytical processing, we use ontologies such as Unified Medical Language System (UMLS)[2] and the Gene Ontology (GO)[3] , which is a kind of a directed acyclic graph, rather than a set of balanced and non-ragged trees. When we assume that each node of the hierarchy corresponds to a dimension, many missing values and a set of multiple values for the node could possibly be introduced. In addition, because the number of nodes in the hierarchy becomes very large and a complex relationship among the nodes exists, we cannot store the data in the star schema and efficiently aggregate the data within the hierarchy under a straightforward implementation.

In this paper, we propose a data representation, and algebraic operations to integrate a multidimensional model with ontologies to analyze a huge set of textual documents. This paper describes

- how we design the data representation and its algebraic operations to realize multidimensional model and to integrate with ontologies,

[1] The number of distinct keywords is 13,640,593.
[2] UMLS: http://umlsinfo.nlm.nih.gov
[3] GO: http://www.geneontology.org

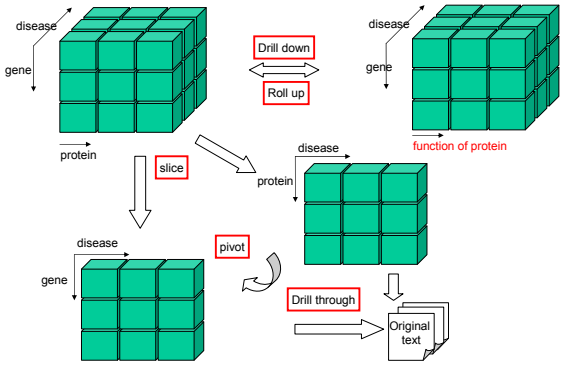


Figure 1: Operations for a Multidimensional Database

- how we store very high dimensional data from text documents in a relational database,
- how we efficiently aggregate the distributions of documents for each cell in the cube view, and
- how we can get sufficient performance to provide user interactivity.

The rest of this paper is organized as follows. Section 2 addresses a hierarchy and an ontology, and Section 3 defines our data representation and its algebraic operations. In Section 4, we introduce our schemas to efficiently compute the distributions and their implementations. Section 5 presents experimental results using about 500,000 journal abstracts, and Section 6 discusses related works. Finally, Section 7 concludes this paper.

2 Hierarchy and Ontology

In this section, we give formal definitions of a hierarchy and an ontology according to [3]. If S is a nonempty set, and $\leq \subseteq S \times S$, then $(S, \leq)$ is an ordering[4] . If $x \leq x$ for $x \in S$, then S is reflexive. If $x \leq y$ and $y \leq z \rightarrow x \leq z$ for $x,\ y,\ z \in S$, then S is transitive. If $x \leq y$ and $y \leq x \rightarrow x = y$ for $x,\ y \in S$, then S is anti-symmetric. $(S, \leq)$ is a partial ordering if S is a reflexive, transitive, and anti-symmetric binary relation on S.

Definition 1 (better): Let $(S, \leq_1)$ and $(S, \leq_2)$ be two orderings. We say $(S, \leq_1)$ is better than $(S, \leq_2)$ iff $\forall x, y \in S\ (x \leq_1 y \rightarrow x \leq_2 y)$. In addition, we say that $(S, \leq_1)$ is strictly better than $(S, \leq_2)$ iff $(S, \leq_1)$ is better than $(S, \leq_2)$ and $(S, \leq_2)$ is not better than $(S, \leq_1)$.

Definition 2 (hierarchy): Let $(S, \leq)$ be a partial ordering. A hierarchy of S is an ordering $(S, \preceq)$ such that

1. $(S, \preceq)$ is better than $(S, \leq)$,
2. $(S, \leq)$ is the reflexive, transitive closure of $(S, \preceq)$, and
3. there is no other ordering $(S, \sqsubseteq)$ satisfying the preceding two conditions such that $(S, \sqsubseteq)$ is strictly better than $(S, \leq)$.

Definition 3 (ontology): Suppose Σ is some finite set of strings and S is some set. An ontology w.r.t. Σ is a partial mapping Θ from Σ to hierarchies for S.

For example, when S is given as $\{tire, car, hubcap\}$, where tire is a part of car, hubcap is a part of car, and hubcap is a part of a tire. In addition, everything is a part of itself. For the set S, a partial order is defined as $\{(tire,\ tire),\ (car,\ car),\ (hubcap,\ hubcap),\ (tire,\ car),\ (hubcap,\ car),\ (hubcap,\ tire)\}$, and only one hierarchy is defined as $\{(tire, car), (hubcap, tire)\}$, as shown in Figure 2.

[4] This paper uses $\leq$ to represent a direct relation between two elements in the set S, $<^*$ to represent its transitive closure, and $\leq^*$ to represent its transitive closure or represent that the elements are equal.

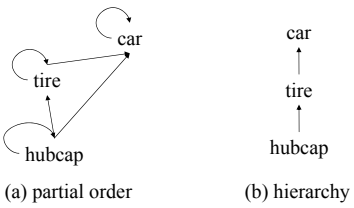


Figure 2: Examples of a Partial Order and a Hierarchy

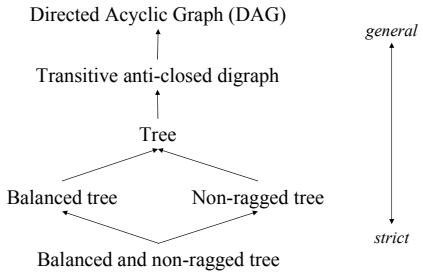


Figure 3: Taxonomy of Hierarchy

Hierarchies can be classified according to their generality as shown in Figure 3 [18].
DAG: Directed acyclic graph (DAG) which is a directed graph with no directed cycles is the general class for the taxonomy. The hierarchy introduced above to define the ontology is a proper subclass of this class.
transitive anti-closed digraph: The transitive closure can be constructed as follows. If there is a path from node A to node B of length 2 or more, then add an edge from A to B. On the other hand, the anti-closure can be produced as follows. If there are an edge of length 1 and a path from node A to node B of length 2 or more, then remove the path of length 1 from A to B. The hierarchy defined above is in this class.
tree: A tree is a DAG where each node can only have one parent, except for one node which has no parents and which is called the root.
balanced tree: An unbalanced hierarchy with levels that have a consistent parent-child relationship but have a logically inconsistent levels. The hierarchy branches also can have inconsistent depths. For example, an unbalanced hierarchy can represent an organization chart. Figure 4 (a) shows a chief executive officer (CEO) on the top level of the hierarchy and at least two of the people that might branch off below including the chief operating officer and the executive secretary. The chief operating officer has more people branching off also, but the executive secretary does not. The parent-child relationships on both branches of the hierarchy are consistent. However, the levels of both branches are not logical equivalents. An executive secretary is not the logical equivalent of a chief operating officer[5] .
non-ragged tree: A ragged hierarchy in which each level has a consistent meaning, but the branches have inconsistent depths because at least one member attribute in a branch level is unpopulated. A ragged hierarchy can represent a geographic hierarchy in which the meaning of each level such as city or country is used consistently, but the depth of the hierarchy varies. Figure 4 (b) shows a geographic hierarchy that has Continent, Country, State, and City levels defined. One branch has North America as the Continent, United States as the Country, California as the State, and San Francisco as the City. However, the hierarchy becomes ragged when some member does not have an entry at all of the levels. For example, another branch has Europe as the Continent, Greece as the Country, and Athens as the City, but has no entry for the State level because this level is not applicable to Greece for the business model in this

[5] http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=
/com.ibm.db2.udb.db2_olap.doc/cmdhierarchy.htm

example. In this example, the Greece and United States branches descend to different depths, creating a ragged hierarchy.

balanced and non-ragged tree: Most of traditional multidimensional databases use hierarchies of this class.

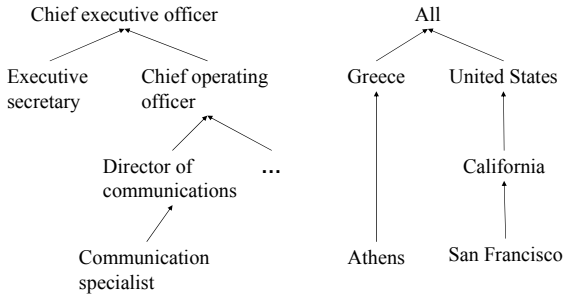


Figure 4: Examples of an Unbalanced Tree and a Ragged Tree

3 Data Object and Operations

In this section, we give formal definitions of our data representation and operations according to [19].

3.1 Data Object

Given a hierarchy (or an ontology) $(S, \leq)$, a fact schema is defined as $\mathcal{S} = (\mathcal{F}, \mathcal{T})$, where $\mathcal{F}$ is a fact type and $\mathcal{T}$ is a hierarchy type $\mathcal{T} = (\mathcal{C}, \leq_{\mathcal{T}}, \top_{\mathcal{T}})$ which is strictly better than $(S, \leq)$ and the relations in $(S, \leq)$ required for analyzing the documents are remaining in $\mathcal{T}$. The hierarchy type is a three-tuple $(\mathcal{C}, \leq_{\mathcal{T}}, \top_{\mathcal{T}})$, where $\mathcal{C} = \{\mathcal{C}_{|}, | = \infty, \cdots, \backslash\}$ is a set of category types of $\mathcal{T}$, and $\leq_{\mathcal{T}}$ is a partial order on the $\mathcal{C}$'s, with $\top_{\mathcal{T}} \in \mathcal{C}$ being the top element of the ordering. The intuition is that the top element of the ordering logically contains all other elements, that is $\forall \mathcal{C}_{|} \in \mathcal{C},\ \mathcal{C}_{|} \leq^{*} \top_{\mathcal{T}}$.

A hierarchy instance T of type $\mathcal{T}$ is a two-tuple $T = (C, \leq)$, where C is a set of categories c_j such that $Type(c_j) = \mathcal{C}_{|}$, and $\leq$ is a partial order on C. Functions to give the set of immediate predecessors and successors of a category c_j are defined as $pred : C \rightarrow 2^C$ and $succ : C \rightarrow 2^C$. That is, $pred(c_j) = \{c' \mid c' > c_j\}$ and $succ(c_j) = \{c' \mid c' < c_j\}$, respectively. Each category $c \in C$ has an associated set $dom(c)$ called its domain. The members of $dom(c)$ are called values of the category c. An element in $dom(c)$ is represented as $c : v$. In addition, a function *below* to give a set of values is also defined as $below(c) = \{dom(c') \mid c' \leq^{*} c\}$.

For example, we have a hierarchy instance T, a part of which is depicted in Figure 5 (a). Categories such as "Software", "OS", "Middleware", "Application", "Windows", "Linux", "AIX" are contained in C. $pred(OS)$ has only one element "Software", and $succ(OS)$ contains {"Windows", "Linux", "AIX"}. $dom(Windows)$ contains "Windows XP", "Windows Me", "Windows 2000", and so on. $below(All)$ contains all values of all categories.

Let $F = \{f_i, i = 1, \cdots, m\}$ be a set of facts. A fact-hierarchy relationship between F and T is a set $R = \{(f, c : v)\}$, where $f \in F$, $c \in C$, and $v \in dom(c)$. Thus, R links facts to hierarchical values. We say that fact f is characterized by a hierarchical value $c : v$, written by $f \rightsquigarrow c : v$, if $\exists c' \in C((f, c' : v') \in R \wedge c' \leq^{*} c \wedge v = v')$.

Our data object is a four tuple $D = \{\mathcal{S}, \mathcal{F}, \mathcal{T}, \mathcal{R}\}$, where $\mathcal{S} = (\mathcal{F}, \mathcal{T})$ is the fact schema, F is a set of facts where $Type(f) = \mathcal{F}$, $T = (C, \leq)$ is a hierarchy instance where $Type(c_j) = \mathcal{C}_{|}$ for $c_j \in C$ and $\mathcal{C}_{|} \in \mathcal{C}$, and R is a set of fact-hierarchy relations such that $(f, c : v) \in R \Rightarrow f \in F \wedge \exists c \in C\ (v \in dom(c))$.

For example, we have the hierarchy instance T and an analyzed document which is depicted in Figure 5 (b). F contains a set of document identifiers. Terms in the document whose

document id is 1 in Figure 5 are annotated in preprocessing, e.g., a category "windows" and "workstation" are assigned to a term "windows 2000" and "IntelliStation 6217", respectively, and $(1, windows : windows\_2000)$ and $(1, workstation : IntelliStation\_6217)$ are stored in R.

Conceptually, R corresponds to a relation $R' \subseteq 2^{dom(c_1)} \times \cdots \times 2^{dom(c_n)}$ which is not a normalized relation. R' corresponds to a fact table for a star schema, and each row and column in R' correspond to a document (fact) and a category (dimension value in the star schema), respectively. Because the relation has many missing values and a set of values for each attribute c_j, the number of attributes in the relation becomes very large and a complex relationship among the attributes (columns) exists, a naive method cannot store the data in a relational database and efficiently aggregate the data along the hierarchy.

For example, a hierarchy instance T used in Section 5 is a transitive anti-closed digraph which has more than 240,000 nodes (categories) and more than 340,000 edges and whose depth is 24. If the hierarchy instance is a tree, $V = E + 1$, where V and E are the numbers of categories and edges of the hierarchy, respectively. However, because the difference between the numbers of the categories and edges in T is so large, the hierarchy instance T used in Section 5 have a very complex relationship. In addition, about 36,400,000 elements in a conceptual relation R' have values. Since the number of attributes, n, for R' is greater than 240,000 and R' has more tuples than 500,000 in Section 5, most of elements in R' are missing values. Furthermore, more than 7,600,000 elements in R' has a set of values, and $dom(\top)$ has about 193,000,000 distinct values.

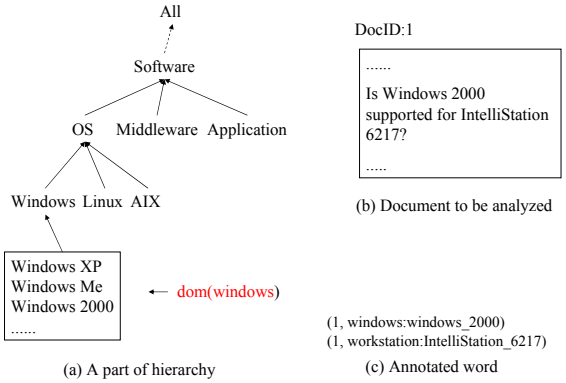


Figure 5: Examples of a Hierarchy and Fact-Hierarchy Relations

The function $g(c)$ is defined as a user-defined function to return a set of fact-hierarchy relations, and another function $G(g(c))$ is defined as $G(g(c)) = \{(c : v) \mid (f, c : v) \in g(c)\}$. For $k = 1, \cdots, q$ and $(c_k : v_k) \in G(g_k(c_k))$, we define a function $Group$ as $Group(c_1 : v_1, \cdots, c_q : v_q) = \{f \mid f \in F \wedge (f, c_1 : v_1) \in g_1(c_1) \wedge \cdots \wedge (f, c_q : v_q) \in g_q(c_q)\}$. These functions are used to aggregate the distributions of documents for each keyword and category. For example, $g(c)$ is provided as the following functions $g^{(1)}, g^{(2)}, g^{(3)}$ $(g \in \{g^{(1)}, g^{(2)}, g^{(3)}, \cdots\})$.

First, $g^{(1)}(c)$ is defined as $g^{(1)}(c) = \{(f, c' : v') \mid (f, c' : v') \in R \wedge c = c' \wedge v' = dom(c')\}$. This function is used to aggregate the distributions for keywords belonging to the specified category c. The second function $g^{(2)}(c)$ is defined as $g^{(2)}(c) = \{(f, c : c'') \mid (f, c' : v') \in R \wedge c'' \in succ(c) \wedge c' \leq^* c'' \wedge v' \in dom(c')\}$. This function is used to aggregate the distributions for the immediate successors (subcategories) of the specified category c. The third one is defined as $g^{(3)}(c) = \{(f, c' : v') \mid (f, c' : v') \in R \wedge c' \leq^* c \wedge v' \in dom(c')\}$. This function is used to aggregate the distributions for keywords belonging to $below(c)$. Users can define any additional functions as required for the intended analysis of a set of documents.

3.2 Operations

This subsection defines operations on our data object.

selection σ': Given a compound predicate $P = p_1$ *or* $\cdots$ *or* p_l where each atomic predicate p_i is represented in the form of $c : v$ or $c : *$. The selection σ' is defined as $\sigma'_P(D) = (\mathcal{S}, \mathcal{F}', \mathcal{T}, \mathcal{R}')$, where $F' = \{f \mid f \in F \wedge (f \rightsquigarrow p_1 \vee \cdots \vee f \rightsquigarrow p_l)\}$, and $R' = \{(f', c : v) \in R \mid f' \in F'\}$. For

example, a set of documents having any keywords belonging to a category 'software' is given by $\sigma'_{'software':*}(D)$. Other examples $\sigma'_{'gene\_name':'BIKE'}(D)$ and $\sigma'_{c:v_1}(\sigma'_{c:v_2}(D))$ represent a set of documents having a term 'BIKE' belonging to a category 'gene_name' and a set of documents having a term v_1 and v_2 belonging to a category c, respectively.

difference $-$: Given two data objects $D_1 = (\mathcal{S}_\infty, \mathcal{F}_\infty, \mathcal{T}_\infty, \mathcal{R}_\infty)$ and $D_2 = (\mathcal{S}_\in, \mathcal{F}_\in, \mathcal{T}_\in, \mathcal{R}_\in)$ such that $\mathcal{S}_\infty = \mathcal{S}_\in = \mathcal{S}$, the difference is defined as $(\mathcal{S}, \mathcal{F}_\infty, \mathcal{T}_\infty, \mathcal{R}_\infty) - (\mathcal{S}, \mathcal{F}_\in, \mathcal{T}_\in, \mathcal{R}_\in) = (\mathcal{S}, \mathcal{F}', \mathcal{T}_\infty, \mathcal{R}')$, where $F' = F_1 - F_2$ and $R' = \{(f, c:v) \mid f' \in F',\ (f', c:v) \in R\}$. For example, a set of documents which have a term v_1 and does not have the term v_2 is $\sigma'_{\top:v_1}(D) - \sigma'_{\top:v_2}(D)$.

projection π': The projection is defined as $\pi'_{c_1 \vee \cdots \vee c_l}(D) = (\mathcal{S}, \mathcal{F}, \mathcal{T}, \mathcal{R}')$, where $R' = \{(f, c:v) \in R \mid f \in F \wedge (f \rightsquigarrow c_1 : * \vee \cdots \vee f \rightsquigarrow c_l : *)\}$.

aggregation α: Given a set of categories and functions $T = (c_1, \cdots, c_q, g_1, \cdots, g_q)$, the aggregation α is defined as $\alpha[T, 'count'](D) = (\mathcal{S}', \mathcal{F}', \mathcal{T}', \mathcal{R}')$, where $\mathcal{S}' = (\mathcal{F}', \mathcal{T}')$, $\mathcal{F}' = \in^{\mathcal{F}}$, $\mathcal{T}' = (\mathcal{C}', <'_{\mathcal{T}'}, \top'_{\mathcal{T}'})$, $\mathcal{C}' = \mathcal{C} \cup \{'\rfloor\downarrow\sqcap\backslash\sqcup'\}$, $<'_{\mathcal{T}'} = <_{\mathcal{T}} \cup \{('count', \top)\}$, $\top'_{\mathcal{T}'} = \top_{\mathcal{T}}$, $F' = \{Group(c_1 : v_1, \cdots, c_q : v_q) \mid (c_1 : v_1, \cdots, c_q : v_q) \in G(g_1(c_1)) \times \cdots \times G(g_q(c_q)) \wedge Group(c_1 : v_1, \cdots, c_q : v_q) \neq \emptyset\}$, $T' = (C', <')$, $C' = C' \cup \{'count'\}$, $<' = <_T \cup \{('count', \top)\}$, $R' = R'_1 \cup R'_2$, $R'_1 = \{(f', c' : v') \mid \exists (c_1 : v_1, \cdots, c_q : v_q) \in G(g_1(c_1)) \times \cdots \times G(g_q(c_q))\ (f' = Group(c_1 : v_1, \cdots, c_q : v_q) \wedge f' \in F' \wedge \exists k\ (c_k : v_k = c' : v'))\}$, and $R'_2 = \bigcup_{(c_1:v_1, \cdots, c_q:v_q) \in G(g_1(c_1)) \times \cdots \times G(g_q(c_q))} \{(s, 'count' : |s|) \mid s = Group(c_1 : v_1, \cdots, c_q : v_q) \wedge s \neq \emptyset\}$. $('count', \top)$ represents a partial relation $'count' < \top$.

For example, Table 1 shows the results for a query $\alpha['protein', g, 'count1'](\sigma'_{'disease':'diabetes'}(D))$ for a set of biomedical documents. The results correspond to a top N ranking for the traditional multidimensional analysis. The result lists the names of proteins relevant to diabetes. Table 2 shows the results for a query $\alpha['company', 'GeneSymbol', g_1, g_2, 'count2'](D)$. If the result is analyzed for a set of patent documents, a strategist for a pharmaceutical company might be able to find associations between companies and genes. A query $\alpha['protein', 'protein', g_1, g_2, 'count3'](D)$ may be useful to find interactions between proteins. It represents that we can select the same categories as cube's axes unlike the traditional multidimensional database.

Table 1: Top N ranking

protein	# of documents
Flavohemoprotein	347
Lamin L	240
Insulin	151
nterferon gamma precursor	97
...	...

Table 2: 2 Dimensional Map

	LEPR	TM4SF2	INS	ADAM2	...
company1	x_{11}	x_{12}	x_{13}	x_{14}	...
company2	x_{21}	x_{22}	x_{23}	x_{24}	...
company3	x_{31}	x_{32}	x_{33}	x_{34}	...
...	...	...	...	...	...

By using above operators, we will show how common OLAP operators can be defined.

roll-up & drill-down: In the traditional multidimensional database, there are two types of rolling up operation, one is dimensional rolling up and the other is hierarchical rolling up. For example, let S be the fact table $S(product, city, time, sale)$, where sale is a measure, and $L(city, state, country)$ be one of the dimension tables. The dimensional rolling up is represented as ${}_{product,city}\chi_{product,city,SUM(sale)}(S)$ in the case of one dimension being

dropped, and ${}_{city}\chi_{city,SUM(sale)}(S)$ in the case of two dimensions being dropped[6] . The hierarchical rolling up is represented as ${}_{product,state,time}\chi_{product,state,time,SUM(sale)}(S \bowtie_{city} L)$. It is possible to define more than 2^k roll-up queries for the k dimensions of the traditional multidimensional database. In our case, the dimensional rolling up corresponds to moving from $\alpha[c_1, \cdots, c_q, g_1, \cdots, g_q, 'count'](D)$ into $\alpha[c_1, \cdots, c_h, c_{h+2}, \cdots, c_q, g_1, \cdots, g_h, g_{h+2}, \cdots, g_q, 'count'](D)$ for the case of one dimension being dropped, and the hierarchical rolling up corresponds to moving from $\alpha[c_1, \cdots, c_q, g_1^{(1)}, \cdots, g_q^{(1)}, 'count'](D)$ to $\alpha[c_1, \cdots, c_h, c'_{h+1}, c_{h+2}, \cdots, c_q, g_1^{(1)}, \cdots, g_h^{(1)}, g_{h+1}^{(2)}, g_{h+2}^{(1)}, \cdots, g_q^{(1)}, 'count'](D)$, where $c'_{h+1} \in pred(c_{h+1})$.

slice & dice: The slice operation performs a selection on one dimension of the given cube, resulting in a subcube, and the dice operation defines a subcube by performing a selection on two or more dimension. For example, in the traditional multidimensional database, slice and dice operations are represented as ${}_{city,time}\chi_{city,time,SUM(sale)}(\sigma_{product=p_1}(S))$ and ${}_{product,city,time}\chi_{product,city,time,SUM(sale)}(\sigma_P(S))$, where $P = (product \in \{p_1, p_2\}\ and\ city \in \{c_3, c_4\})$. In our case, the slice is represented as $\alpha[T, 'count'](\sigma'_p(D))$, and the dice is represented as $\alpha[T, 'count'](\sigma'_{p_1\ or\ p_2}(\sigma'_{p_3\ or\ p_4}(D)))$.

pivot: The pivot operation is a visualization operation that rotates the data axes in view in order to provide an alternative presentation of the data, which corresponds to moving from $\alpha[c_1, c_2, g_1, g_2, 'count']$ into $\alpha[c_2, c_1, g_2, g_1, 'count']$.

4 Implementation

A key strategy for speeding up cube view processing for the traditional multidimensional database is to use pre-computed cube views. The pre-computation makes it possible for response times to queries potentially involving huge amounts of data to be fast enough to allow interactive data analysis in the traditional approaches. However, it is impossible to precompute or pre-aggregate in advance of receiving queries for all of the combinations of values in our situation, because the situation where each document has many values and there are a lot of categories is combinatorially explosive. For example, the average number of annotated terms which each documents have is about 380 and the number of categories is more than 240,000 for the data used in Section 5.

In this section, we design table schemas and data structures to achieve query response times that are as fast as possible. Since a hierarchy for analyzed documents constitutes a transitive anti-closed digraph rather than a set of balanced and non-ragged trees, it cannot be stored in a star schema or snowflake schema. For computation efficiency in aggregating the distributions of documents, the hierarchy is indexed as follows. A depth first search traverses the hierarchy from root category c_{root} whose type $Type(c_{root})$ is equal to $\top_{\mathcal{T}}$ assigning a preorder, postorder, and depth to each category, and it backtracks if and only if it reaches leaf nodes. This means that it does not backtrack when it reaches any internal nodes which it has already visited.

The assigned preorders and postorders make it possible to handle ancestor-descendent containment in the hierarchy [7]. In other words, it can check the containment by assigning a preorder and a postorder to each node in a hierarchy and comparing the numbers assigned to the two nodes. If a node A is an ancestor of a node B,

$$\begin{aligned} &A's\ preorder < B's\ preorder\ \&\\ &A's\ postorder > B's\ postorder. \end{aligned} \tag{1}$$

For example, the hierarchy in Figure 6 (a) is traversed to return the tree shown in Figure 6 (b) where each node has a category, an assigned preorder, postorder, and depth. In this figure, all descendents of a category c_2 have preorders which are greater than the preorder of c_2 and have postorders which are less than the postorder of c_2. We call a tree in Figure 6 (b) a traversed tree.

We define two tables, `CATEGORY` H and `KEYWORD` V, to store the traversed tree and fact-hierarchy relations as

```
CATEGORY (CATEGORYNAME  CHARCTER,
```

[6] ${}_{a}\chi_{a,sum(b)}$ represents an SQL query "SELECT a, SUM(b) FROM ... GROUP BY a".

```
          PATH          CHARCTER,
          PREORDER1     INTEGER,
          PREORDER2     INTEGER,
          PARENT        INTEGER), and
KEYWORD (ID          INTEGER,
         PREORDER    INTEGER,
         VALUE       CHARCTER),
```

respectively. Each record in the table H corresponds to a node in the traversed tree and `CATEGORYNAME`, `PATH`, `PREORDER1`, `PREORDER2`, and `PARENT` in the table H are a name of the category, a path from root node to the corresponding node, a preorder, a value for a postorder plus a depth, and a preorder of its parent of the corresponding node. The reason why we use `PREORDER2` instead of the postorder is that we can check ancestor-descendent containment in the hierarchy as

$$\begin{aligned} &A's\ preorder1\ (= A's\ preorder)\ < B's\ preorder \\ &\quad \leq A's\ preorder2 = A's\ postorder + A's\ depth \end{aligned} \tag{2}$$

instead of using condition (1), and it can reduce the space to store the tables and their indexes. Each record in the table V corresponds to $(f, c : v)$ in R, and `ID`, `PREORDER`, and `VALUE` in the table V are a document ID f, a preorder of the category c, and a value v in $dom(c)$, respectively.

By using these tables, we can implement the operations introduced in Section 3.2 as follows. In the following definitions, c is provided as input. Although there are multiple records whose values of `CATEGORYNAME` in the table H are c, a record arbitrarily chosen from the records is used in the following operations. In other words, "$\sigma_P(H)$" for $P = (categoryname = c)$ is replaced into "$\sigma_P(H)$ `FETCH FIRST 1 ROWS ONLY`", denoted as H_c. The choice has no influence on its result.

selection σ': The selection is defined as $\sigma'_{c:v}(D) = (H, V')$, where $V' = \sigma_{id\ in\ I}(V)$, $I = \pi_{id}(V \bowtie_P H_c)$ and $P = (preorder1 \leq preorder \leq preorder2\ and\ value = v)$. Although this calculation needs to join H with V, this calculation runs as fast as the selection of V, since only one record is returned from H_c.

difference $-$: The difference is defined as $(H, V_1) - (H, V_2) = (H, V')$, where $V' = \sigma_{id\ in\ (\pi_{id}(V_1) - \pi_{id}(V_2))}(V_1)$.

projection π': The projection is defined as $\pi'_c(D) = (H, V')$, where $V' = \pi_{id,preorder,value}(V \bowtie_P H_c)$ and $P = (preorder1 \leq preorder \leq preorder2)$.

aggregation α: The aggregation is defined as $\alpha[(c_1, \cdots, c_q, g_1, \cdots, g_q), '\,count']$ $(D) = {}_{value_1,\cdots,value_q}\chi_{value_1,\cdots,value_q,count(distinct\ id)}(X)$, where $X = g_1(c_1) \bowtie_{id=id} \cdots \bowtie_{id=id} g_q(c_q)$. The user-defined functions $g^{(1)}(c)$, $g^{(2)}(c)$, and $g^{(3)}(c)$ are defined as $g^{(1)}(c) = \pi_{id,preorder,value}(V \bowtie_P H_c)$, where $P = (preorder1 = preorder)$, $g^{(2)}(c) = \pi_{id,H.preorder1,H.categoryname}((H_c \bowtie_{P_1} H) \bowtie_{P_2} V)$, where $P_1 = (H.parent = H_c.preorder1)$ and $P_2 = (H.preorder1 \leq V.preorder \leq H.preorder2)$, and $g^{(3)}(c) = \pi_{id,preorder,value}(V \bowtie_P H_c)$, where $P = (preorder1 \leq preorder \leq preorder2)$.

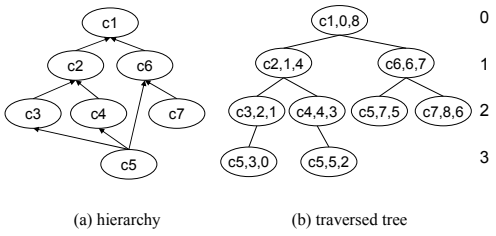


Figure 6: A hierarchy and a Traversed Tree

5 Experiments

5.1 Data and Preprocess

For testing, we used biomedical documents from MEDLINE[7] . Life science researchers typically use MEDLINE, a bibliography database that covers the biomedical area. MEDLINE is administered by the National Center for Biotechnology Information (NCBI)[8] of the United States National Library of Medicine (NLM)[9] . It contains approximately 13 million biomedical citations, dating from the mid-1960s up to the present time. Citations in MEDLINE are collected from over 4,600 biomedical journals published worldwide. Biomedical citations in MEDLINE are available to the general public at the PubMed[10] . We selected 503,989 abstracts from Medline which contain structured information such as authors and Mesh Terms and unstructured information such as titles and abstracts.

To prepare a fact-hierarchy relation from the documents, the documents written in English are parsed by CCAT [5], a shallow syntactic parser. Because this is a general-purpose parser that has not been trained for biomedical documents, it is difficult to obtain optimized results by parsing documents from various domains . We solve this problem by first annotating the text with domain dictionaries as shown in Figure 7. The annotations facilitate the parsing of the documents even when the parser has not been specifically trained for the domains.

In the first step of the preprocessing, the term annotator finds words in the input text using the term dictionary and identifies these words with their canonical forms. The term dictionary contains pairs of surface forms and canonical forms. Fox example, most of the technical terms in the medical domain are compound words. The compound noun "repetitive sequence-based polymerase chain reaction" consists of an adjective (repetitive), a past participle of a verb (sequence-based) and three nouns (polymerase, chain, reaction). Thus, biomedical terms tend to consist of a combination of numerals, symbols, and verbs, making it very difficult to find term boundaries. In addition, there can be multiple expressions that are synonymous with a particular technical term. These can arise from abbreviations or acronyms as well as from spelling variations. If these variations are recognized as different entities, it can often cause problems for aggregating documents. For instance, "DNA" and "deoxyribonucleic acid" are synonyms. The dictionary contains spelling and abbreviation variants and their canonical forms. By reducing these variants to a single canonical form, we treat them as the same entity.

In the second step, the text annotated with a technical term dictionary is passed to the syntactic parser. The parser outputs segments of phrases labeled with their syntactic roles, for example NP (noun phrase) or VG (verb group). In the third step, the category annotator assigns categories to the terms in these segments and phrases. The category dictionary consists of a set of canonical forms and their categories, which also indicate the node labels in the category hierarchy (ontology). A category assigned to each term is an internal node or leaf in the hierarchy.

Figure 8 shows an example of the preprocessing of a sentence. When "Repetitive sequence-based polymerase chain reaction effects deoxyribonucleic acids" is given as input, an annotator assigns "DNA" as canonical and "proper noun" as part-of-speech to "deoxyribonucleic acids". After parsing the annotated text, categories are assigned to each term. In Figure 8,

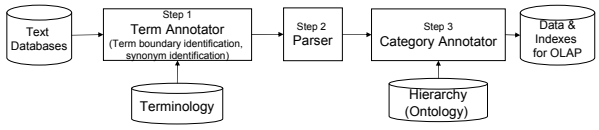


Figure 7: Preprocessing Documents

[7] MEDLINE: http://www.nlm.nih.gov/databases/databases_medline.html
[8] NCBI: http://www.ncbi.nlm.nih.gov
[9] NLM: http://www.nlm.nih.gov/
[10] PubMed http://www.ncbi.nlm.nih.gov/entrez

".A.1.2.23.4" represents a path from a root node to the corresponding node in a category hierarchy.

After preprocessing the 503,989 abstracts, the numbers of $(f, c : v)$, records in the table H, and different terms were 193185919, 340154, and 13640593, respectively. The categories contain categories for publication dates, authors, affiliations for the authors, and so on.

5.2 Implementation using a Document-Term Matrix

To compare with the method mentioned in Section 4, we used a straightforward method with a Document-Term Matrix (DTM) as a proprietary method. In general, a proprietary algorithm and index can compute faster than a method with a persistent store, *e.g.*, the method in Section 4, although it is more difficult to add some functions into the proprietary method and to integrate the proprietary method with other systems compared to the persistent-store method. This subsection explains the method using the proprietary algorithm and index, and the next subsection will explain that the method using the persistent store is comparable to the proprietary method.

This subsection focuses how to compute $\alpha[c, g, count](\sigma'_P(D))$ by a DTM using a simple example, because this is the most fundamental method. Let the sets of terms and documents be $T = \{t_1, t_2, \ldots, t_n\}$ and $D = \{d_1, d_2, \ldots, d_m\}$, respectively. A DTM is a matrix $M = (m_{ij})$ of $m \times n$, and an elements m_{ij} represents how many times the term t_j appears in the document d_i. Although storing the whole matrix requires a lot of memory, it can be compressed by storing a pair for each element that is not zero and its corresponding index in each row or column, because the matrix is very sparse.

As mentioned in the previous section, since some categories are assigned to each term, we use a modified DTM. Rows in our DTM correspond to a set of documents $D = \{d_1, \ldots, d_m\}$ similar to the conventional DTM, and columns correspond to a set of pairs of categories and terms $P = \{c_j : v_{jk} \mid v_{jk} \in dom(c_j),\ j = 1, \cdots, n\} \cup \{c_j : * \mid j = 1, \cdots, n\}$. The value $c_j : *$ is used to facilitate aggregating the number of documents for each subcategory, and an element for d_i and $c_j : *$ is not zero when $d_i \rightsquigarrow c_j : *$.

Figure 9 presents how the method using DTM computes the results for $\alpha[c_3, g^{(1)}, ' count'](\sigma'_{c_1:v_{11}}(D))$, when a user has specified the category c_3 after narrowing down to the documents containing the term v_{11} whose category is c_1. First, the method narrows the search down to the document set $\{d_2, d_6, d_{10}\}$ containing $(c_1 : v_{11})$ (1). In parallel with this Process 1, a set of terms $\{c_3 : v_{31}, c_3 : v_{32}, c_3 : v_{33}, c_3 : v_{34}, c_3 : v_{35}\}$ whose categories are c_3 is output (2). After Processes 1 and 2, the distribution of the documents for the terms appearing in the documents $\{d_2, d_6, d_{10}\}$ is returned as $\{(c_3 : v_{31}) : 2, (c_3 : v_{35}) : 1\}$ (3). Process 3 requires much more computation time than Processes 1 and 2. For example, when the user selects a "commonnoun" category, Process 2 returns 340,154 terms for the dataset used in the experiments described in Section 5.3.

When the user specified a category which is an internal node in the category tree, it also computes the distribution of the documents for each subcategory of the specified category, which corresponds to $\alpha[c_3, g^{(2)}, ' count'](\sigma'_P(D))$. In this case, a set of categories $\{c'_3 : * \mid c'_3 \in$

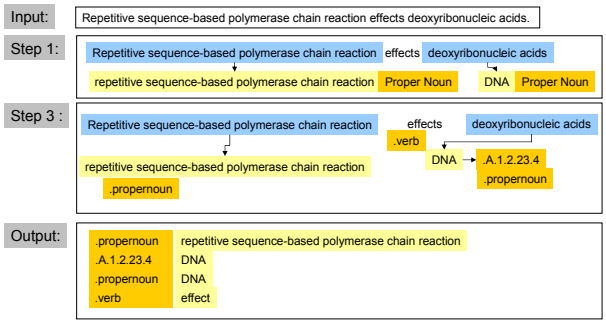


Figure 8: Example of the Preprocessing

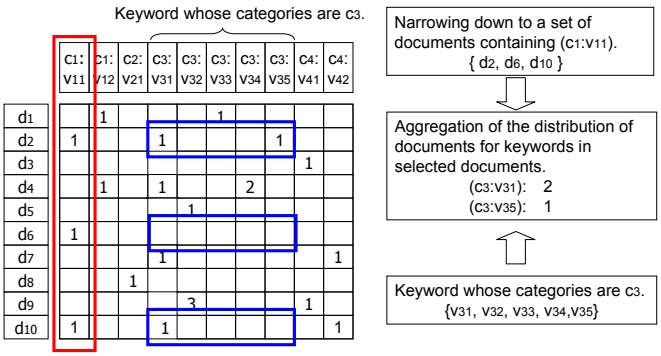


Figure 9: Implementation using a DTM

$succ(c_3)\}$ is returned in Process 2.

5.3 Experimental Comparison

The method in Section 4 was implemented in Java. It generates SQL queries and accesses a relational database via JDBC (Java Database Connectivity). The method in Section 5.2 was implemented in C++ to compare with the above method. For the evaluation experiments, an IBM IntelliStation with Windows XP was used, with an Opteron-2.2 GHz CPU and 2 GB of main memory installed. The efficiency of our approach has been confirmed with respect to the computation time.

Figure 10 shows the results of the response time for a query $\alpha[c, g,' count'](D)$ for all the 340,154 categories. The DB and DTM in the figure correspond to implementations of the methods mentioned in Section 4 and Section 5.2, respectively. KW and SUB represents the cases where $g^{(1)}$ and $g^{(2)}$ as g in $\alpha[c, g,' count'](D)$ were used, respectively. "DTM SUB 1k" shows the results for 1,000 documents sampled randomly from all of the documents. The method of the DB does not compute for the sampled documents but for all of the documents. Each point (x, y) in the figure means that the method returns the result within x seconds for y% of all of the 340,154 categories. The ideal method is at the upper left corner. Although DB could return the result for about 89% of the categories within 0.1 second, DTM could return results for about 60% of the categories for 1,000 sampled documents, and for about 0.01% of categories for 10,000 sampled documents for KW. In addition, $\alpha[c, g,' count'](\sigma_{\top,'cancer'}(D))$ was calculated for the various categories, and the results for 11,914 documents containing "cancer" were similar to Figure 10. As shown in Figure 10, DB is superior to DTM for most of the categories.

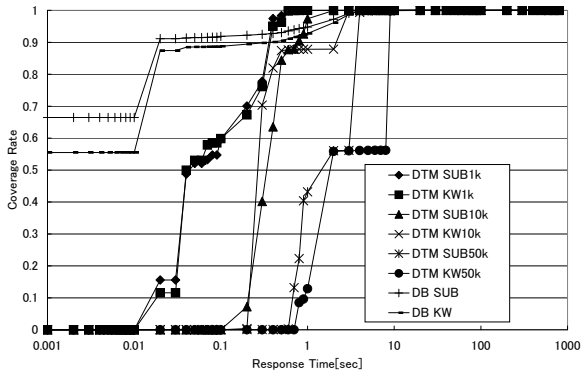
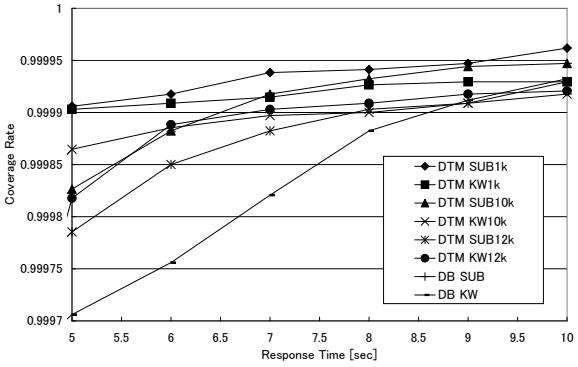


Figure 10: Results for All Documents

From another viewpoint of the empirical 8-second rule saying that a webpage should be

loaded within 8 seconds of a request, we can conclude that the better method is the one whose coverage rate in about 10 seconds is better. Figure 11 focuses from 5 seconds to 10 seconds for the response time and from 99.97% to 100% for the coverage rates for the result $\alpha[c, g,' count'](\sigma_{\top:'cancer'}(D))$. Figure 11 shows that DB is inferior to DTM within 8-second constraint. Tables 3 and 4 summarize the experimental results for $\alpha[c, g,' count'](D)$ and $\alpha[c, g,' count'](\sigma_{\top:'cancer'}(D))$, respectively. Although the average computation time of DB is lower than for DTM, the number of categories for which DB cannot return within 10 seconds may become greater than for DTM.



∗The coverage rate for "DB SUB" was about 0.99775.

Figure 11: Experimental Results for Documents Containing "cancer"

Table 3: Comp. Times for All Documents

Method	avg. comp. time [sec]	10 sec. †	100 sec. ‡
DTM KW 1k	0.143	1	0
DTM KW 5k	0.284	26	0
DTM KW 10k	0.526	16	0
DTM KW 50k	4.293	75	0
DB KW	0.185	10	0
DTM SUB 1k	0.138	2	1
DTM SUB 5k	0.176	2	0
DTM SUB 10k	0.405	5	1
DTM SUB 50k	2.091	33	0
DB SUB	0.137	20	2

† The number of categories for which the results is not returned within 10 seconds. ‡ The number of categories for which the results is not returned within 100 seconds.

As shown in Table 4 and Figure 11, the averages of the computation times for DB are superior to DTM. However, the number of categories for which DB cannot respond within 10 seconds is greater than for DTM. When documents containing "cancer" are selected, an SQL query to obtain V' for $\sigma'_{\top:'cancer'}(D) = (H, V')$ is represented as

$$\begin{aligned} V' &= \sigma_{id\ in\ I}(V) \\ &= \pi_{id,V^{(b)}.preorder,V^{(b)}.value}(V^{(a)} \bowtie_{P_2} V^{(b)}), \end{aligned} \tag{3}$$

where $I = \pi_{id}(V \bowtie_{P_1} H_c)$, $P_1 = (value = 'cancer')$, $P_2 = (V^{(a)}.id = V^{(b)}.id\ and\ V^{(a)}.value = 'cancer')$, and $V = V^{(a)} = V^{(b)}$. The query (3) is represented as

```
SELECT Vb.ID, Vb.PREORDER, Vb.VALUE
```

Table 4: Computation Times for Documents Containing "cancer"

Method	avg. comp. time [sec]	10 sec. †	100 sec. ‡
DTM KW 1k	0.177	24	5
DTM KW 5k	0.355	116	5
DTM KW 10k	0.511	28	7
DTM KW 12k	0.594	27	7
DB KW	0.267	23	0
DTM SUB 1k	0.195	13	1
DTM SUB 5k	0.352	65	0
DTM SUB 10k	0.484	18	1
DTM SUB 12k	0.534	24	2
DB SUB	0.413	706	5

```
FROM V AS Va, V AS Vb
WHERE Va.ID=Vb.ID AND Va.VALUE='cancer'.
```

The reason why DB requires so much computation time for certain categories is the self-join of the table `KEYWORD` V which contains 193,185,919 records. Therefore, we divided the table into multiple tables as follows. Let $id(V)$ be a function which returns a set of document IDs in a table V. We divide V into multiple tables V_i which satisfies $id(V) = \bigcup_i id(V_i)$ and $id(V_i) \cap id(V_j) = \emptyset$ for $i \neq j$. Since the SQL query (3) contains $V^{(a)}.id = V^{(b)}.id$ in its WHERE phase, the following SQL query can avoid joining tables that do not contain the same documents IDs.

$$\alpha[c, g^{(1)}, 'count'](\sigma_{\top:'cancer'}(D)) = {}_{value}\chi_{value,sum(count)}(\bigcup_i R_i), \quad (4)$$

where $R_i = {}_{value}\chi_{value,count(distinct\ id)\ count}(V_i')$, $\bigcup$ represents `UNION ALL` operation, and each V_i' is calculated by the SQL query (3). The SQL query (4) is represented as

```
SELECT VALUE, SUM(COUNT)
FROM (
  SELECT Vb.VALUE, COUNT(DISTINCT Vb.ID)
    FROM V_0 AS Va, V_0 AS Vb
    WHERE Vb.PREORDER=pre   AND
       Va.ID=Vb.ID AND Va.VALUE='cancer'
    GROUP BY VALUE
  UNION ALL
   ...
  UNION ALL
  SELECT Vb.VALUE, COUNT(DISTINCT Vb.ID)
    FROM V_k AS Va, V_k AS Vb
    WHERE Vb.PREORDER=pre   AND
       Va.ID=Vb.ID AND Va.VALUE='cancer'
    GROUP BY VALUE
 ) A
GROUP BY VALUE,
```

where *pre* is a preorder of c.

Figure 12 shows the experimental results when we compared the aggregation with `KEYWORD` divided into 10 tables with DTM and the aggregation using a single table for `KEYWORD`. By dividing the table, the coverage rate within 10 seconds rises from 99.993% to 99.999% for KW, and from 99.79% to 99.93% for SUB. Although these experiments were run with a single computer, we can easily run on multiple computers, because such commercial database systems support parallelization.

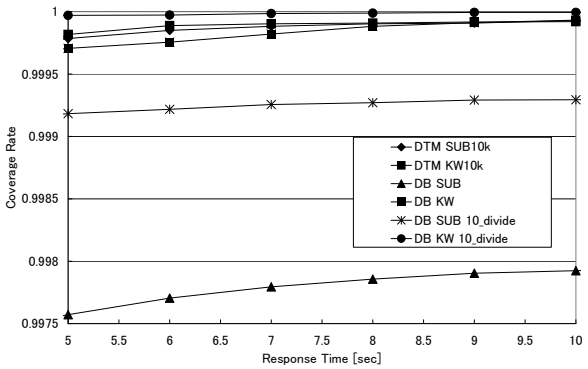


Figure 12: Computation Times for KEYWORD Divided into 10 Tables

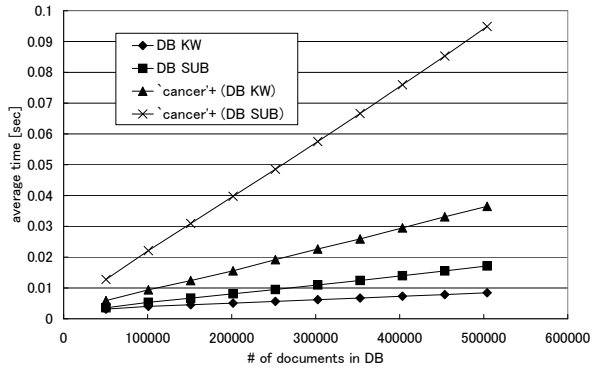


Figure 13: Average Computation Time for the Number of Documents

Figure 13 shows the average computation time for the number of documents for all of the categories. The computation time is observed to be proportional to the number of documents. The high scalability of our method has been confirmed for the amount of data.

Table 5 shows the computation times for $\alpha[c, g, 'count'](\sigma_{\top:'cancer'}(D))$, where each selected category has many different terms in its domain $dom(c)$. This shows that the proposed method can more quickly compute the results for a category compared to the conventional method.

6 Discussion

A top N ranking query often returns a trivial result which contains terms frequently appearing in the documents. To measure characteristic terms, the following relative frequency is used. This measure compares the current document subset to the initial document set. Assume D is the initial document set. A selection operation due to query $c : v$ returns D_s. The relative frequency for a term $p_{jk} = (c_j : v_{jk})$ in the document set D_s is calculated as

$$relative\_frequency(p_{jk}, D_s) = \frac{\frac{c(p_{jk}, D_s)}{|D_s|}}{\frac{c(p_{jk}, D)}{|D|}},$$

where $c(p_{jk}, D)$ is the number of documents that contain the term p_{jk} in the set D. For example in Figure 9 in Section 5.2, $relative\_frequency((c_3 : v_{31}), D_s)$ is $\frac{\frac{2}{3}}{\frac{4}{10}} = \frac{5}{3}$, where D_s is $\{d_2, d_6, d_{10}\}$.

It is very critical to achieve query response times that are as fast as possible to analysis interactively a huge mount of text data. A key strategy for speeding up to aggregate the data

Table 5: Computation Times for Categories to which Many Different Terms Belong

category	# of different terms	DTM KW 5k	DTM KW 12k	DB KW
.commonnoun	340,154	231.2 sec.	238.7 sec.	53.0 sec.
.propernoun	7,903	11.8 sec.	10.8 sec.	2.8 sec.
.verb	32,826	73.9 sec.	74.4 sec.	9.7 sec.
.adj	23,629	61.6 sec.	62.4 sec.	7.2 sec.
.rel_vn	1,337,891	171.3 sec.	177.7 sec.	7.6 sec.
.rel_nv	601,256	85.7 sec.	85.3 sec.	3.6 sec.
.rel_vnn	5,439,810	145.6 sec.	162.9 sec.	9.8 sec.
.rel_nvn	2,294,012	99.3 sec.	101.2 sec.	4.4 sec.
.rel_nnv	1,812,501	90.7 sec.	89.5 sec.	3.8 sec.
.affiliation	282,729	53.7 sec.	55.3 sec.	3.2 sec.
.pnsubstance	48,512	66.2 sec.	68.0 sec.	4.3 sec.
.majormesh	89,275	83.6 sec.	85.1 sec.	3.1 sec.
.minormesh	111,761	126.2 sec.	128.5 sec.	7.8 sec.
.chemical	10,146	17.9 sec.	18.2 sec.	3.1 sec.
.genesymbol	15,310	48.1 sec.	48.8 sec.	7.3 sec.
.protein	15,562	57.1 sec.	58.2 sec.	6.7 sec.
.biomedicalterms	91,670	133.7 sec.	137.8 sec.	21.1 sec.

is to use indexing technology. As mentioned in Section 4, we use the preorders and postorders to check ancestor-descendent containment in a category hierarchy. If we do not use preorders and postorders in a traversed tree, we need to join the table `CATEGORY` n times to check where a node A is an ancestor of a node B, where n is the length of a path from the A to the B. However, we do not need any join operations to ancestor-descendent containment in the hierarchy.

The method to index the tree was proposed in 1982, and it recently draws attention as the method to index XML (eXtensible Markup Language) database and to map XML data into the relational database [9], since each XML document is modeled as a DOM (Document Object Model) tree. Several Methods such as prefix label [6], Dewey order [23], prime label [24], VLEI code [12], embedding into a k-ary tree [13] are used to index XML. A disadvantage of the methods such as preorder-postorder method and prime label is to need to re-assignment of preorders and postorders of nodes when inserting some nodes into a tree. Since each node has the same label as a prefix of its children in the methods such as prefix label and Dewey order, they do not need to reassign the labels when inserting some nodes. However, because they need to compute functions to process string to check ancestor-descendent containment, they need more computations time than the preorder-postorder method. Since we assume that a category hierarchy is rarely updated and records in the table `KEYWORD` V are often inserted, we used the preorder-postorder method to index the category hierarchy.

Our data representation is similar to the bag-of-words approach [14], although each term is assigned categories. In the bag-of-words approach, the following sentences are treated as the same content,:

- "(a) X did fail",
- "(b) X did not fail", and
- "(c) Did X fail?" [17].

Besides the negation and interrogative mood, some auxiliary verbs such as "can" and some verbs such as "want" often indicate the author's communicative intentions. It is important to associate communicative intentions with predicates by analyzing grammatical features and lexical information. "fail" in the previous examples (a) to (c) are assigned categories and restored in R as (a) complaint:fail, (b) commendation:not_fail, and (c) question:fail. These

extractions are instrumental in facilitating problem detection and workload reduction for analysts at customer help centers.

Some papers such as [15] and [16] proposed OLAP systems to analyze a set of documents. The following MDX (Multi-Dimensional eXpression) query which was used in [15] finds all the documents which contain a term "forests" and are published in New York in the first quarter of 1998.

```
SELECT not empty [DocId].members on rows,
    {[Measure].[Tf]} on columns
FROM docInfo
WHERE ([Term].[forest],[1998][quarter 1],
                        [location].[New York])
```

In the query, `[Term].[forest]` means that a depth of a hierarchy in a TERM dimension is 2, and term "forests" is a child node of "TERM" corresponding to $\top$ in the TERM dimension. Hierarchies that the existing OLAP systems for texts assume is so simple that it is difficult to integrate the hierarchies with a complex ontology with a huge set of nodes.

Other papers proposed an OLAP system to analyze a set of documents [22, 21]. Since a sales written in multiple documents (newspapers) would be a fact in the papers, operations in the system is similar to the conventional OLAP system for structured data. An example operation in the system is to analyze the average sales per product and year. In our case, since a document would be a fact, we can find a change in the number of documents with time. For example, in a call center in a company, call takers make reports of each call by typing in customer information such as name and phone number, selecting call categories such as "technical QA" and typing in brief descriptions of questions or messages from the customer and brief descriptions of answers and/or actions taken. The brief descriptions are writtin in natural language. The manager of the call center wants to improve productivity, reduce cost, improve customer satisfaction, etc. For example, in a large number of documents related to customers' calls, we want to find what kinds of topics have recently been increasingly mentioned and which product is associated with specific topics, so that we can take appropriate actions for the improvement of call center productivity and product quality, or create a FAQ (frequently asked question) database.

7 Conclusion

In this paper, we proposed a data model, and a relational algebra to integrate ontologies with OLAP systems to analyze a huge set of textual documents. The proposed method was implemented with a persistent store using preorders and postorders in a hierarchy. The efficiency of our approach has been confirmed with respect to the computation time.

References

[1] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proceedings of 22th International Conference on Very Large Data Bases*, pages 506–521, 1996.

[2] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. Spatio-temporal retrieval with rasdaman. In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 746–749, 1999.

[3] P. A. Bonatti, Y. Deng, and V. S. Subrahmanian. An ontology-extended relational algebra. In *Proceedings of the 2003 IEEE International Conference on Information Reuse and Integration*, pages 192–199, 2003.

[4] V. T. Chakaravarthy, H. Gupta, P. Roy, and M. K. Mohania. Efficiently linking text documents with relevant structured information. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 667–678, 2006.

[5] E. Charniak. *Statistical Language Learning*. The MIT Press, Cambridge, Massachusetts, 1996.

[6] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic xml trees. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 271–281, 2002.

[7] P. F. Dietz. Maintaining order in a linked list. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pages 122–127, 1982.

[8] S. Goil and A. N. Choudhary. High performance multidimensional analysis of large datasets. In *Proceedings of International Workshop on Data Warehousing and OLAP*, pages 34–39, 1998.

[9] T. Grust. Accelerating xpath location steps. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 109–120, 2002.

[10] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for olap. In *Proceedings of the Thirteenth International Conference on Data Engineering*, pages 208–219, 1997.

[11] M. Gyssens and L. V. S. Lakshmanan. A foundation for multi-dimensional databases. In *Proceedings of 23rd International Conference on Very Large Data Bases*, pages 106–115, 1997.

[12] K. Kobayashi, W. Liang, D. Kobayashi, A. Watanabe, and H. Yokota. Vlei code: An efficient labeling method for handling xml documents in an rdb. In *Proceedings of the 21st International Conference on Data Engineering*, pages 386–387, 2005.

[13] Y. K. Lee, S.-J. Yoo, K. Yoon, and P. B. Berra. Index structures for structured documents. In *Proceedings of the 1st ACM International Conference on Digital Libraries*, pages 91–99, 1996.

[14] C. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, Massachusetts, 1999.

[15] M. C. McCabe, J. Lee, A. Chowdhury, D. A. Grossman, and O. Frieder. On the design and evaluation of a multi-dimensional approach to information retrieval. In *Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 363–365, 2000.

[16] J. Mothe, C. Chrisment, B. Dousset, and J. Alau. Doccube: Multi-dimensional visualisation and exploration of large document sets. *Journal of the American Society for Information Science and Technology*, 54(7):650–659, 2003.

[17] T. Nasukawa and T. Nagano. Text analysis and knowledge mining system. *IBM Systems Journal*, 40(4):967–984, 2001.

[18] T. Niemi, J. Nummenmaa, and P. Thanisch. Logical multidimensional database design for ragged and unbalanced aggregation. In *Proceedings of the 3rd International Workshop on Design and Management of Data Warehouses*, page 7, 2001.

[19] T. B. Pedersen and C. S. Jensen. Multidimensional data modeling for complex data. In *Proceedings of the 15th International Conference on Data Engineering*, pages 336–345, 1999.

[20] T. B. Pedersen and C. S. Jensen. Multidimensional database technology. *IEEE Computer*, 34(12):40–46, 2001.

[21] J. M. Pérez, R. B. Llavori, M. J. Aramburu, and T. B. Pedersen. A relevance-extended multi-dimensional model for a data warehouse contextualized with documents. In *Proceedings of ACM 8th International Workshop on Data Warehousing and OLAP*, pages 19–28, 2005.

[22] J. M. Pérez, T. B. Pedersen, R. B. Llavori, and M. J. Aramburu. Ir and olap in xml document warehouses. In *Proceedings of 27th European Conference on IR Research*, pages 536–539, 2005.

[23] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 204–215, 2002.

[24] X. Wu, M.-L. Lee, and W. Hsu. A prime number labeling scheme for dynamic ordered xml trees. In *Proceedings of the 20th International Conference on Data Engineering*, pages 66–78, 2004.