Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Václav Nidrle

Vektorová DIS cartridge pro Oracle
Vector IRS cartridge for Oracle

Department of Software Engeneering
Supervisor: RNDr. Michal Kopecký, Ph.D.
Study program: Computer Science

I would like to thank my supervisor, RNDr. Michal Kopecký, Ph.D., for his valuable advice. Special thanks belong to Bc. Zuzana Nidrlová, who has enhanced this thesis to a higher linguistic level. Also I would like to thank my parents for their permanent support during my studies. Finally I would like to express my gratitude to my girlfriend Melinda for standing by me at all times.

Contents

Název práce: *Vektorová DIS cartridge pro Oracle*
Autor: *Václav Nidrle*
Katedra (ústav): *Katedra softwarového inženýrství*
Vedoucí diplomové práce: *RNDr. Michal Kopecký, Ph.D.*
e-mail vedoucího: *michal.kopecky@mff.cuni.cz*
Abstrakt: *Oracle Text je dokumentografický informační systém, který tvoří nedílnou součást databáze Oracle. Je postaven na základním boolském modelu rozšířeném o další možnosti jako jsou přiřazení vah jednotlivým termům, výpočet podobnosti dotazu a dokumentů nebo fuzzy logika.*
Jako nadstavba Oracle Text je implementována nová databázová cartridge Vector Text, její základní filosofii však tvoří vektorový model. Díky tomuto modelu jsou tak vylepšeny možnosti řízení velikosti výstupu a poměřování relevantnosti vracených dokumentů. Pro tyto účely je ve Vector Text připraveno mnoho funkcí pro výpočet váhy termu a míry podobnosti dotazu s dokumenty a nechybí také jednoduché rozhraní pro jejich další rozšiřování. Uživatelské rozhraní cartridge Vector Text maximálně zachovalo podobu i principy použité v Oracle Text. Dále byla ponechána a navíc rozšířena možnost využití některých speciálních funkcionalit Oracle Text použitelných v obou modelech, jako jsou např. tezaurus nebo slovník stop slov.
Klíčová slova: *DIS, vektorový model, Oracle Text*

Title: *Vector DIS cartridge for Oracle*
Author: *Václav Nidrle*
Department: *Department of Software Engeneering*
Supervisor: *RNDr. Michal Kopecký, Ph.D.*
Supervisor's e-mail address: *michal.kopecky@mff.cuni.cz*
Abstract: *Oracle Text is an information retrieval system, which is included in Oracle database. It is built up on the basic Boolean model being further extended by features like assigning weights to individual tokens, calculating similarity of query and documents or fuzzy logic.*
The new database cartridge called Vector Text is implemented as the extension of Oracle Text, however it is based on the Vector space model. Thanks to this fact the possibilities of controlling the output amount and measuring the relevance of the returned documents are improved. For this purpose there is prepared a lot of functions for calculating token weight and similarity measure of query and document in Vector Text. The simple interface for their further extensions is also present there. The user interface of Vector Text cartridge kept the same principles used in the Oracle Text. Furthermore, the possibility of using special functionalities of Oracle Text being applicable in both models like thesaurus or stoplist have been preserved.
Keywords: *IRS, Vector space model, Oracle Text*

Chapter 1

Introduction

The history of the information retrieval (IR) is very long, however the IR used to be for a substantial period of time regarded an activity that would engage only a few people such as librarians. With the recent boom of PC know-how and the Internet, hundreds of millions of people got the access to the enormous amount of information which they needed to be able to search effectively. That is why the IR-enabling systems and systems serving up the results of IR to the user had to be developed. These systems specialized in the IR in text documents are known as the Information Retrieval Systems (IRS).

Oracle is one of the largest IT firms specialized for developing and offering systems based on its own object-relational database system. The growing need of IR forced the integration of the special full-text retrieval module into their database. This module is called Oracle Text (formerly known as interMedia Text in Oracle database version 8 and Oracle ConText in database version 7). It uses the SQL extensions to index, search and analyze text and documents stored in different locations. Its core is based on the basic Boolean model which works on the principle of inverted index and queries including words connected by boolean connectives. During the years in which the Oracle Text was developed, a great number of extending features was progressively implemented which often came out of the extended Boolean model principles.

1.1 Goals

The elementary goal of the work is to implement an easy-to-use extension of Oracle Text using IR Vector space model. The Vector space model works on the principle of projecting documents and queries as the vectors into n-dimensional spaces and calculating of the similarity of such vectors. The emphasis should be placed on the Vector space model implementation where the advantages of Vector space model approach compared to the Boolean model approach should become evident. The resultant extension should offer a useful tool for the distinct IR methods comparison and for the confrontation of usability and results of different similarity measures in concrete situations.

The future users of the new extension would certainly appreciate the user interface of the extension being kept as similar to Oracle Text as possible. It would make the process of mastering the use of the new extension easier for those already being familiar with the Oracle Text. That is why the projected extension should also preserve as much existing Oracle Text functionality as possible.

1.2 Structure of the Thesis

At first, there is a brief description of Information Retrieval (IR) and Information Retrieval Systems (IRS) given in the Chapter 2. This chapter also introduces an overview of existing IR models and describes the most common IRS problems together with the techniques being used for their solutions.

The thesis then follows by the description of principles of both the Boolean model as well as the Vector space model. This description, together with the discussion of both the advantages and disadvantages of each model, form a content of the Chapter 3.

The fundamentals of Oracle Text features and their implementation with a lot of examples are described in the Chapter 4. This chapter discuss also its disadvantages and mentions key reasons for implementing the Vector Text cartridge.

Afterwards, the Chapter 5 introduces the new Vector Text cartridge. It starts with the discussion of possibilities of solving different implementation problems. Then it follows by the description of Vector Text design and processing and finally introduces all implemented similarity measure and token weight methods.

Chapter 6 introduces the test data, describes used test environment and presents achieved results.

The user documentation for Vector Text cartridge can be found in the Appendix A. It consists of the installation manual and user's reference describing all implemented statements, operators and packages with a lot of examples for their simple usage.

Chapter 2

Information Retrieval

The general definition of the *information retrieval* according to Manning [7] says: "Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfy an information need from large collections (usually on local computers, servers or on the Internet)“. But the use of IR in the practice shows that this definition is not fully accurate. IR research has focused mainly on the retrieval of natural language texts including books, news articles or email messages and all these documents has some structure, e.g. each book has an author, a publisher, it can be divided into chapters etc. However each of its parts contains an amount of unstructured text and so all these documents are qualified as semi-structured.

The main goal of IR is to find documents in a given collection that are dealing with a given topic expressed by a user's query. The given collection can be divided in two parts. One part includes the documents that fulfill the user's demands and needs, these documents are called *relevant*. The other part contains documents that are not about the given topic and they are called *non-relevant*. The same collection can be also divided by the IR engine in two different parts, one of which includes the documents that the engine pronounces to be relevant for the user according to his query and the other one is built up of those documents seen as non-relevant according to the engine. The question is, does the IR engine return the proper documents?

There are used two measures of success in the IR which are based on the concept of relevance. They are called *precision* and *recall*. According to Saracevic [16] the precision is defined as the ratio of relevant items retrieved to all items retrieved, or the probability given that a retrieved item will be relevant. The recall is then defined as the ratio of relevant items retrieved to all relevant items in a collection, or the probability given that a relevant item will be retrieved. These two measures are contradictory. When all documents from the collection are returned to the user then all relevant documents are fetched and the recall equals one. But on the other hand all non-relevant documents are returned as well and because the number of relevant documents is typically very small in comparison to the size the document collection, the precision is low (close to zero). Ideally the user would like to retrieve all the relevant documents and only them. Normally the user wants to achieve the best combination of good precision and good recall.

The *Information Retrieval System* (IRS) is the system for offering information retrieval services. It is build and developed for the purpose of work with the textual data, their holding, processing and retrieving. The typical IRS works on the following principle:

1. user enters the query
2. the query is compared to the collection of documents
3. the documents regarded by the system as relevant according to given query are then returned to the user
4. the user optimizes the query and repeats the steps 1.-3. according to his satisfaction with the returned results
5. the user asks for the detail of the resultant document
6. the user receives the document

2.1 IR models summary

This section serves the basic overview of different approaches to IR and presents the list of existing IR models.

It is hard to construct the taxonomy of all known IR models. In this case the main distribution according to Kraaij [3] is applied - by comparing the way in which the notion of relevance is treated in individual models. The main classes of IR models according to the criteria given are the logical models, the Vector space models and the probabilistic models. Not all the existing models can, however, be assigned to one of these three classes. Some models can be based on other approaches such as neural networks, genetic algorithms etc, but such approaches are more suitable for the information filtering or routing tasks and that is why they are not dealt with further in this text.

2.1.1 Logical models

The earliest and the most known logical IR model is the *Boolean model* which was introduced in the 1950s. Its basic idea is to represent each document by a set of index tokens, which are included in given document. The query is then expressed by using this index tokens and operators from the Boolean algebra: conjunction, disjunction and negation. Every query is thus a sententional formula. If and only if this formula holds for a document, the document is then considered to be relevant by the IR system and it is therefore retrieved. Besides, every query can be rewritten in a disjunctive normal form, because that way it can be efficiently evaluated for each document. The Boolean model is the only one which ignores uncertainity and is not based on the ranking of relevance. It only divides the documents set into two classes one of which answers the query and the other does not. More details about Boolean model can be found below in the Section 3.1.

Some extensions of the Boolean model were introduced with the purpose of taking out the disadvantage of missing relevance ranking. The *Co-ordination Level Matching* [15] defines the retrieval status value as the number of unique query terms found in the document. The higher this number, the higher the co-ordination level by which the result set of documents is being ordered.

The *Proximity Matching* [1] extends the previous method by enhancing the precision of retrieval result. It processes the cover density ranking which means calculating a relevance based on the distance between query tokens in the document. The closer the tokens are, the more relevant is the document for given query. This method is especially suitable for short queries.

The *Fuzzy Set* model [13] comes with a different evaluation of term membership in document. Unlike in Boolean models, where the term-document membership values are binary, fuzzy membership values range between 0 and 1. The advantage of this approach is that the uncertainty and its degree can be encoded there.

The *Extended Boolean* model [14] integrates token-weighting and distance measures into the Boolean model. Like in Fuzzy Set model, the index tokens can be weighted between 0 and 1. The boolean operators are modeled as similarity measures based on non-Eucledian distances in a V-dimensional space. Despite the sophisticated concept, extended Boolean models have not become too popular. They are less clear for the user because of various semantic changes. For longer queries the Vector space or probabilistic models are being preffered [7].

The logical models class got a new impulse in 1986 when van Rijsbergen introduced non-classical logics [17] on which the framework for IR can be built. He showed that different retrieval models (Boolean,probabilistic) can be re-expressed as a computation of logical implication. The basic conception is that a retrieval of relevant documents can be expressed by the implication $q \leftarrow d$ where d stands for the document and q presents the query. Because of the uncertainity involved, this is not a material implication in first logic order, but it requires a conditional logic. Van Rijsbergen's work stimulated the renewal of interest in logical and uncertainity models for IR and so a lot of new models were presented during the 1990s.

2.1.2 Vector space models

The first ideas for a text representation based on weighted term vectors were presented in the late 1950s by Luhn [6]. His ideas were then further developed by Gerard Salton during the following thirty years.

For the *Vector space model* (VSM) the relevance of a document for a query is defined as a distance measure in a high-dimensional space. The distance measure actually serves as a metric to compute the similarity between queries and documents. In order to compute this similarity measure it is necessary to firstly project documents and queries in the high-dimensional space defined by the vocabulary of index tokens. The main assumption of Vector space model is the token independence, but this can not be reached in practical use. The common practice is to circumvent this dependency problem. More details about Vector space model including the basic model formalization are to be found below in the Section 3.2.

In 1990, Deerwester et al. [2] introduced the *Latent Semantic Indexing model* (LSI) based on approximation of data dependency set with a model with fewer dimensions. The dependence is detected here by the co-occurrence of tokens. If the tokens co-occur in a document frequently then they might be semantically related. The co-occurring tokens are projected onto the same dimension of the document-by-term matrix, independent tokens onto different dimensions. Finally the singular value decomposition is applied to this matrix. The synonyms are projected onto the same dimension and so the recall of a query is significantly improved. LSI was not implemented in many IR systems because it is computationally very demanding, the resulting dimensions are hard to characterize and token co-occurrence can be exploited in a cheaper way (eg. by applying pseudo-relevance feedback).

The *Generalized Vector space model* (GVSM) [18], [19] uses as the basic indexing unit so called *generalized term* which is defined as a set of binary token weights for each token in the collection. A collection with vocabulary size of L tokens yields 2^L possible generalized terms corresponding to all possible patterns of co-occurrence. Both queries and documents are mapped into this space. The generalized term vectors are linearly independent and orthogonal, while the index tokens can be dependent. GVSM produces slightly better results on small collection than standard VSM based on binary indexing scheme. However, for the purpose of general use the standard VSM is still more suitable.

2.1.3 Probabilistic models

The probabilistic models exploit different distributions of tokens in the class of relevant and the class of non-relevant documents. They use various probabilistic and statistic methods for it.

Probabilistic Relevance models [8] try to estimate the relevance of a document directly based on the idea that query tokens have different distributions in relevant and non-relevant documents. These models presuppose having available relevance information (for instance in a routing task), otherwise they are equivalent to VSM using inverse collection frequency weighting. The most widely known Probabilistic Relevance models are the Binary Independence Retrieval model and 2-Poisson distribution based model.

Inference based models are blended between logic and probability theories. They are highly extendible, collection independent and they include two subclasses: *In-*

ference networks which apply Bayesian inference for the computation of relevance score and *Probabilistic inference models* which are based on non-classical logics and probability theory.

In the late 1990s a new class of IR models turned up - the *Language models*. Basically they are modeling the probability that given document could be the basis for the user's query instead of modeling the probability of relevance. As the examples of this class can be mentioned the *Ponte and Croft model* [12] or the *Hiemstra and Miller et al. models* [9], other new models are still under development now.

2.2 Common IRS tasks and problems

Presented in this section are the most common tasks and problems which are being solved in the majority of IR systems and should be addressed by the proposed cartridge.

Tokenization

The process of tokenization is basically dividing the content of a document into individual words, which are called *tokens*. It also includes throwing away certain characters, e.g. white space, punctuation etc. But the question is how to find the right characters to be taken away? This decision is clearly content and language specific as in some document the character in question should be thrown away, while in others its preservation is desirable. For instance the dot character should be left out from a usual text as opposed to the cases where the dots represent an essential part of IP or email address.

Another problematic step of tokenization process is collocations identification where collocations should be regarded as a single token (e.g. New York, white space). One solution is the use of the phrase index, e.g. biword index where each index token consists of the two following tokens from the document. In this case the queries longer than the ones including two tokens can be afterwards rewritten to the conjunction of several collocations being composed by two tokens. Another and much more common solution is the use of a positional index where a list of positions of their occurrences in given document exists for each token of the index dictionary. The positional index is very often used in inverted indexes.

A lot of languages create new words by compounding several shorter words in one long word. For instance the German language creates compounds very often (e.g. Computerlinguistik). Some east Asian languages (e.g. Chinese, Japanese) even do not have any spaces between words and so there is a need of word segmentation. It can be processed using large dictionaries and taking the longest matching word from the dictionary with some heuristic for unknown words. The different approaches are the use of machine learning sequence models (hidden Markov fields) or indexing via short subsequences of words or characters called *character k-grams*.

Stop words

The term *stop word* means the token which is too common and non-selective in the collection of documents and therefore it should be excluded from the indexing. Key searching using such a word would not be efficient in the most of cases anyway. The stop words are generally recognized by the highest frequencies of their occurrence in the collection of documents and they are collected into the *stop lists*. The use of stop list significantly reduces the number of entries that an IR system has to store, e.g. excluding the 30 most frequent words reduces the document data size by approximately 30% [7].

Token normalization

The token normalization is involved in the process of tokenization to be able to properly decide if given token in query matches the token from the document because the query need not be entered using exactly the words used in the relevant text.

The problem of token normalization is generally being solved by creating the equivalence classes of words. Hence the set of general equivalence rules is defined which specify for instance removal of hyphens, dots, single quote marks etc. Each unnormalized token is then processed according to those rules and the resultant normalized tokens are creating the equivalence classes.

The other way to process the token normalization is by creating relations between unnormalized tokens. It can be achieved by the two methods. The first one creates an index with unnormalized tokens and then maintains a query expansion list which results in the query containing disjunctions of all possible equivalents for each used word. The alternative method performs a contraction during the index creation. All equivalent tokens are indexed under one given representative. Both of these methods are less efficient then the equivalence classing, by contrast they have better results for synonyms processing.

Another issue to deal with during the tokenization is the capitalization. The differences between the upper and the lower case of characters in tokens need to be cleared out to decide if the words match. The most common strategy is to reduce all the letters to the lower case which solves the differences caused by e.g. use of a capital letter at the beginning of a sentence or entering the whole query in the lowercase. This strategy brings some problems with personal names identification (e.g. Bush vs. bush, Bean vs. bean) and that is where the alternative method of truecasing can be used. It is based on a machine learning sequence model for deciding when to case-fold which reduces to lower case only tokens according to adjusted rules. But even the truecasing is not almighty, it does not help in situations where the user enters in the whole query in the lower case.

Each language has its other specific features and issues (e.g. diacritics, accents etc.) which need to be handled by the token normalization. The normalization has to be processed in terms of the used language. Generally speaking, the token normalization is in most of the cases helpful yet sometimes it use results in unexpected consequences.

Stemming and lemmatization

The goal of both stemming and lemmatization is to get rid of different inflectional and grammatical forms of the words in text. These two methods convert such words into their common base form, but they vary in their way of doing so. The stemming performs the heuristic analysis and separates the ending of the words, while the lemmatization converts the words into their basic form by using the directory and morphological analysis of the words.

The results of using these methods are ambiguous. It helps a lot for some queries. However it can greatly reduce the performance of other queries. In general, the stemming is more convenient for treating the derivationally related words, whereas the lemmatization is more suitable for converting the different inflectional forms to the basic form. The stemming also enhances the recall to the prejudice of precision.

Inverted index creation

The searching in unstructured documents would be very inefficient and that is why the inverted index structure optimized for holding indexed data is usually used by the IRS systems. The basic format of inverted index consists of dictionary of tokens. A list of documents the token occurs in is saved there for each token. The tokens in dictionary are usually alphabetically sorted. Each document in the list is identified by its explicit id and the list is sorted by those ids. Commonly, the token frequency from the whole document collection is holded separately for each token in the dictionary. The token frequency of the token in the given document is also saved for each document in the list together with the document id. Additionally the list of positions of all occurrences in the document for the token can be also saved.

Thesaurus

The thesaurus represents a set of words with the defined semantic relations between them. Such relation can be either hierarchical (e.g. broader term which means more general token, narrower term which means more specific token) or it can describe synonyms or near-synonyms. IRS often use thesauri for automatic query expansion by which it solves the problem of synonyms. Thesaurus can also be offered to the users for entering more complex queries.

Chapter 3

Boolean vs. Vector space model

3.1 Boolean model

Let's introduce the basic *Boolean model* more deeply here. As mentioned in the previous chapter, a Boolean query is an expression in which a set of tokens is combined by the logical operators of conjunction, disjunction and negation. For illustration, the query t_1 *AND* t_2 is satisfied by a given document D_1 if and only if D_1 includes both tokens t_1 and t_2. Similarly, the query t_1 *OR* t_2 is fulfiled by D_1 if and only if it contains token t_1 or token t_2 or both of them. And finally, the query t_1 *AND NOT* t_2 satisfies D_1 if and only if it includes t_1 and does not include t_2. Any more complex Boolean query can be built up out of these operators and evaluated according to the rules showed above. For the purpose of more efficient evaluation, every query can be rewritten in a disjunctive normal form. It means that the individual tokens or tokens grouped by the AND operation are ORed together. The evaluation is easier for such form because it is enough to find only one part of the query which is satisfied by the document. The whole query will be thus satisfied as well with the help of the disjunction. The classical Boolean operator evaluates its arguments and returns a value of either true, respectively one (document matches the given query) or false, respectively zero (document does not match given query). The concept of relevance in the Boolean model is binary, in the sense that a document is either relevant or it is not to a considered query.

The basic Boolean model, as it was introduced up to now, has many fundamental disadvantages. The possibility of ranking of relevant documents according to their relevance does not exist here. The documents are only divided into two groups as relevant and non-relevant ones. This problem also implies the difficulty connected with the output size control, because when user reduces the amount of relevant document to be retrieved, the most relevant ones will not necessarily be retrieved in all the cases. Boolean operators cause also another trouble with output control. The AND operator is too severe, e.g. it does not distinguish between the situations where none tokens are satisfied and those where all except one are satisfied, whereas the

OR operator does not differentiate enough. It results frequently in a situation where either all or no documents are retrieved.

Many different extensions were thus introduced to remove the mentioned disadvantages of Boolean model. The user can specify one of proximity operators, which check if the distance between two tokens in the document satisfies the query condition. In general, the proximity operators specify a unit (e.g. word, sentence or paragraph) and a value of maximal distance, which still satisfies the query condition. Based on this fact there can be specified queries such as that two tokens have to be located in the same sentence or that the distance between two tokens has to be maximally 5 words. When the proximity operators are required, the inverted index has to include the exact positions of token occurrences in the documents. The relevance ranking can be built on the proximity in the Boolean model. Its basic idea suggests to measure the relevance of documents by the proximity of tokens that specifies the query. The closer the tokens are, the more relevant the individual document is.

Another relevance ranking, which is very common in Boolean model implementations, is based on the frequence of token ocurrence in individual documents. It is assumed that the higher the number of occurrences is, the more important is the token for the document. The measure of relevance of the individual document can be calculated by the summation of token frequencies of all tokens being specified in the query. The exact value of the *token frequency* is defined as the frequency of occurrences of the given token within the given document divided by the total number of all tokens occurrences in the document.

More additional features can be built into the Boolean model. For instance the principles of the uncertainity and fuzzy logic can extend the possibilites of the Boolean model. In that case the membership value of each token in individual document is enlarged from the binary values to the range between 0 and 1. The measure of assumption and beliefs can be expressed thanks to it. However, the evaluation of Boolean operators has to be changed to the minimum or maximum of the membership values.

The Boolean model is used inside the majority of IRS because its concept is easy to implement and at the same time it is computationally efficient. The Boolean retrieval works best when a query requires selection of an entire and definite result set. Other advantages of Boolean queries are their expressiveness and clarity. The rules applied are always very simple, e.g. the synonyms are specified using disjunction, whereas the phrases are grouped by conjunction.

On the other side the Boolean model has some more disadvantages except the ones being introduced before. The Boolean operators are the reason for some users' difficulties with constructing the query. The meaning of the operator concerning the query is different from the use in a natural language. The statement A and B can in a conversation refer in to more entities then A used alone would, while in IR the number of documents referred to will be lower then it would be when retrieved by A alone. By contrast, the A or B statement refers in conversation either to A only or to B only, whereas IR would retrieve all documents including either one of the two entities or both of them. The query formulated in natural language cannot be used in Boolean model and thus neither citations nor abstracts can be used to enter the query easily.

3.2 Vector space model

The *Vector space model* has been developed as the successor of the Boolean model. It attempted to remove the most prominent disadvantages of the Boolean model. As stated in the previous chapter, the key idea is based on representing the documents as vectors in high-dimensional spaces.

After all the tokens are extracted from a given document, their normalization is done, stop words are eliminated and stemming is processed, the resultant set of tokens defines a document space so that each distinct token represents one dimension in that space. Every token in the document is evaluated by a numeric weight, representing usefulness of given token as a descriptor of the given document. The rule used for evaluation says that the more useful token, the higher weight value. To formalize this concept, each document D_i can be represented by vector

$$D_i = (w_{i,1}, w_{i,2}, ..., w_{i,n})$$

where n stands for the total count of different tokens which occur in the query or document collection and $w_{i,j}$ means the weight of j-th token in i-th document from the collection.

The user can specify a query as a set of tokens each associated with numeric weight. It may be even specified in natural language and in this case the query can be processed exactly like a document (including removal of stop words, stemming etc.). This results into the fact that a query can always be interpreted as another document in document space and so each query Q can represented by vector

$$Q = (q_1, q_2, ..., q_n)$$

where n stands for the total count of different tokens which occur in the query or document collection and q_i means the weight of i-th token in query.

Now the question remains how to distinguish the relevancy of the documents for the given query. Both document and query vectors can be projected into the n-dimensional space created by all tokens from the query and document collection. The relevance of documents can be measured there by comparing the similarity of the document and query vectors. The set of documents with high similarity is expected to include a high proportion of the relevant documents, while the set of documents with very low similarity will probably include very few relevant documents. The usual similarity measure employed in document vector space is the *inner product* between the query vector and a given document vector [13] [14]. Let q_j be the j-th element of query vector, let D_i be the i-th document in collection and let $w_{i,j}$ be the j-th element of document vector for i-th document. The similarity measure calculating the inner product is then given by the formula

$$sim(Q, D_i) = \textstyle\sum_{j=1}^{n} q_j * w_{i,j}$$

The inner product is not often used practically because it for instance does not take into account the length of the document (long documents have higher similarity values). See the Section 5.5 for the examples of different similarity measures being used in practise along with their brief descriptions and comparation.

The variety of weighting schemes have been used so far. Manual assigment is expensive and unrealistic for large collections, that is why the automatic generation of weights is being processed. Weights based on the token frequency values are used most commonly. The usage of token frequency brings some disadvantages, e.g. it does not take into consideration the length of document (long documents have higher frequencies of tokens) or all resultant values of token frequency are very low. That is why a great deal of other weighting schemes were developed in attemp to eliminate the above mentioned problems. They use normalization techniques which are being processed with regard to the document or the whole collection. See the Section 5.6 for the examples of different weighting schemes along with their brief descriptions and comparison.

The Vector space model enables good ranking of retrieved documents. They can be retrieved sorted from the most relevant ones having the highest similarity values. That further results into an easy control of the output amount since the strict amount of the most relevant documents can be specified. The weighting of index query tokens brings the possibility to prefer the importance of one query token to others. User has newly the opportunity to retrieve the documents most similar to the specified one. This document can be used directly in the query and the similarity of other documents in collection is being evaluated.

The Vector space model is handicapped by the fact that there is a lot of different methods used to calculate token weight and document similarity values. To choose the suitable ones for each searching conditions according to the specified query, collection, computational effectiveness of the methods etc. is thus crucial. The phrase queries can be somewhat tricky, too, because the relative order of tokens is lost in the vector space index, thus this information cannot be used in the process of calculation of the similarity measure values.

Chapter 4

Oracle Text

4.1 Oracle data cartridge fundamentals

Oracle data cartridge is a mechanism being used for extending the potentialities of the Oracle server. It consolidates specific datatypes, functionalities and features in a single, compact, reusable component. The data cartridge is server-based. This means that all its parts and the most of its processing reside on the server.

The most frequent reason for production of a data cartridge is a business need to create, handle and operate on complex data objects that make up the essence of this business. These complex data objects can be made of the new user-defined object types, collections, relationship types, internal large objects, external files etc. Then the functionality upon these complex data objects have to be built, so that the database server runtime environment is extended by user-defined methods, functions or procedures which can be developed using PL/SQL or external languages as C or Java. The new operators are often created for simple querying over the new data object.

There is also a need for new indextypes for fast query evaluation over the user-defined datatypes. These are called *domain indexes*. A domain index is a schema object and it is created, interpreted and accessed by common interface. It consists of routines implemented as methods of a user-defined object type, called an *indextype*. Another part of the data cartridge is created by the *statistics type*, which extends the capabilities of the database optimizer. The extensible optimizer functionality allows creating statistics collections as well as selectivity and cost functions. This information is subsequently used by the optimizer for choosing a query plan. The cost-based optimizer is thus extended to use the user-supplied information, whereas the rule-based optimizer is left unchanged.

All previously introduced components are ordinarily packaged, so the whole data cartridge can be installed as a unit. Once installed, the data cartridge is integrated with the server so that the optimizer, query parser, indexer and other server mechanisms recognize and use all this components.

4.2 Introduction to Oracle Text

Oracle Text is a full-text retrieval data cartridge integrated in the Oracle Standard and Enterprise Database Editions. It extends standard SQL to index, search and analyze text and documents stored in different locations. Oracle Text can perform linguistic analysis on documents and search text using a variety of strategies based on the extended Boolean model, which was introduced in Section 3.1. Oracle Text also makes use of many additional techniques being described in Section 2.2.

4.3 Indexing process

This section gives a brief description of the Oracle Text indexing process which is initiated by the `CREATE INDEX` statement. The diagram of this process is displayed in the Figure 4.1 according to the Oracle Text Reference [10].

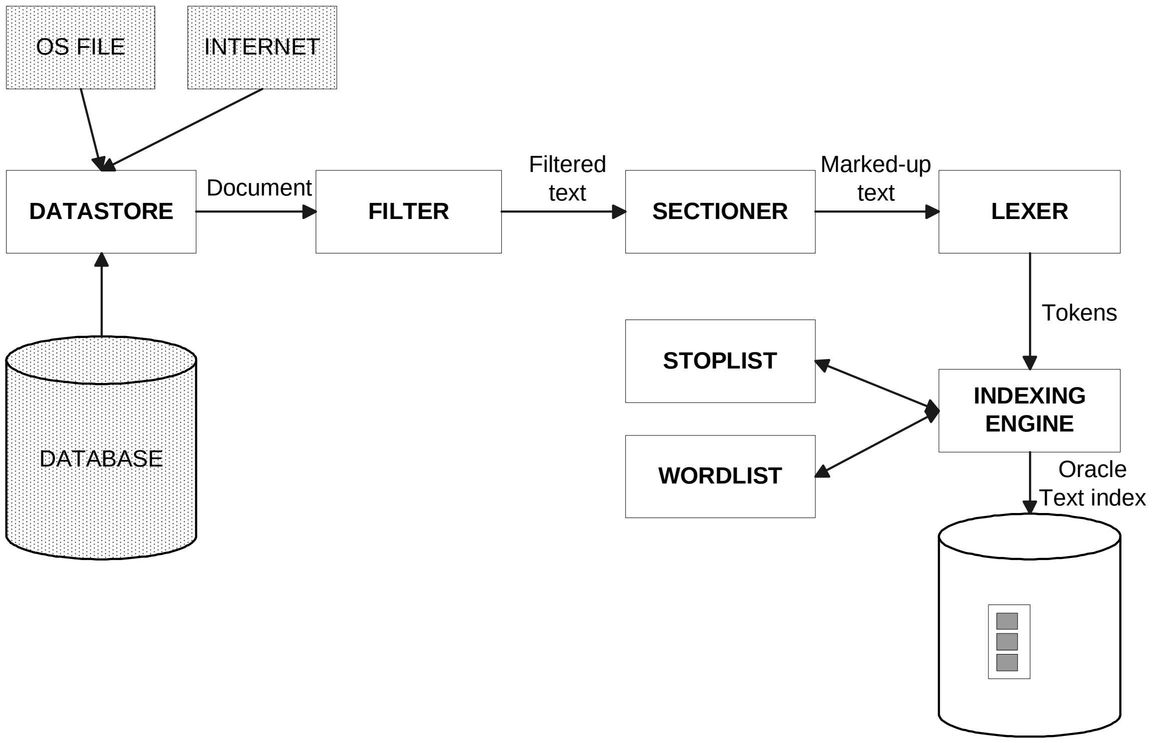

Figure 4.1: Oracle Text indexing process

Datastore

The indexing process starts with reading documents from the datastore they are stored in. This datastore can be presented by one or more columns in database table, operating system file, external file on intranet or internet. This is specified by the `DATASTORE` preference in the `CREATE INDEX` statement.

Filter

The filter that is used on each document allows indexing of formatted documents. They are being filtered into plain texts which are used for indexing afterwards. The filter also enables conversion of a non-database character set to the database character set. The specific filter parameters are specified by the `FILTER` preference in the `CREATE INDEX` statement.

Sectioner

The sectioner is used afterwards to separate documents into individual text sections according to document structure. Thanks to this the queries can be built to search only specific document sections using the `WITHIN` operator. This is used for marked-up texts such as HTML, XML etc. The `SECTION GROUP` preference in the `CREATE INDEX` statement specifies used sectioner parameters.

Lexer

The lexer then processes both tokenization and token normalization. The specific lexer parameters are specified by the `LEXER` preference in the `CREATE INDEX` statement. They include the stemmer specification, definitions for the characters that separate tokens such as punctuation and white space, whether or not to convert the text to all upercase or leave it in mixed case, how to process compounds etc.

Indexing engine

Finally the indexing engine creates the inverted index that maps tokens to the documents that contain them. This engine applies the specified stoplist to exclude stopwords from index and the wordlist with prefix and substring indexing parameters.

Stoplist

The stoplist identifies the words for specific language that are not to be indexed. It is specified by the `STOPLIST` preference in the `CREATE INDEX` statement. Oracle Text provides default stoplists for the most of languages.

Wordlist

The wordlist preference enables the query options such as stemming, fuzzy matching, substring and prefix indexing, which improve wildcard queries. It is specified by the

`WORDLIST` preference in the `CREATE INDEX` statement.

Now let's describe the indexing process again on the very artificial example, which was designed to show the use of all previously described features at once. Let's assume that the documents to be indexed are saved in the directory `C:\docs`, they are holding German texts in XML format and they are saved in separate PDF files. Let's further suppose that the user enters the following statements for the creation of the context index with required preferences. The detailed description of the whole process is described bellow the example:

```
CREATE TABLE german_news (
  id NUMBER,
  xml_texts VARCHAR2(2000)
);

BEGIN
 CTX_DDL.CREATE_PREFERENCE('text_files_dir','FILE_DATASTORE');
 CTX_DDL.SET_ATTRIBUTE('text_files_dir','PATH','C:\docs');

 CTX_DDL.CREATE_SECTION_GROUP('xml_group','XML_SECTION_GROUP');

 CTX_DDL.CREATE_PREFERENCE('german_wordlist','BASIC_WORDLIST');
 CTX_DDL.SET_ATTRIBUTE('german_wordlist','STEMMER','GERMAN');

 CTX_DDL.CREATE_PREFERENCE('german_lexer','BASIC_LEXER');
 CTX_DDL.SET_ATTRIBUTE('german_lexer','COMPOSITE', 'GERMAN');

 CTX_DDL.CREATE_STOPLIST('german_stoplist','BASIC_STOPLIST');
 CTX_DDL.ADD_STOPWORD('german_stoplist','der');
 CTX_DDL.ADD_STOPWORD('german_stoplist','die');
 CTX_DDL.ADD_STOPWORD('german_stoplist','das');
END;

CREATE INDEX ind ON german_news('xml_texts')
  INDEXTYPE IS ctxsys.context
  PARAMETERS('
    DATASTORE text_files_dir
    FILTER ctxsys.inso_filter
    SECTION GROUP xml_group
    WORDLIST german_wordlist
    LEXER german_lexer
    STOPLIST german_stoplist
  ');
```

The process of handling the statement above looks as follows. The document collection for indexing is taken from the directory `C:\docs` specified by the `DATASTORE` preference. Each file stands for one document and each value in the column `xml_texts` of `german_news` table holds the name of one file. After accessing the files, every document is processed by the INSO_FILTER which decodes the text from their binary form in PDF format. Afterwards, the sectioner `xml_group` is used which separates the XML document into the sections. The querying can be then more specific and the user can easily search only in the requested sections of the XML structure. The lexer then processes tokenization which means it breaks the document into individual tokens. The used `german_lexer` defines that the german composite words are indexed as one token and they are not broken into the basic words from which is the composite word concatenated. The processing continues by removing all stopwords defined in the `german_stoplist`. The user here specified his own stoplist preference, which manages removing of only three german articles: *der*, *die* and *das*. And finally, the tokens are prepared for being queried using special wildcard or fuzzy operators and so the special modifications of the resultant inverted index structure are done yet. In the example, the user specified by the `german_wordlist` preference that he wants to use the german stemmer in the queries.

4.4 Extended query features

The Oracle Text is based on the standard Boolean model and thus it accepts queries including the boolean operators `AND`, `OR` and `NOT`. However, it contains a lot of features from the extended Boolean models, which will be briefly introduced here.

The most important is the existence of documents ranking by using a score value. This value is returned by the `CONTAINS` operator and for the goal of sorting the result the `SCORE` operator can be used as well. The calculation of score value for the given document is based on the inverted token frequency principle, but the exact definition of the algorithm is not generally available.

To find more accurate results in cases with frequent misspellings in the document collection, the `FUZZY` operator can be used. It expands queries to include words that are spelled similarly to the specified token. The expanded words have to be contained in the index and there can be specified their maximum number, the lowest similarity score of the token to be included into expansion and if the tokens should be weighted according to their similarity to the given token.

The use of a proximity operator is possible thanks to the `NEAR` operator. It returns a score based on the distance between two or more query tokens in a text, the higher scores for the closer tokens. The duty of keeping the order of specified tokens can be specified there too.

Users can also use thesaurus for extending the query, the Oracle Text interface includes a lot of functions working with the thesaurus and returning for the given word e.g. all synonyms, broader, narrower or related tokens etc. For details about manipulating and using thesaurus see the public package `CTX_THES` in Oracle Text Reference [10].

The special stemmer operator `$` can be used to search for tokens having the same linguistic root as the given query token. The wildcard operators are available in Oracle Text as well. They can be used in query to expand word searches into pattern searches. The `%` wildcard specifies that any characters can appear in multiple positions represented by the wildcard. The _ wildcard specifies a single position in which any character can occur.

In expressions that contain more than one query term, the weight operator `*` can be used to adjust the relative scoring of the query tokens. The score of a query token can be reduced by using the weight operator with a number less than 1, whereas it can be increased by using the weight operator with a number greater than 1 and less than 10. The weight operator is useful in accumulate, disjunctive or conjunctive queries when the expression has more than one query token. With this weighting on individual tokens, the user is able to formulate that one token in query is more important for him then the other one.

4.5 Index design

This section describes the individual Oracle Text indextypes. Furthermore it closer addresses to the context index because only this index type is used for the Vector Text implementation. Its design is thoroughly described as well as the process of its data being saved in the database and processed by DML statements.

4.5.1 Index types

There are three possible types of indexes in the Oracle Text - context, ctxcat and ctxrule.

The *context* index is well-suited for indexing large coherent documents such as plain text, HTML or MS Word documents. Such index can be customized in a variety of ways, but it is not transactional and needs to be synchronized after DML statements manually. The `CONTAINS` operator is used to build queries over context index. More details about context index type processing are described in the following sections.

The *ctxcat* index is designed for indexing small text fragments and related pieces of information, so the other columns of base table can be included in the index to improve a mixed query performance. This index type is transactional, so the synchronization after DML statements is processed automatically and immediatelly by Oracle. The ctxcat index is comprised of sub-indexes defined for the created index set and the index customization variability is much lower then for the context index type. The `CATSEARCH` operator is used for issuing queries over the ctxcat index.

The next example shows the use of ctxcat index type in practice. Let's assume that the user has the following table:

```
CREATE TABLE thesis (
   isbn     NUMBER,
   title    VARCHAR2(200),
   author   VARCHAR2(100),
   abstract VARCHAR2(2000),
   year     NUMBER(4),
   rating   NUMBER
);
```

The user's goal is to search for the thesis saved in this table according to specified content of the abstract and he wants to reduce the result by some constraints on year and rating values. The best solution is the following:

```
BEGIN
  ctx_ddl.create_index_set('thesis_idx_set');
  ctx_ddl.add_index('thesis_idx_set','rating');
  ctx_ddl.add_index('thesis_idx_set','year');
END;

CREATE INDEX thesis_abstract ON thesis (abstract)
  INDEXTYPE IS ctxsys.CTXCAT
  PARAMETERS ('INDEX SET thesis_idx_set');
```

The user creates index set `thesis_idx_set` on `rating` and `year` columns and then he creates the ctxcat index on `abstract` column specifying the `thesis_idx_set` to be used. As the next step, he can enter queries according to his needs mentioned above, e.g. the following query that returns the specification of all thesis which discuss the information retrieval or IRS themes, which were published after the year 2000 and which are retrieved to user sorted by their rating.

```
SELECT isbn, author, title
  FROM thesis
 WHERE CATSEARCH ( abstract,
                  '(information retrieval) | IRS',
                  'year > 2000 order by rating desc') > 0;
```

Finally, the *ctxrule* index is well-suited for document classification or routing. This index is created on the table of queries, where the queries define the classification or routing criteria. The documents can be then classified using the `MATCHES` operator.

Let's introduce an example of such document classification here. The following statements prepare the table of queries, which will control the classification of documents into groups: sport, living and travel documents.

```
CREATE TABLE categ_query (
  query_id   NUMBER,
  category   VARCHAR2(100),
  query_spec VARCHAR2(2000)
);

INSERT INTO categ_query VALUES(1,'Sport','ABOUT(sport)');
INSERT INTO categ_query VALUES(2,'Living','house or flat');
INSERT INTO categ_query VALUES(3,'Travel','ABOUT(travel)');
```

The ctxrule index has to be created for the table of queries to be used in classification.

```
CREATE INDEX ON categ_query(query_spec)
  INDEXTYPE IS ctxsys.CTXRULE;
```

Suppose that there is a table `article` and new records with individual articles are being progressively inserted. The structure of the table can be the following:

```
CREATE TABLE article (
  article_id      NUMBER,
  author          VARCHAR2(30),
  publishing_date DATE,
  article_text    CLOB
);
```

The user would like to classify all articles which will be inserted there. The classification should be saved in the separate table having the following structure:

```
CREATE TABLE article_category (
  article_id NUMBER,
  category   VARCHAR2(100)
);
```

For the purpose of processing the classification a trigger `set_article_category` is created which will decide about the corresponding categories for each inserted record and it will create requested records in the `article_category` table.

```
CREATE TRIGGER set_article_category
  AFTER INSERT ON article FOR EACH ROW
BEGIN
  -- find matching queries
  FOR c1 IN (SELECT category
               FROM categ_query
              WHERE MATCHES(query_spec, :new.article_text)>0)
  LOOP
```

```
    INSERT INTO article_category(article_id, category)
      VALUES (:new.article_id, c1.category);
  END LOOP;
END;
```

4.5.2 Context index tables

The Oracle Text context index includes four tables which are referred to the *$I*, *$K*, *$N* and *$R* tables. These tables are being created in the schema of the index owner and their names are built up by concatenating "DR$", the index name, and corresponding suffix (e.g. "$K"). Figure 4.2 shows the global design of the context index.

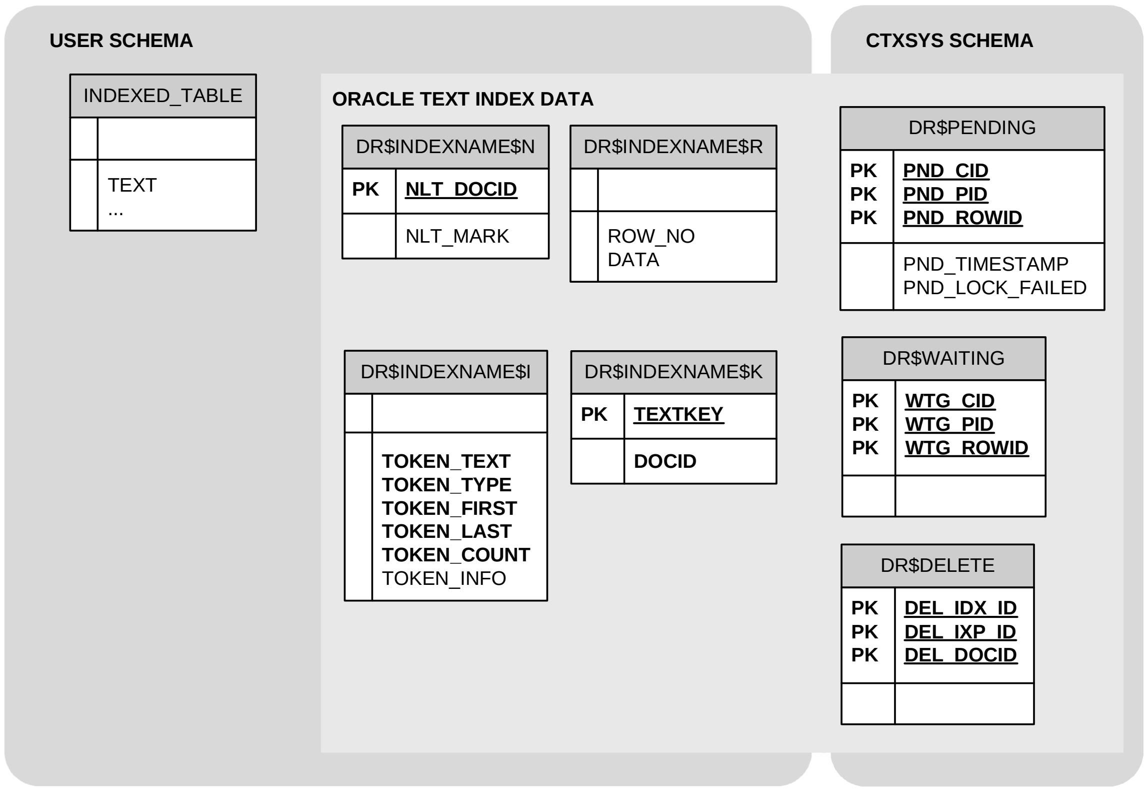

Figure 4.2: Oracle Text context index design

$I The inverted index table

This table holds the inverted index data. That means it stores all the indexed tokens together with a binary structure which holds all documents the given token occurs in as well as the exact positions of occurrences within those documents. Each document in this structure is represented by an internal docid value.

$K The docid mapping table

This table maps the internal docid value to the external rowid value for each indexed document. A single docid/rowid pair can be found for each row of the table being indexed. This table is used for fetching a docid when the rowid value is known.

$R The rowid mapping table

This table is designed for the opposite mapping against the *$K* table. The rowid value is being fetched when the docid value is known. Because of the facts that rowids have a fixed length and docids are being allocated sequentially, it is possible to save all rowids in a binary structure and to get any specific rowid by reading the fixed length of data at specific position from the very start. To prevent a single row from getting too large, this binary structure is split over several rows in the *$R* table in practice.

$N Negative row table

This table contains a list of docid values for those documents deleted from the indexed table. These data are used and cleaned up by the index optimization process.

4.5.3 Global context index tables

These tables exist within the `CTXSYS` schema and they are used for all context indexes of Oracle Text. That is why the identification of appropriate index has to be contained in each saved record.

DR$PENDING

This table contains rowids of rows which were newly inserted into the indexed table and have not yet been processed into *$I* table by index synchronization. The timestamp of insert operation is present there as well.

DR$WAITING

The rowids of rows, which were at first inserted and then updated in the indexed table, are saved here. They have not yet been processed into *$I* table by index synchronization in this case either.

DR$DELETE

This table contains rowids of rows, which were deleted from the indexed table and the commit has not been processed yet.

4.5.4 DML processing

Inserts

When a new record is inserted into a table with an Oracle Text context index, the appropriate index creation routine is called. The new row containing the rowid of the new record is created in the `DR$PENDING` by this routine and no other operations are being executed at this time. This means that the indexes are not updated to reflect the new record. This is done later by the means of the index synchronization process. The row inserted into `DR$PENDING` is in the same commit unit as the new record being inserted into the base table, thus they will both be either rolled back or committed together.

Deletes

When an indexed row is deleted, the following three operations are processed: the corresponding row from the *$K* table is deleted too, a row containing id of index and docid is inserted into the `DR$DELETE` table, and a row containing rowid is inserted into the *$N* table. When the user commits his delete, all these events are committed too. Removing the row from *$K* means that functional lookups in the index will not lookup the deleted row. Adding a row into `DR$DELETE` means that normal index lookups will not find the deleted row and the commit callback can process the delete of row from the *$R* table. The row added into the *$N* table will be used during the index optimization to remove the docids which are not needed from the *$I* table.

If the row is inserted and deleted again in the same commit unit, then there will be no row in the *$K* at the start of this process and no special action needs to be taken.

Updates

An update of the indexed table is basically treated as a delete followed by an insert. The record is deleted according to the previous section and then the rowid for the record is inserted into the `DR$PENDING` table. An extra step needs to be taken when a given row in the `DR$PENDING` table already exists. This means an early insert of the row given is not synchronized yet. In this case the row is inserted into the `DR$WAITING` table. The reasoning behind this functionality is the fact that the row in the `DR$PENDING` table may already being processed by an index synchronization. If this is true the indexed data have to be resynchronized later again.

Commit callback

The commit callback is invoked at a commit time with an internal id for the index to be updated. This callback fetches all docids for the given index id from the `DR$DELETE` table . For each fetched docid an update of the LOB in *$R* table is performed where the corresponding rowid strings are set to nulls. Finally it deletes all the rows of the index given from the `DR$DELETE` table.

Query time

There are two types of index lookup in the Oracle Text - a normal and a functional lookup. The normal lookup returns all the rowids that satisfy the query criteria, while the functional lookup decides whether the given row satisfies the query criteria.

The normal lookup fetches at first a set of docids from the *$I* table and then it converts them to rowid values using the *$R* table. If a record has been deleted in the current session but not committed yet then the *$R* table will not have been modified yet. That is why during a normal lookup the `DR$DELETE` table must be checked and all unneeded docid values found in this table must be removed before converting fetched values to rowids using the *$R* table. This control applies only to records modified in user's own session - other session's uncommited modifications are invisible to the current user anyway. And once they are committed, the *$R* table will have the old docids nulled out.

In the case of a functional lookup there is no need for any special processing because the functional lookup reads from the *$K* table and this table is updated immediately after the record is changed.

Index synchronization

The index synchronization occurs when the user executes the SQL statement `ALTER INDEX indexname REBUILD ONLINE PARAMETERS ('sync')` or calls a PL/SQL routine `CTX_DDL.sync_index`. The synchronization process loads all changes done by insert and update operations on the indexed table into the index structure. It looks at first into `DR$PENDING` and `DR$WAITING` tables for rowids of records to be updated. A new internal docid value is assigned for each fetched rowid. Then the document data are indexed via the indexing pipeline and all the resultant token, docid, and word position data are inserted into the *$I* table. A new row in the *$K* table holding the docid/rowid pair is created and the structure in the *$R* table is extended by the new rowid string on the appropriate possition.

Index optimization

Index optimization occurs when a user executes the SQL statement `ALTER INDEX indexname REBUILD ONLINE PARAMETERS ('optimize')` or calls a PL/SQL routine `CTX_DDL.optimize_index`. The process of optimization performes two actions; it removes old data and minimizes the index fragmentation. The data in the *$I* table are changed, all deleted docids are removed from inverted index structure and the different rows holding the data for the same token are concatenated into one row.

Implications

The delete operation results in an immediate change of index. That means the user's session will no longer find anything for the deleted record from the moment the change has been made. Other users will not find it as soon as the commit has been processed.

The insert and update operations work differently, meaning that the new information will not be visible to text searches until an index synchronization has occurred. The most important effect of this behaviour becomes evident on the updates: when the user makes a small alteration to a document, it becomes effectively invisible for all searches until an index synchronization occurs!

When a record has been deleted, there is no way to process the functional lookup because of using the *$K* table.

4.6 Privileges

While any user can create an Oracle Text index and issue a query using `CONTAINS` operator, Oracle Text provides the `CTXSYS` user for administration and the `CTXAPP` role for application developers.

CTXSYS user

The `CTXSYS` user is created at install time, Oracle Text users are administered by this user.

`CTXSYS` can do the following:

- Modify system-defined preferences
- Drop and modify other user preferences
- Call procedures in the `CTX_ADM` PL/SQL package to set system parameters
- Query all system-defined views
- Perform all the tasks of a user with the `CTXAPP` role

CTXAPP role

The `CTXAPP` role is a system-defined role that enables users to do the following:

- Create and delete Oracle Text preferences
- Use the public Oracle Text PL/SQL packages

4.7 Possible improvements

The Oracle Text has been developed in many years. That is the reason why the scope of this thesis does not allow any bigger improvements based on the Boolean model potential, morover for a person standing out of the Oracle corporation. Therefore all further proposed improvements result from the planned implementation of the Vector space model.

The use of the Vector space model brings along the possibility of experiencing several types of calculation similarity measure and token weight values. In some specific situations it can be more convenient to use the specific type of similarity measure or token weight, because of the either resultant values or computational effectiveness. For very special situations the possibility of designing an own special similarity measure or token weight function that would satisfy the desired conditions can be useful. The resultant score values in the Vector space model are also more transparent because the exact algorithm of calculating the score values in Oracle Text is not generally known.

The Oracle Text also requires the queries being made up of tokens connected by logical conjunctions, which can be confusing to some users. These users would appreciate the possibility of entering queries using just the natural language. Another comfortable feature would be an option of specifying the document to form a query, which search for the other similar documents in the database.

Chapter 5

Vector Text implementation

5.1 Fundamental design decisions

5.1.1 Form of extending the Oracle Text

The form of extending the Oracle Text was the most fundamental design decision and had to be created first. The main question was how to implement the Vector Text index. There were three main options: to implement all starting from document's parsing and tokenization, to re-use the functions implemented inside the Oracle Text or to build the Vector Text index based on the Oracle Text index data.

Own implementation

The first way was to implement all needed functionality from the beginning. It would imply a lot of extra work for implementing every step of the process of creating Vector Text index, e.g. document's parsing, stemming, filtering, enabling use of different data locations etc. This principle would enable the existence of Vector Text cartridge even on Oracle database where the Oracle Text is not installed. Such system could also have a better performance than the system creating the vector index based on the Oracle Text index. The immoderate additional amount of work would have been done anyway. This is, however, definitelly not the goal of the work. That is why the own implementation of document's parsing and tokenization was taken as the worst possibility and would be executed only when there was no other option left.

Use of internal functions from the Oracle Text

The opposite approach for implementing the required functionality was to use existing internal functions from the Oracle Text. It would probably take the least effort and could run as fast as in the Oracle Text, that is why this way was the very much preferred one. Unfortunately all this functionality of Oracle Text is encapsulated inside the wrapped PL/SQL source code and dynamic libraries and it is not available from outside. For this reason this approach could not be used for implementing Vector Text cartridge.

Build index based on Oracle Text index

The last way was to create the Oracle Text context index first and then use its data for creating the Vector Text index. The advantage of this approach is the possibility to use both the context and the vector index at the same time. The Boolean queries can be in this case processed on the table where the vector index is built. This feature can be used for the direct comparison of the application and performance of both cartridges. The obvious disadvantage is the length of the duration of all DDL operations on the vector index because changes has to be processed at first in the context index data and then in the vector index data.

The primary goal here is to take an advantage of the context index data. The first idea, how to get the inverted index data, was to use the existing Oracle Text procedure `ctx_report.token_info` which parses inverted index data for the given token and creates the report with results. Unfortunately using this procedure during the vector index creation was extremely time-consuming and it did not work for common tokens in larger collections of documents either. That is why the only other way was to implement the own decoding of the binary structure of inverted index data, which is fast enough to be used.

5.1.2 Index structure in Vector Text

The basic architecture of the Vector Text index structure had to be designed before start of implementation phase.

The way of processing the Boolean queries is always the same. The Boolean query searches for all documents, which include the token specified in the query. That is why the index can have a structure of one record for one token. The list of documents with occurrences of the given token is thus coded in the binary structure and saved for each token record. However, such index structure is unusable in Vector space model, because this model needs both search directions. It needs to search for documents, which include the specified tokens and it also needs to find all tokens occurring in the specified document.

Two basic designs of the index structure in Vector Text were considered. The first design proposes to create two separate index structures. Each of them would be used for one of the search directions - all documents for a specified token and all tokens for a given document. The second design suggests having only one index structure. One record would be created in the index table for the combination of each token and the document where the given token occurs.

The advantage of creating of two separate index structures is less space requirement for saving index data. This benefit would be enhanced by coding the data into a binary structure similarly to the Oracle Text index principle. By contrast, having only one index structure would result in much faster lookups. The final decision preferred more the faster querying than the less space requirements and that is why Vector Text uses only one main index structure.

5.1.3 Vector Text user interface

The Vector Text cartridge was assigned as the extension of the Oracle Text and the obvious decision was made to make the user interface as similar to Oracle Text as possible. The same naming convention was used for the implemented global and internal methods, tables, structures and scripts. A lot of functionality that is used in Oracle Text such as index preference handling, packages for index reporting or logging was incorporated also into the Vector Text cartridge. There is also the possibility of using some functionality from the Oracle Text, such as specifying thesaurus functions in the queries or defining own stoplists. That all enables the easy understanding and the same style of work to the users being used to working with the Oracle Text. The resemblence between both interfaces for Oracle Text and for Vector Text is provided by the Table 5.1.

Object type	Oracle Text	Vector Text
SCHEMA	CTXSYS	VECTXSYS
ROLE	CTXAPP	VECTXAPP
PACKAGE	CTX_ADM	VECTX_ADM
PACKAGE	CTX_DDL	VECTX_DDL
PACKAGE	CTX_OUTPUT	VECTX_OUTPUT
PACKAGE	CTX_REPORT	VECTX_REPORT
OPERATOR	CONTAINS	CONTAINS
OPERATOR	SCORE	SCORE
INDEXTYPE	CONTEXT	VECTOR

Table 5.1: Similarities in using Oracle Text and Vector Text

5.1.4 Language for writing of methods

The choice of the programming language to be used for implementation of methods had to be made too. There were two main variants - PL/SQL or 3GL represented by the C language. The 3GL should be generally used in Oracle database when it is impractical or impossible to code the required functionality in SQL (PL/SQL). The SQL calls to the database are used in the Vector Text statements implementation all the time and for such work the SQL and PL/SQL languages are the most suitable ones. That is why the use of 3GL was consulted only once for the task of decoding inverted index data of context index. The parsing of binary data and related in-memory operations are much faster when implemented in C language than coded in PL/SQL. The speed of C language parsing outweighs the extproc callout overhead when all DML operations are implemented using arrays and bulk operations. Finally, the decision was made to code the functionality of parsing context index data along with inserting data into vector index tables in C language.

The best PL/SQL practices [4] should be used during the implementation to achieve the best performance of the system for its designed architecture. To be more specific, all SQL statements should be created using bind variables instead of constant values because bind variables minimize the very time-consuming repeated parsing. For the massive multirow operations a bulk processing for reducing traffic and permanent communication between client and database server should be used. The dominant use of static SQL should be considered because it is more scalable, easier to debug and maintain and it runs faster than the same code using dynamic SQL.

But not all these methods could be complied during the implementation because the design of Vector Text cartridge forced the frequent use of dynamic SQL. All data of given index are saved in the index specific tables and thus all general methods, querying the tables which name is not known in the time of the code compilation, have to use dynamic SQL.

5.1.5 Vector Text design complexity

Primary utilization of the Vector Text cartridge was drawn up for academic purpose and studies, that is why the amount of produced similarity measures and token weights and the simple-used interface for their further addition was preferred to the tuning and optimization of the most powerful implementation of such similarity measure. The index creation and manipulation performance had already been affected by the decision to use the context index from Oracle Text as the fundament on which the Vector Text index is built up. That is why the performance of Vector Text index manipulation had to be always lower then the Oracle Text index manipulation.

5.2 Indexing process

This section describes the Vector Text indexing process and the way it extends the Oracle Text indexing process. For the illustration see the diagram in the Figure 5.1.

The Vector Text indexing process creates first the Oracle Text context index where all the document content processings like text filtering, tokenization, stop words elimination etc. are included. Then it uses context index data to prepare additional data for the needs of the Vector Text index.

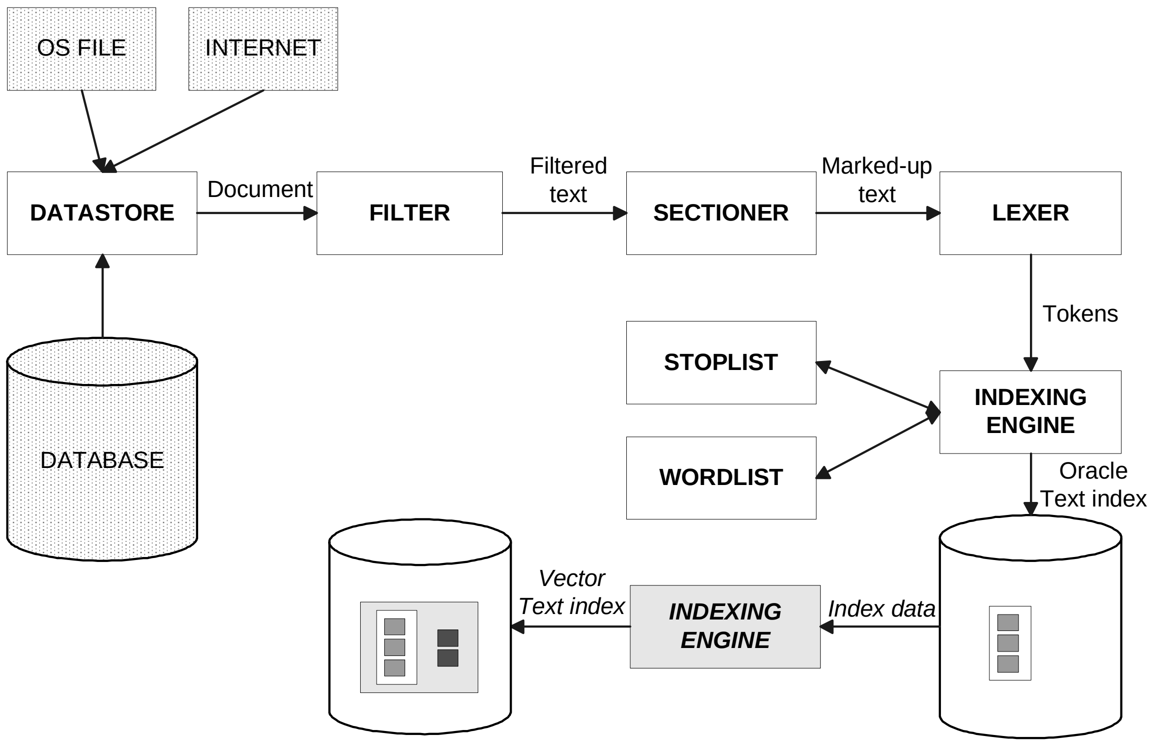

Figure 5.1: Vector Text indexing process

Indexing engine

Vector Text indexing engine fetches data from Oracle Text inverted index in binary form. Then it decodes inverted index data structure and extracts the mapping of individual tokens to documents they are contained in. As the next step token frequency values for individual documents are counted and finally these data are saved into database table for the Vector Text queries processing.

5.3 Index design

This section describes the Vector Text index design, the way the Vector Text index data uses and extends Oracle Text index and how DML statements are being processed. The schema of Vector Text index design shows Figure 5.2.

5.3.1 Vector index tables and triggers

The Vector Text index design extends the Oracle Text index design by the *$TI* and *$TF* tables being created within the schema of the index owner. Two new triggers referred as *$AT* and *$BT* are being created on Oracle Text *$I* table. All

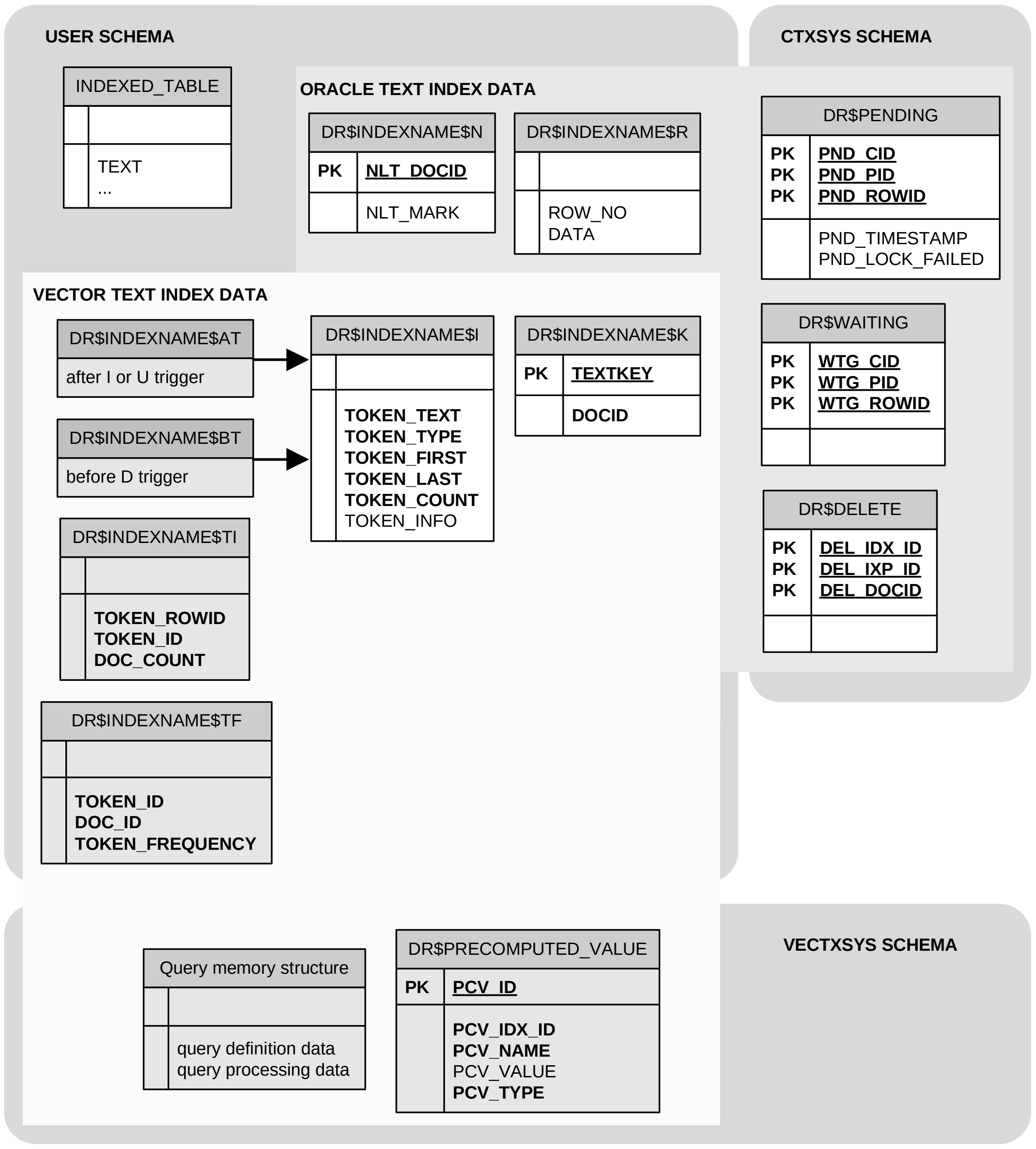

Figure 5.2: Vector Text index design

these new objects have names concatenated from "DR$", the name of the index, and appropriate suffix (e.g. "$TF"). Besides that one global index table named `DR$PRECOMPUTED_VALUES` is added in the `VECTXSYS` schema. The *$I* and *$K* tables from the Oracle Text are used directly by Vector Text cartridge for the purpose of query processing and vector index data creating.

$TI The token identification table

This table maps token records from *$I* table (represented by rowid value) to the internal token ids. Each record in this table holds also the value of document count where the given token occurs. The *$TI* is created as index organized table for the better performance during queries.

$TF The token frequency table

This table consists of all the tokens that have been indexed, together with the internal docids and token frequency values for each document. Each token in this table is represented by the internal id which is mapped through the *$TI* table to the individual token record from the *$I* table.

$AT After insert or update trigger

This trigger identifies inserted/updated rows from the *$I* table and prepares their rowids into the collection in memory. This collection is consequently used by index synchronization or optimization for processing changes into the *$TI* and *$TF* tables.

$BT Before delete trigger

This trigger identifies the rows deleted from the *$I* table and prepares their rowids into the collection in memory. This collection is consequently used by index optimization to process these changes into the *$TI* and *$TF* tables.

DR$PRECOMPUTED_VALUES table

This table holds for given index values that can be repeatedly used during the query processing, similarity measure values calculating etc.

5.3.2 Query memory structure

Two main structures are used as shown in the Figure 5.3 - the first one stores the data for all queried documents being created during query processing and the second one holds the data for the query definition and its tokens.

Opened cursors

The list of opened cursors for given query is saved here. All opened cursors are closed after processing the query.

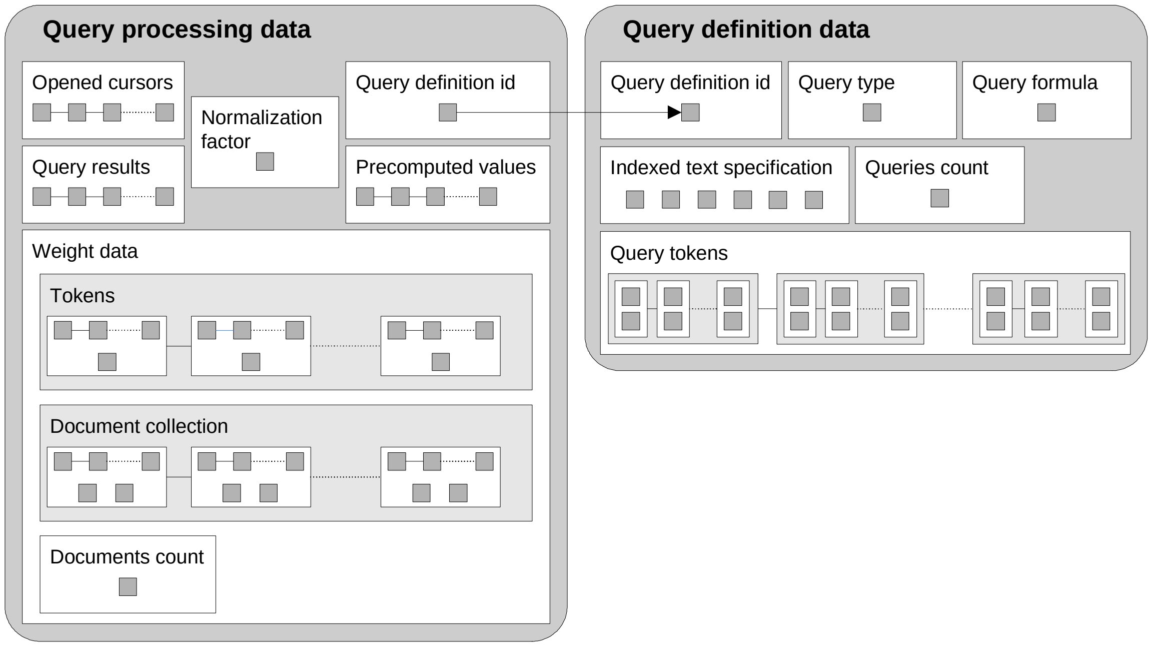

Figure 5.3: Query memory structure

Query results

The list of queried documents with their current score value and normalization factor for query is stored here.

Weight data

Holds the weight data for all queried documents and its tokens. It consists of the following three main parts:

- List of all tokens which take place in queried documents, for each token there are saved the following data:
  - List of documents where the given token is included with its frequency for the particular document
  - Count of documents where the given token is included
- Collection of documents being queried, for each document the following data are saved:
  - List of included tokens with their frequencies for given document
  - Count of all tokens in given document
  - Maximal token frequency in given document

- Count of all documents in collection being queried

Normalization factor

Normalization factor for given query is stored here. It is being used by some similarity measure procedures with the aim to normalize the resultant values.

Query definition id

The identification of the query definition structure, which is used for actual query is saved here.

Precomputed values

Holds the list of precomputed values, which can be used only for the needs of the actual query.

Query type

Identification of the query type is saved here. It holds the information about the operator and the parameters used for creating the query.

Query formula

The query formula is stored here according to the operator used for the query creation. It can be a formula of text vector or the text which has allready been indexed or not.

Indexed text specification

The data about being used only with queries created by `INDEXED_TEXT` operator are to be found here. This data provides the user with the information about indexed table, name of vector and context indexes etc.

Queries count

Number of queries using current query definition is stored here.

Query tokens

Holds the list of tokens from query definition in uppercase, for each token the following data are saved:

- Token frequency and weight for query
- List of tokens in mixedcase which, are equivalent to given token in uppercase. For each mixedcase token there is saved its frequency and weight.

5.3.3 DML processing

Inserts, deletes, updates, commits

These DML statements are processed in the same way described for the Oracle Text context index and they have the same implications. Because of using *$K* table for vector index lookups all deletes are reflected by index immediately. In comparison with that insert and update operations are not considered by index until its synchronization is done.

Query time

As for the Oracle Text index there are two sorts of index lookup used in Vector Text - normal and functional lookups. Document rowids from *$K* table are used for the both types of lookup. In this way the deleted documents are not queried immediatelly after being deleted.

The query processing starts with parsing and creating query definition using presented operator - `TEXT_VECTOR`, `INDEXED_TEXT` or `NONINDEXED_TEXT`. This operator fills up the query definition structure with appropriate data, allocates the query data structure and returns its identificator. If the functional lookup takes place then the presented operator is called separately for each row. Thats why the query formula is saved and when such a formula allready exists in the memory then the query parsing does not need to be processed again.

The `CONTAINS` operator is processed after creating the query definition. This operator chooses the suitable similarity measure procedure from the index preferences as well as it selects a token weight function which should be used for the query. Then it calls the chosen similarity measure procedure for given document or all queried documents according to the query lookup type.

All similarity measure procedures can be processed in different ways according to their definitions. The following example is presented to demonstrate some of those ways. Let's suppose the following query is being processed

```
SELECT SCORE(1),
       id
  FROM news
 WHERE CONTAINS(text,
                TEXT_VECTOR('oracle:1;database:0.5'),
                'similarity_measure cosine_measure
                 token_weight tf_weight',
                1) > 0.5;
```

and there is the Vector Text index called `news_ind` on the `news` table. The presented similarity measure is then processed the following way:

1. Get the actual documents collection using the query

```
SELECT docid
  FROM DR$NEWS_IND$K;
```

2. Go through all tokens in query, which are being saved in the `query tokens` structure

3. For each such token calculate

   (a) its weight in the query: q_j

   (b) its weights for all documents in queried collection:

   $\forall i \in (1, \ldots, n) : w_{i,j}$

```
SELECT SUM(tf.token_frequency)
  FROM DR$NEWS_IND$I i,
       DR$NEWS_IND$TI ti,
       DR$NEWS_IND$TF tf
 WHERE i.token_text = query_tokens(index)
   AND i.rowid = ti.token_rowid
   AND ti.token_id = tf.token_id
   AND tf.doc_id = document_id;
```

4. Calculate unnormalized score values for all documents and saves them into `query results` structure, one value for each document:

   $unnormal\_score(D_i) = \sum_{j=1}^{n} q_j * w_{i,j}$

5. Normalize resultant values of similarities for each document - cosin normalize of token frequency

   (a) get squared weights of all documents:

   $square\_weight(D_i) = \sum_{j=1}^{n} (w_{i,j})^2$

```
SELECT tf.doc_id,
       SUM(tf.token_frequency)
  FROM DR$NEWS_IND$TF tf
 GROUP BY tf.doc_id;
```

   (b) get squared weight of query, more precisely get sum of all squared token frequencies for all tokens being saved in the `query tokens` structure:

   $square\_query = \sum_{j=1}^{n} (q_j)^2$

(c) count the total normalized score for each document and saves normalized values back into `query results` structure:

$$normal\_score(D_i) = \frac{unnormal\_score(D_i)}{\sqrt{square\_weight(D_i)} * \sqrt{square\_query}}$$

The normalized score values from the `query results` structure are returned by the `CONTAINS` operator for each document. Such value can be also returned by the `SCORE` operator for sorting the query results. Only the rows fulfilling the condition for the `CONTAINS` operator are returned to the user. Finally after returning query result the query data and query definition structures are cleaned up.

More details about previously used operators including detailed examples can be found below in the Appendix A, subsections A.2.2 - A.2.6.

Index synchronization

The index synchronization occurs when the user executes the SQL statement `ALTER INDEX indexname REBUILD ONLINE PARAMETERS ('sync')` or calls a PL/SQL routine `VECTX_DDL.sync_index`. At first the synchronization of Oracle Text index is processed which among others inserts new rows into the *$I* table. Rowids of all these inserted rows are saved in a collection in the memory thanks to *$AT* trigger on the *$I* table. The collection is consequently processed by Vector Text index synchronization which decodes inverted index data from *$I* table for all rows included in the collection. Then new token ids are insertd into the *$TI* table. The token frequences for particular documents are further counted and finally saved into the *$TF* table.

Index optimization

The index optimization takes place when the user executes the SQL statement `ALTER INDEX indexname REBUILD ONLINE PARAMETERS ('optimize')` or calls a PL/SQL routine `VECTX_DDL.optimize_index`. To begin with, the optimization of Oracle Text index is being processed. During this process all the old token rows are deleted from *$I* table and more rows for one token are concatenated in the *$I* table. All such deleted and updated rows are saved in a collection in the memory thanks to *$AT* and *$BT* triggers on the *$I* table. This collection is consequently being processed by Vector Text index optimization. Rows are deleted from the *$TI* and *$TF* tables for all corresponding rows previously deleted from the *$I* table. For all previously updated rows inverted index data are being decoded from the *$I* table and counted token frequencies for particular documents. These token frequencies are finally saved into the *$TF* table.

Implications

The implications are the same as for the Oracle Text index - deletes are taken into consideration immediatelly while inserts and updates are not proceeded until index synchronization is done.

5.4 Privileges

While any user can create an Vector Text index and issue a query using `CONTAINS` operator, Vector Text provides the `VECTXSYS` user for administration and the `VECTXAPP` role for application developers.

VECTXSYS user

The `VECTXSYS` user is created at install time, Vector Text users are administered by this user.

`VECTXSYS` can do the following:

- Modify system-defined preferences
- Drop and modify other user preferences
- Call procedures in the `VECTX_ADM` PL/SQL package to set system parameters.
- Query all system-defined views
- Perform all the tasks of a user with the `VECTXAPP` and `CTXAPP` roles
- Implement new similarity measure procedures and token weight functions

VECTXAPP role

The `VECTXAPP` role is a system-defined role that enables users to do the following:

- Create and delete Vector Text preferences
- Use the Vector Text PL/SQL packages
- Perform all the tasks of a user with the `CTXAPP` role in Oracle Text

5.5 Similarity measure functions

This chapter contains the list of similarity measures implemented in the Vector Text cartridge and a guide for implementing user's own similarity measure. The similarity measures are briefly described along with some performance assumptions. The principle of saving some important precomputed values is used by a lot of implemented functions. This results into the efficiency of individual methods, the first query of given type takes longer then the subsequent ones. There has to be calculated all values during the first query, but the other queries can leave out some calculations and use precomputed values instead. The precomputed values are deleted either during the process of synchronization and optimization, or together with dropping or truncating the whole index. The notation in individual formulas is the same as in the Chapter 3 - Q stands for the query, D_i means the i-th document in the collection, q_j represents the j-th element of the query vector and $w_{i,j}$ stands for the j-th element of document vector for the i-th document.

Scalar measure

The Scalar measure is the basic similarity measure implemented in the Vector Text cartridge, its definition is following:

$$sim(Q, D_i) = \sum_{j=1}^{n} q_j * w_{i,j}$$

This formula computes the length of the projection of the document vector onto the query vector. The resultant document score is directly proportional to the length of the document vector. This measure can be used by the `SCALAR_MEASURE` preference.

Cosine measure

The Cosine measure is the Scalar measure being normalized and the result represents the cosine of the angle between the query and document vectors. The definition of Cosine measure is following:

$$sim(Q, D_i) = \frac{\sum_{j=1}^{n} q_j * w_{i,j}}{\sqrt{\sum_{j=1}^{n} q_j^2} * \sqrt{\sum_{j=1}^{n} w_{i,j}^2}}$$

The maximum resultant similarity value is one, which means that two identical vectors have a zero angle between them. The minimum possible similarity value is zero, which means two vectors having nothing in common, the angle between them being 90 degrees. The denominator in this equation discards the effect of the lengths of the documents on their scores. Unfortunately, the computation of the normalization factor is extremely expensive because it requires an access to every token in the document and not just the tokens specified in the query. This measure can be used by the `COSINE_MEASURE` preference.

Approximated cosine measure

The Approximated cosine measure executes the normalization of Scalar measure too but only by number of tokens in given document. See the definition:

$$sim(Q, D_i) = \frac{\sum_{j=1}^{n} q_j * w_{i,j}}{\sqrt{number\,of\,tokens\,in\,D_i}}$$

The effect of the normalization is not as strong as for the Cosine measure, but the computational requirements are much lower and so this measure works better in IRS implementations. This measure can be used by the `APPROXIMATED_COSINE_MEASURE` preference.

Jaccard measure

The Jaccard measure is defined the following way:

$$sim(Q, D_i) = \frac{\sum_{j=1}^{n} q_j * w_{i,j}}{\sum_{j=1}^{n} q_j^2 + \sum_{j=1}^{n} w_{i,j}^2 - \sum_{j=1}^{n} q_j * w_{i,j}}$$

Briefly speaking, it is the weight of all tokens included in both query and document divided by the weight of all tokens included in query or document. This implies the fact that the shorter document will result with a higher weight than the same document, which is only enlarged by the text including tokens that are not queried. The shorter documents are more descriptive and therefore preferred by this measure. The computational requirements needed are similar to those used for the Cosine measure. The Jaccard measure can be used by the `JACCARD_MEASURE` preference.

Dice measure

The Dice measure is very similar to the Jaccard measure and it is defined as follows:

$$sim(Q, D_i) = \frac{2 * \sum_{j=1}^{n} q_j * w_{i,j}}{\sum_{j=1}^{n} q_j^2 + \sum_{j=1}^{n} w_{i,j}^2}$$

The same principle is valid for the resultant weights according to document length and descriptiveness as for the Jaccard measure. This measure can be used by the `DICE_MEASURE` preference.

Overlap measure

The definition of Overlap measure is following:

$$sim(Q, D_i) = \frac{\sum_{j=1}^{n} q_j * w_{i,j}}{\sum_{j=1}^{n} \min(q_j^2, w_{i,j}^2)}$$

The computation of the normalization factor is not as expensive as the one for the Cosine measure, since only the tokens from the query need to be accessed thanks to using the minimum function. This measure can be used by the `OVERLAP_MEASURE` preference.

Asymmetric measure

The Assymmetric measure is defined as follows:

$$sim(Q, D_i) = \frac{\sum_{j=1}^{n} \min(q_j, w_{i,j})}{\sum_{j=1}^{n} w_{i,j}^2}$$

The needed computational requirements are similar to those of the Cosine measure because the access to every token from the document is required in both cases. This measure can be used by the `ASYMMETRIC_MEASURE` preference.

Pseudocosine measure

The Pseudocosine measure is very similar to Cosine measure and is defined the following way:

$$sim(Q, D_i) = \frac{\sum_{j=1}^{n} q_j * w_{i,j}}{\left(\sum_{j=1}^{n} q_j^2\right) * \left(\sum_{j=1}^{n} w_{i,j}^2\right)}$$

This measure can be used by the `PSEUDOCOSINE_MEASURE` preference.

5.5.1 Implementing user's own measure procedure

The Vector Text cartridge was designed to enable implementing and using own similarity measures easily. The three main steps have to be completed for user's own similarity measure use. First, the procedure has to be implemented, second, the new preference using given procedure has to be created and last but not least, the index with this new preference has to be created.

Measure procedure implementation

User's own similarity measure procedure can be developed only under the `VECTXSYS` schema. This procedure has to have the following declaration structure:

```
procedure user_measure_procedure(
       index_info in vectxsys.VectorIndexInfo
     , query_id in pls_integer
     , doc_id in pls_integer);
```

The first parameter of `VectorIndexInfo` type contains all necessary information about index against which the query is being processed. The second parameter identifies the processed query and the third parameter identifies the document for which the similarity measure should be counted. The following figure shows the skelet of every similarity in the measure procedure:

```
loop
  --get next token from query
  query_token := vectxsys.driqry.fetch_token(...);

  --set weight_info object with actual attributes
  weight_info := TokenWeightInfo(...);

  --run appropriate token weight procedure
  -- on query token and get its weight
  ...

  --insert query token with its weight into memory structure
  vectxsys.driqry.insert_token(...);

  if doc_id not null then
    --set weight_info object with actual document and token
    weight_info := TokenWeightInfo(...);

    --run token weight function and get document weight
    ...

    --save document weight into memory structure
    vectxsys.driqry.increase_doc_score(...);
  else
    loop through all documents
      --set weight_info object with actual document and token
      weight_info := TokenWeightInfo(...);

      --run token weight function and get document weight
      ...

      --save document weight into memory structure
      vectxsys.driqry.increase_doc_score(...);
    end loop;
  end if;

end loop;

--run appropriate normalizing procedure
...
```

This skeleton forms the basis of implemented similarity measure, more useful functions are prepared for usage in Vector Text drifun package (eg. `get_document_weight` function for calculating document weight using many different algorithms). The simplest way of the implementing user's similarity measure procedure is to call `drifun.user_measure_prototype` procedure inside. This procedure is implemented according to previously shown skeleton and it can process many types of calculation according to chosen input parameters. The details of this procedure can be found in the technical documentation. The following example creates a procedure to implement Asymmetric similarity measure using described prototype:

```
create procedure user_asymmetric_measure(
   index_info vectxsys.VectorIndexInfo,
   query_id pls_integer,
   doc_id pls_integer
)
is
begin
   drifun.generic_measure_prototype(
      index_info,
      query_id,
      doc_id,
      'ASYMMETRIC_MEASURE',
      'MIN',
   );
end user_asymmetric_measure;
```

For the purpose of creating own similarity measure the implementation of own normalizing procedure can be necessary. Such a procedure process normalization of calculated similarity measure and implementation of the procedure outside of the similarity measure procedure is appropriate for enabling the use of the same normalization principle in more similarity functions. This procedure can be created only in `VECTXSYS` schema and it has to have the following declaration structure:

```
procedure user_normalize_procedure(
      index_info vectxsys.VectorIndexInfo
    , query_id number);
```

The first parameter of VectorIndexInfo type contains all the necessary information about the index against which the query is being processed and the second parameter identifies the processed query. The following figure shows the skelet of every normalization procedure for similarity measure:

```
loop
  --get score value for the next document
  score_value := driqry.get_next_doc_score(...);
  --exit when all document score values normalized
  exit when doc_id is null;

  --normalize actual document score value
  ...

  --save normalized score value into memory structure
  driqry.insert_doc_score(query_id, doc_id, score_value);
end loop;
```

Preference creation

The new preference for the implemented measure has to be created. Then all the attributes used by implemented procedure have to be set. The only compulsory attribute that has to be set is named `PROCEDURE` and its value means implemented measure procedure name. All other optionally attributes used by the particular similarity measure procedure have to be set too.

The following example creates the preference `my_new_measure` which uses the procedure named `my_new_measure` from the `user_procs` package for calculating similarity measure and the function named `cosin_measure` from the `drifun` package is taken for normalization processing in the given measure.

```
begin
  VECTX_DDL.CREATE_PREFERENCE(
     'MY_NEW_MEASURE',
     'USER_MEASURE');
  VECTX_DDL.SET_ATTRIBUTE(
     'MY_NEW_MEASURE',
     'PROCEDURE',
     'user_procs.my_new_measure');
  VECTX_DDL.SET_ATTRIBUTE(
     'MY_NEW_MEASURE',
     'NORMALIZE_PROCEDURE',
     'drifun.cosine_normalize');
end;
```

New measure usage

The newly implemented measure can be used as a default for all queries on the created index.

The following statement creates the vector index called `news_ind` on `text` column in `news` table and `my_new_measure` preference is used as the default similarity measure for all queries on given index.

```
CREATE INDEX news_ind ON news(text)
  USING INDEXTYPE vectxsys.VECTOR
  PARAMETERS('similarity_measure my_new_measure');
```

The other way of using the new measure for specific query is by specifying optional parameter of `CONTAINS` operator.

The following query uses `my_new_measure` preference for the similarity measure calculation even if the default measure preference is different for the given vector index.

```
SELECT *
  FROM news
 WHERE CONTAINS(
          text,
          TEXT_VECTOR('oracle:1;database:0.5'),
          'similarity_measure my_new_measure'
       ) > 0;
```

5.6 Token weight functions

This chapter contains the list of token weight functions implemented in the Vector Text cartridge and a guide for implementing user's own token weight function. Token weight functions are using the same principle of saving important precomputed values as similarity measure functions.

TF weight

The basic and most simple option for token weighting is the use of token frequency. The token frequency is defined as follows:

$$TF_{i,j} = \frac{token_j\, occurrence\, count\, in\, document\, D_i}{all\, tokens\, occurrence\, count\, in\, document\, D_i}$$

$$w_{i,j} = TF_{i,j}$$

It grants the basic principle: the more occurrences of token in given document the more significance of the token for the document. This token weight function can be used by the `TF_WEIGHT` preference.

LOG_TF weight

The Logarithmic token frequency is defined as following:

$$LOG\_TF_{i,j} = \begin{cases} 1 + ln\ TF_{i,j} & \text{if } TF_{i,j} > 0, \\ 0 & \text{otherwise.} \end{cases}$$

$$w_{i,j} = LOG\_TF_{i,j}$$

It reduces the importance of token frequency in collections of individual documents with various length and it also reduces the effect of a token with an unusually high token frequency within the given document. This token weight function can be used by the `LOG_TF_WEIGHT` preference.

NTF weight

The normalized token frequency weight is defined as follows:

$$NTF_{i,j} = \begin{cases} 0 & \text{for } TF_{i,j} < \epsilon, \\ \frac{1}{2} + \frac{1}{2} * \frac{TF_{i,j}}{\max_k (TF_{i,k})} & \text{otherwise.} \end{cases}$$

$$w_{i,j} = NTF_{i,j}$$

Such a definition decreases the size of data to be processed because it distinguishes the significant tokens from the insignificant ones. It also raises the weight values to normal digit places because even for the most frequent terms the tf value is very low. However the normalized tf value of all significant tokens vary between 0.5 and 1.

The normalized token frequency can be used as token weight function by the `NTF_WEIGHT` preference.

TF_ITF weight

The inverted token frequency (ITF) represents the importance of token in the whole collection and not only in the given document. It is defined in the following way:

$$ITF_j = \log\left(\frac{documents\ count\ in\ collection}{documents\ count\ containing\ token_j}\right)$$

The most simple weight function including ITF is then defined as follows:

$$w_{i,j} = TF_{i,j} * ITF_j$$

The descriptors suitable for such definition characterize only a relatively small number of documents from the given collection. They have higher weight than unsuitable descriptors, which are so common that they occur in many documents from the given collection.

This token weight function can be used by the `TF_ITF_WEIGHT` preference.

NTF_ITF weight

This token weight function is similar to the previous one and is defined as follows:

$$w_{i,j} = NTF_{i,j} * ITF_j$$

It integrates the advantages of data size decreasing from NTF weight and the descriptors stress from TF_ITF weight.

This token weight function can be used by the `NTF_ITF_WEIGHT` preference.

Normalized NTF_ITF weight

The normalized NTF_ITF weight minimizes the influence of document length on the token weight. The definition of weight function is following:

$$v_{i,j} = NTF_{i,j} * ITF_j$$

$$w_{i,j} = \frac{v_{i,j}}{\sqrt{\sum_k v_{i,k}^2}}$$

The resultant effect of the weight normalization is that the importance of the token in a given document depends on its frequency of occurrence relative to other tokens in the same document and not on its absolute frequency of occurrence.

This token weight function can be used by the `NORM_NTF_ITF_WEIGHT` preference.

5.6.1 Implementing user's own token weight

The process for user's own token weight implementation and use is analogous to the similarity measure implementation process. The three main steps have to be taken again. To begin with, the function has to be implemented, then the new preference using given function has to be created and finally the index with this new preference has to be created. The mentioned function can be created only in the `VECTXSYS` schema and it has to possess the following declaration structure:

```
function user_token_weight(
    weight_info vectxsys.TokenWeightInfo
  )
  return number;
```

The only parameter of `TokenWeightInfo` type contains all necessary information about token, document and index against which the token weight value is being calculated. The actual data for calculation should be taken from the *$TF* index table or memory structure for actual query. The calculated token weight value has to be returned as the function result.

Preference creation

The new preference for implemented weight function has to be created. All the attributes used by the implemented function have to be set. The only compulsory attribute that has to be set is named `FUNCTION` and its value means the name of the implemented token weight function.
The following example creates the preference named `my_new_weight` which uses the `my_new_weight` function from the `user_funcs` package for calculating the token weight value.

```
 begin
  VECTX_DDL.CREATE_PREFERENCE(
     'MY_NEW_WEIGHT',
     'USER_WEIGHT');
  VECTX_DDL.SET_ATTRIBUTE(
     'MY_NEW_WEIGHT',
     'FUNCTION',
     'user_funcs.my_new_weight');
end;
```

New token weight use

The newly implemented token weight function can be used as the default for all queries on the index created.
The following statement creates a vector index named `news_ind` on `text` column in the `news` table and the preference called `my_new_weight` is used as the default token weight for all queries on given index.

```
CREATE INDEX new_ind ON news(text)
   USING INDEXTYPE vectxsys.VECTOR
   PARAMETERS('token_weight my_new_weight');
```

The another way to use new token weight for the specific query is to specify the optional parameter of `CONTAINS` operator.
The following query uses `my_new_weight` preference for the token weight calculation even if the default token weight preference is different from the given vector index.

```
SELECT *
  FROM news
 WHERE CONTAINS(
          text,
          TEXT_VECTOR('oracle:1;database:0.5'),
          'token_weight my_new_weight'
       ) > 0;
```

Chapter 6

Test

6.1 Test environment and data

The LISA (Library and Information Science Abstracts) collection was used for the purpose of testing the Vector Text implementation. LISA [1] is one of standard IR collections. It is provided by Peter Willett of Sheffield University to support research investigations. The collection contains 6004 abstracts of documents. Along with the documents it also provides 35 queries together with the list of all relevant documents which should ideally be returned by the IR system. Each pre-defined query contains its definition in natural language. Several words or collocations which should be used for the searching are included too. These document relevance judgments were prepared manually as a part of course which took place at Sheffield University.

All tests were performed on the notebook with the parameters that are presented in Table 6.1.

Processor	Intel T7200 Core2Duo, 2.00 GHz
Disk	120 GB, 5400 RPM
Memory	1 GB RAM
OS	Microsoft Windows XP Professional, Service Pack 2
Oracle database	version 9.2.0.1

Table 6.1: Parameters of the test environment

6.2 Index manipulation test

The first test measures the time needed for creation of the chosen index. Both indextypes, context as well as vector, were created on the test table which was holding the whole LISA collection. Its structure was the following:

[1] downloaded from http://www.dcs.gla.ac.uk/idom/ir_resources/test_collections/lisa/

```
CREATE TABLE test_docs (
  id NUMBER,
  text VARCHAR2(4000)
);
```

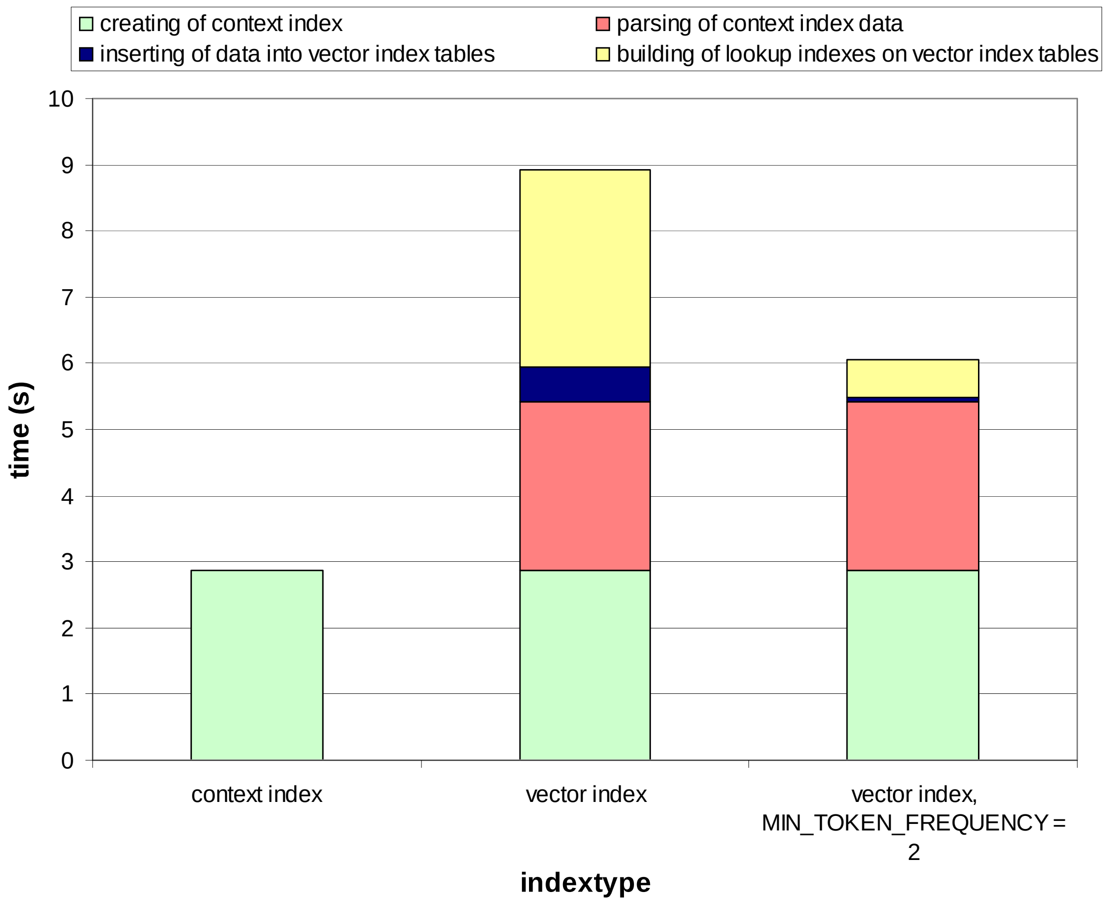

Figure 6.1: Time of index creation

The Figure 6.1 shows the comparison of time spent on creating the context and vector indexes. Time needed for creating the vector index has to be always higher than for the context index because of the design of Vector Text cartridge. At first it is created the context index and then - using its data - the vector index is subsequently created. The main difference in the time result is caused by the number of records being hold by the index data tables. The precise counts of held records shows the Table 6.2. The context index holds one record for one text token, whereas the vector index holds one record for the combination of token and the document where the given token occurs. This difference in the amount of held records results in the growth of time needed

	Oracle Text	Vector Text	Vector Text optimized
$I	19164	19164	19164
$K	6004	6004	6004
$TI	-	19164	5784
$TF	-	263626	41887

Table 6.2: Number of records being hold by index tables

for inserting those records into the *$TF* table and building lookup indexes on this table. The distinct structure of index data is forced by the different approach for the searching using Boolean or Vector space model. The Boolean model always searches for all documents to the specified token, whereas the Vector space model often needs to find tokens occuring in the specified document as well as documents including the specified token. More details about differences in context and vector index structures are discussed in Section 5.1.2.

The Figure 6.2 shows the other effect of the higher amount of records which need to be held in the vector index tables. The index implementation has increased demands on the disk space for holding the vector index data than for holding the context index data.

The previously described disadvantages of the Vector Text cartridge can be decreased by specifying `MIN_TOKEN_FREQUENCY` value (see Appendix A.2.1 for details). Thanks to this option, the extremely low values, which will not fundamentally influence the query results, can be left out of vector index data. Both Figures 6.1 and 6.2 show in the last columns the influence of creating vector index having set the `MIN_TOKEN_FREQUENCE` to 2. Let's realize that this example is too penetrative because the abstracts are short texts, which do not include a lot of words more then once. So the effect in the collections holding the bigger text documents would not be as massive as presents this example.

6.3 Querying test

This section contains the results for queries, which were evaluated on the LISA collection. It was chosen 5 out of the total 35 pre-defined queries. This choice was made randomly, the only request was to include both queries having associated many and low relevant documents into the test. Only the first 20 returned documents having the highest score values were considered in the following test. The queries used for Oracle Text and Vector Text searches had to be slightly different because of the necessary usage of logical connectives in Oracle Text queries, but their meaning was the same. Each query was built only from the words and collocations that are specified for each pre-defined query.

The Vector Text queries were optimized during the test by specifying query token

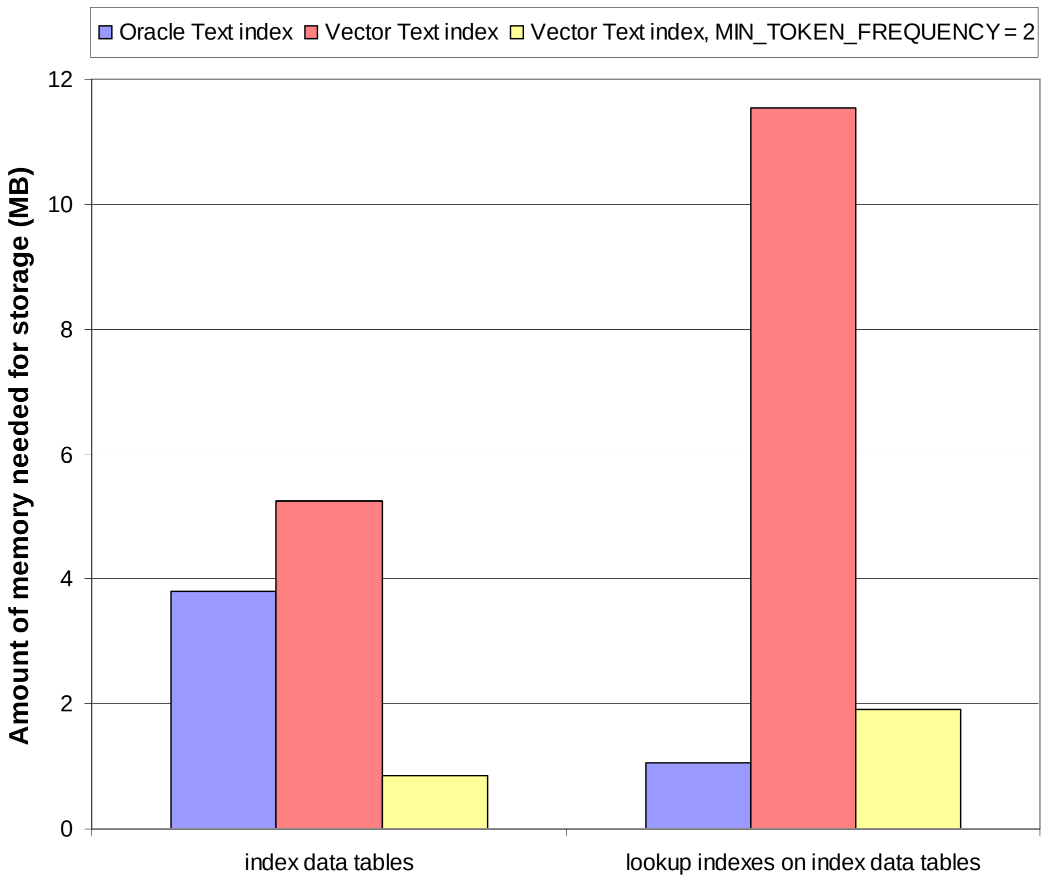

Figure 6.2: Memory demands for saving index data

weights to achieve the best results for each query. The final results of original and optimized queries were compared and they are presented further in this section.

Let's present here the example of the specific queries that were used during the testing. The query 3 used for the searching of context index was the following:

```
SELECT SCORE(1) sc,
       id
  FROM test_docs
 WHERE CONTAINS(text,'CHEMISTRY or CHEMICAL or PATENTS',1) > 0
 ORDER BY sc DESC;
```

When the vector index wanted to be searched instead of the context index, the same query 3 was specified the following way:

```
SELECT SCORE(1) sc,
       id
  FROM test_docs
 WHERE CONTAINS(
          text,
          NONINDEXED_TEXT('CHEMISTRY, CHEMICAL, PATENTS'),
          'similarity_measure SCALAR_MEASURE
           token_weight TF_WEIGHT',
          1) > 0
 ORDER BY sc DESC;
```

The queries using Vector Text cartridge was optimized to return the best possible results, let's see the example of such optimization via specifying the weights for query tokens:

```
SELECT SCORE(1) sc,
       id
  FROM test_docs
 WHERE CONTAINS(
          text,
          TEXT_VECTOR('CHEMISTRY:0; CHEMICAL:1; PATENTS:4'),
          'similarity_measure SCALAR_MEASURE
           token_weight TF_WEIGHT',
          1) > 0
 ORDER BY sc DESC;
```

The full list of queries, which were used in the test, is introduced by the Appendix B. All tables presented in Appendix B include only the different parts of queries being specified by the CONTAINS operator for the individual queries, because their structure was always the same as demonstrated above.

The Figures 6.3 and 6.4 present the first part of the test results, where the count of relevant documents being returned by the individual queries was compared. The comparisons according to the used token weight method are represented on the Figure 6.3, the scalar measure was used for counting all score values in this case. By contrast, the Figure 6.4 shows the results of the use of different similarity measure methods evaluated on the index using the TF weighting.

The other part of test results is composed by the Figure 6.5 and 6.6, where the proportion of returned relevant documents according to their order in the returned documents is presented. The exact definition of the proportion value is introduced further in this section. The Figure 6.5 shows the comparison according to the used token weight method, whereas the differences in use of alternative similarity measure methods are presented by the Figure 6.6.

The computation of the proportion values considers the first 20 documents returned by the query, they are ordered starting from the ones having the highest score

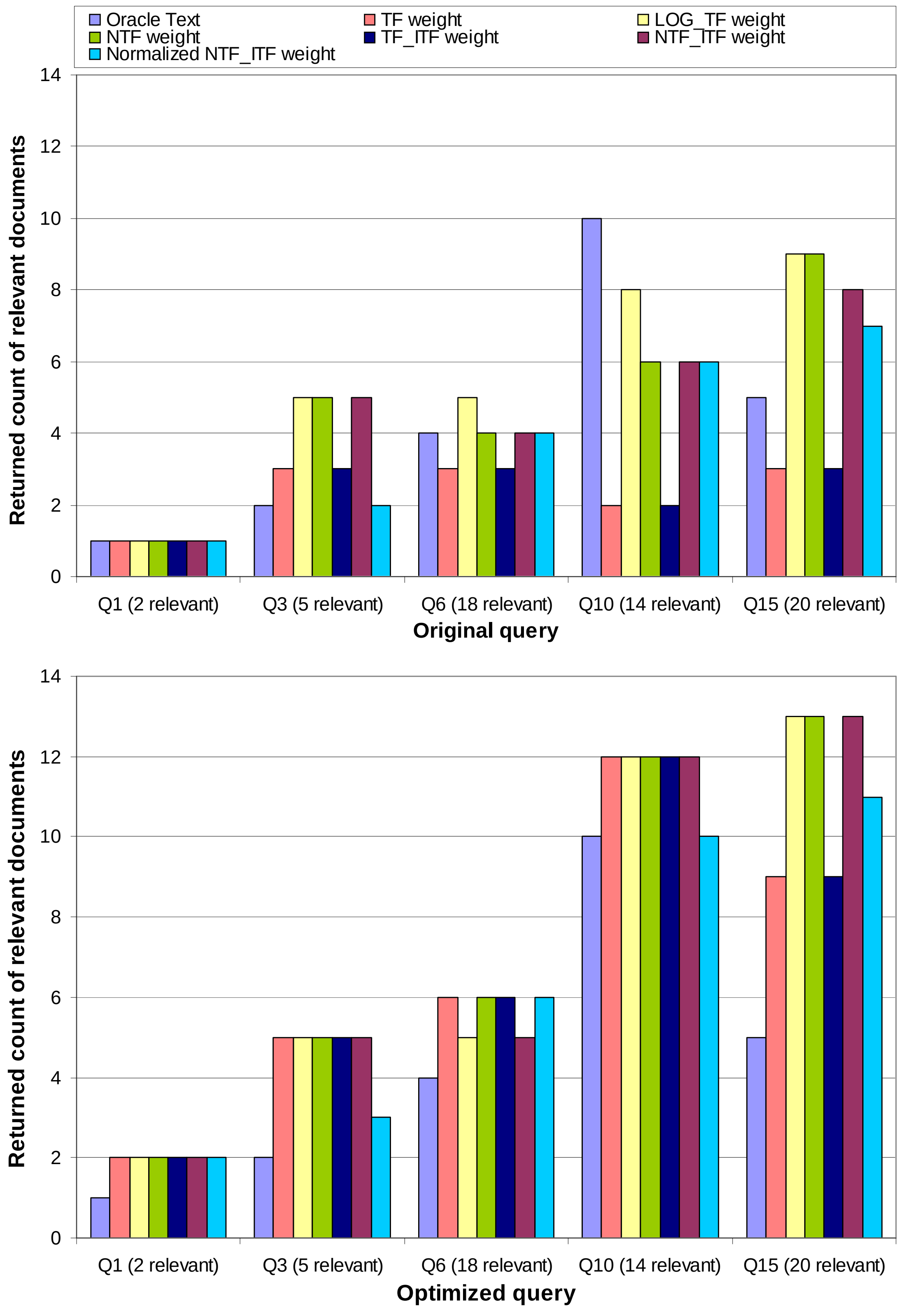

Figure 6.3: Count of retrieved relevant documents for the different token weight methods - original and optimized queries

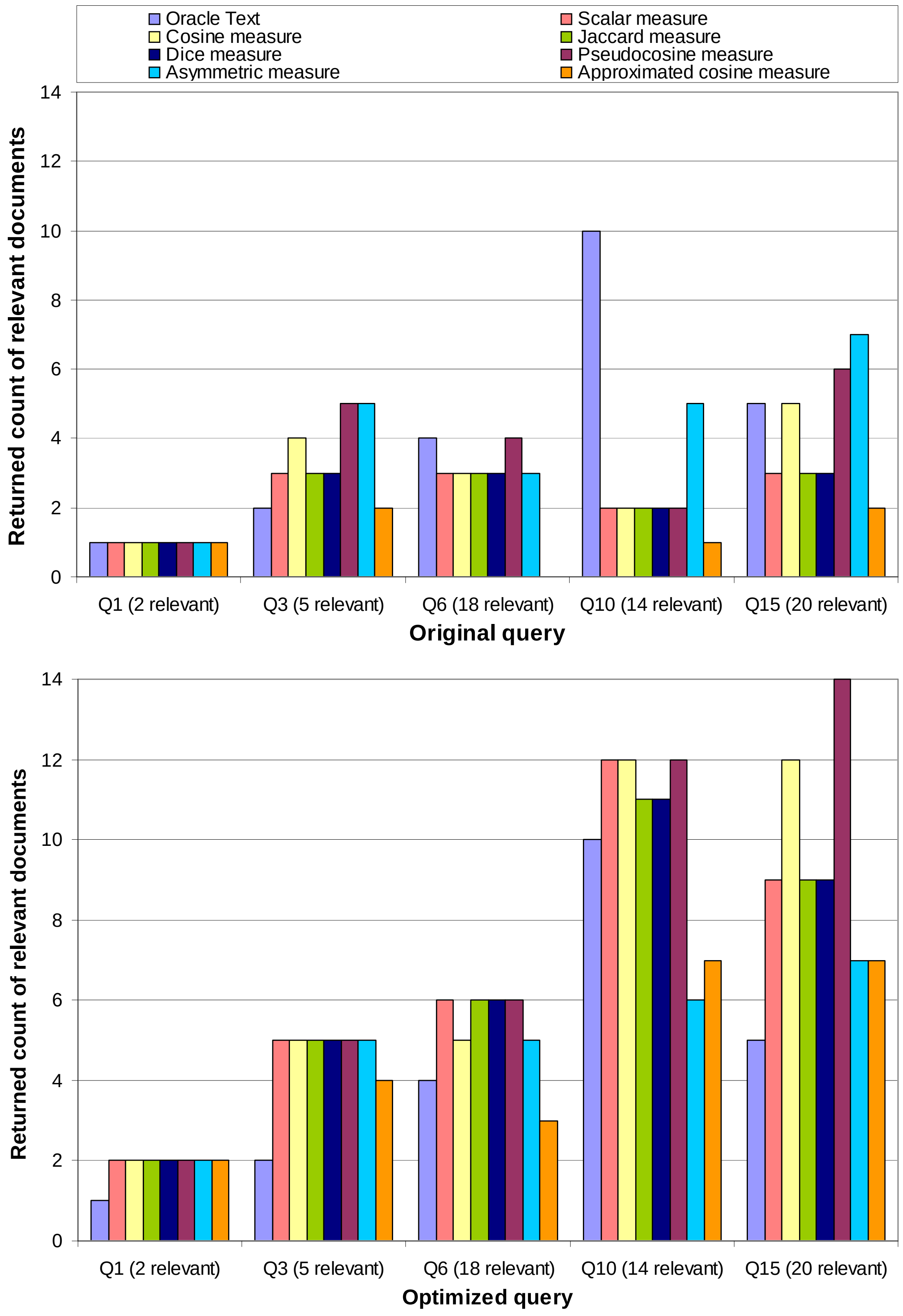

Figure 6.4: Count of retrieved relevant documents for the different similarity measures - original and optimized queries

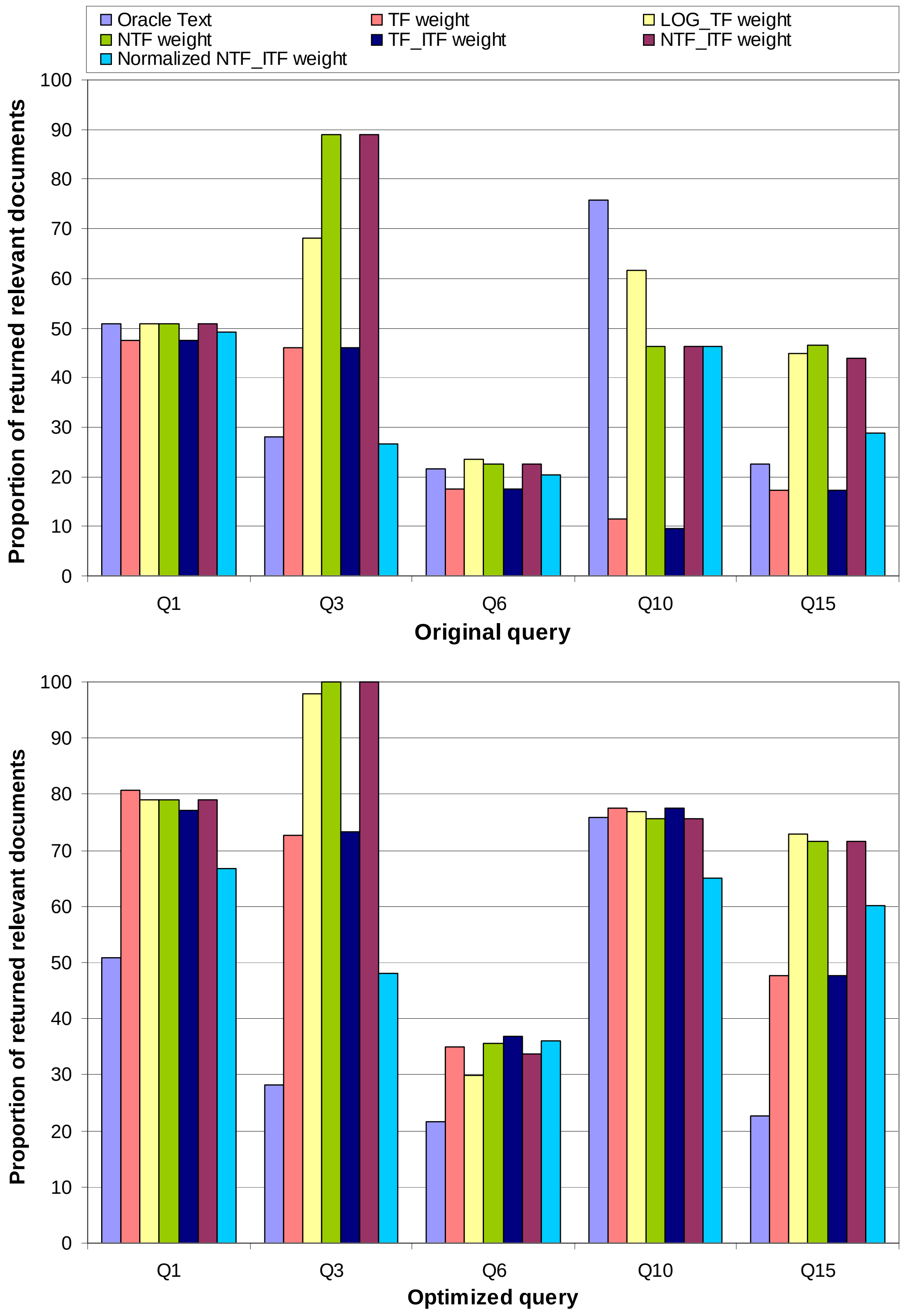

Figure 6.5: The proportion of retrieved relevant documents for the different token weight methods - original and optimized queries

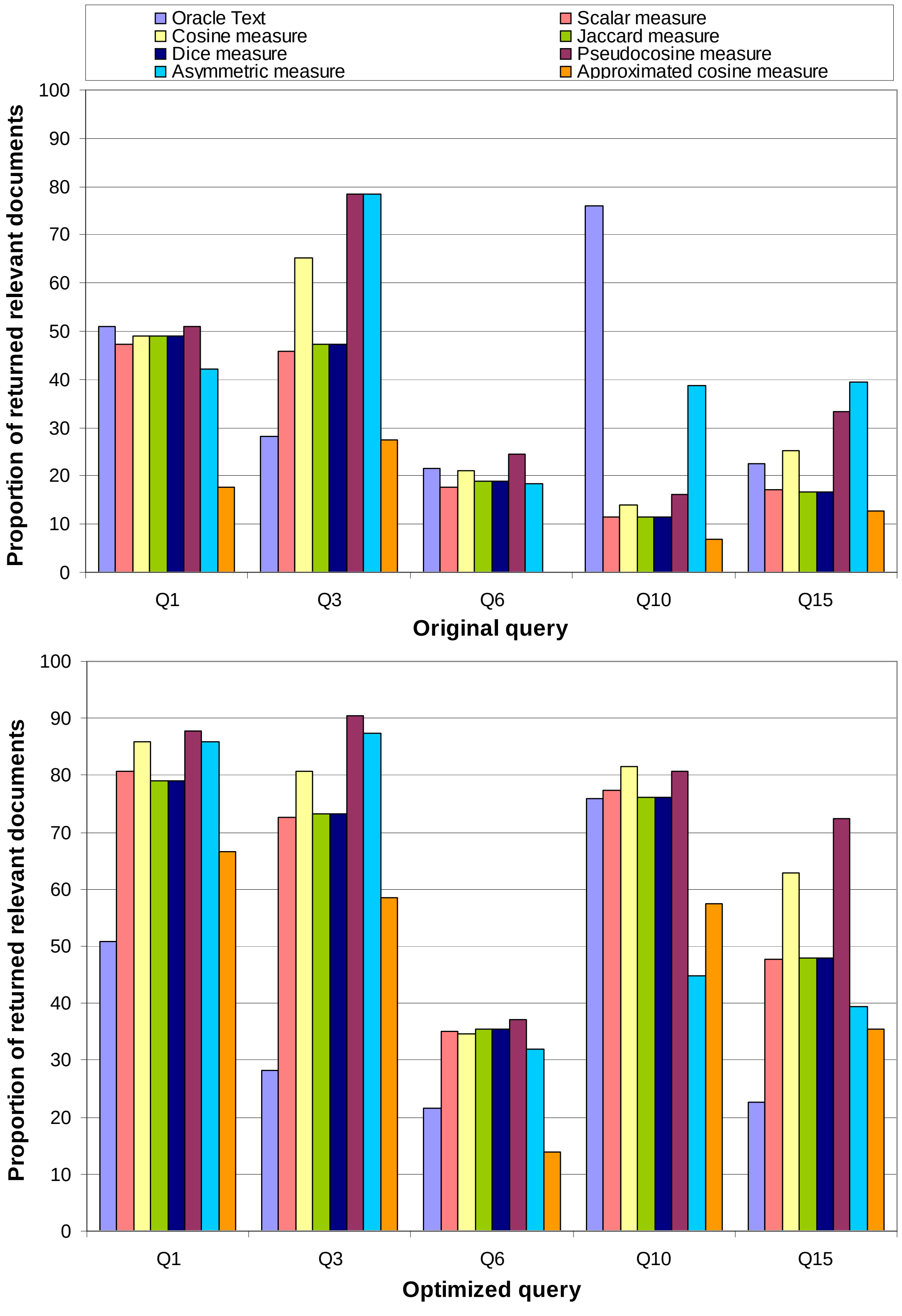

Figure 6.6: The proportion of retrieved relevant documents for the different similarity measure methods - original and optimized queries

value. Let's present here the formula for calculating the proportion value. Let's define the sum of orders of all relevant documents the following way

$$S = \sum_{i=1}^{n} o_i + (k - n) * E$$

where n stands for the number of relevant documents being returned to the user among the first 20 documents, o_i means the order of i-th relevant document in the returned collection of documents, k presents the total number of relevant documents and E stands for default order of the relevant document, which is not returned to the user among first 20 returned documents ($E > 20$ is the required condition, the test value of E was set to 30). The resultant proportion value is then defined by the formulas

$$MAX_S = k * E$$
$$MIN_S = \frac{n*(n+1)}{2}$$
$$P = \frac{MAX_S - S}{MAX_S - MIN_S}$$

where P presents the resultant proportion value, MAX_S stands there for the maximal value of S which means that no relevant documents were retrieved and MIN_S stands there for the minimal value of S which means that all relevant documents were retrieved before the first nonrelevant one.

The definition presented above is not easily understandable and so it follows the description of the proportion value calculation principles. The 100% proportion means that all relevant documents were returned by the query before the first document being non-relevant, whereas the 0% proportion means that no relevant document was returned by the query. Furthermore the lower order has a given relevant document in the returned collection the higher is the resultant proportion value.

Let's summarize here some facts, which can be observed from the results of the query testing. According to the retrieved values, there is no universal method, which gives the best results in every situation. The weight methods that are using normalization give slightly better results then the same methods without the normalization. On the other hand their processing takes longer. The methods implemented by the Vector Text cartridge are undoubtedly comparable to the Oracle Text. What is more, the query optimization can be processed easily in Vector Text via specifying the weights for query tokens. These optimized queries have much better results comparing to the Oracle Text.

Chapter 7

Conclusion

The vector IRS cartridge designed in this thesis fulfills all the proposed goals and the implementation shows that it is viable.

The resultant cartridge highlights the main advantages of the Vector space model approach. A high amount of different similarity measure and token weight functions is prepared to be used for querying. An easy interface for creating and adding more such functions is available too. Each of the similarity measure and token weight functions can be suitable for different searching conditions, with the purpose of enable the user test easily several functions and decide which one will work best in the particular situation. By contrast, Oracle Text uses only one algorithm for score calculation. By creating vector index via Vector Text cartridge, user can experiment and compare results of queries over both context and vector indexes.

The possibilities of entering the query were also extended in Vector Text cartridge. Oracle Text offers only one way of query specification (searched tokens connected by logical conjunctions), whereas Vector Text enables more possibilities. Users can specify query vector with weights assigned to its individual tokens, they can also enter part of a text in natural language or even the whole document to which they search the similar documents. The wide potential of queries using a thesaurus is enabled in Vector Text too.

On the other hand, some extra features of query specification from the Oracle Text was not implemented into the Vector Text cartridge. The vector index data does not include the positions of token occurrences in individual documents, thus the proximity operator and collocations cannot be specified in the Vector Text. The fuzzy operator also absents from the Vector Text, because any existing implementation of this operator from Oracle Text could not be re-used in Vector Text cartridge. The own implementation of fuzzy operator could be a subject for a future possible extension of the Vector Text cartridge. But the solution for the case, that the use of any of the mentioned operators is imperative, is very simple. The Boolean query can be always specified. The context index, on which the vector index is built, will process such query.

Due to the chosen architecture, the performance of all DDL operations being processed for Vector Text index is worse than for Oracle Text index. That is due to the context index creation in Vector Text first and vector index building up to make use of the context index data later then. This obvious disadvantage of Vector Text performance is counterbalanced with many benefits, eg. the possibility to use both context and vector indexes on specific data at the same time, the possibility of comparison of their results and performance of individual queries, the possibility to keep all different options during the index creation that are used in Oracle Text etc.

One of the main contributions of the Vector Text cartridge is that it kept the user interface very similar to the Oracle Text. For example, several operators were left and even extended while the principle of index preferences is also preserved there. Similar public packages for application developers in Vector Text were implemented as well.

A lot of original functionality is available in Vector Text too. All the parameters inexhaustible for index creation from the documents storage options via the filtering and tokenization to the stoplists usage possibilities were preserved.

I believe that the Vector Text cartridge will be widely used thanks to its features for both querying and further extending, comparing of results of the different methods etc.

Bibliography

[1] C. L. Clarke, G. V. Cormack & A. E. Tudhope. *Relevance ranking for one to three term queries*. Proceedings of RIAO'97, 388-400, 1997.

[2] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer & R. Harshman. *Indexing by latent semantic analysis*. Journal of the American Society for Information Science, 41(6), 391-407, 1990.

[3] W. Kraaij. *Variations on Language Modeling for Information Retrieval*. Print Partners Ipskamp, Enschede, 2004.

[4] T. Kyte. *Effective Oracle by Design*. Osborne ORACLE Press Series, 2003.

[5] D. L. Lee, Huei Chuang, K. Seamons. *Document ranking and the vector-space model*. Software IEEE, 14(2), 67-75, 1997.

[6] H. Luhn. *A statistical approach to mechanized encoding and searching of literary information*. IBM journal, 309-317, 1957.

[7] Ch. D. Manning, P. Raghavan, H. Schutze. *An Introduction to Information Retrieval, Preliminary Draft*. Cambridge University Press, 2007.

[8] M. Maron & J. Kuhns. *On relevance, probabilistic indexing and information retrieval*. Journal of the Association for Computing Machinery, 7, 216-244, 1960.

[9] D. R. H. Miller, T. Leek & R. M. Schwartz. *A hidden markov model information retrieval system*. Proceedings of the 22th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '99), ACM Press, 214-221, 1999.

[10] *Oracle Text Reference*. Release 9.2, http://download-uk.oracle.com/docs/cd/B10501_01/text.920/a96518.pdf, 2002.

[11] *Oracle9i Data Cartridge Developer's Guide*. Release 2 (9.2), http://download-uk.oracle.com/docs/cd/B10501_01/appdev.920/a96595.pdf, 2002.

[12] J. M. Ponte & W. B. Croft. *A language modeling approach to information retrieval.* Proceedings of the 21th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '98), ACM Press, 275-281, 1998.

[13] G. Salton. *Automatic Text Processing - The Transformation, Analysis and Retrieval of Information by Computer.* Addison-Wesley Publishing Company, Reading(MA), 1989.

[14] G. Salton, E. Fox & H. Wu. *Extended boolean information retrieval.* Communications od the ACM, 26(12), 1022-1036, 1983.

[15] G. Salton & C. Buckley. *Term-weighting approaches in automatic text retrieval.* Information Processing & Management, 24(5), 513-523, 1988.

[16] T. Saracevic. *Evaluation of Evaluation in Information Retrieval.* Proceesings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, 137-146, 1995.

[17] C. van Rijsbergen. *A non-classical logic for information retrieval.* Computer Journal, 29, 481-485, 1986.

[18] S. K. M. Wong, W. Ziarko, V. V. Raghavan & P. C. N. Wong. *On extending the vector space model for boolean query processing.* Proceesings of the 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '86), ACM Press, 175-185, 1986.

[19] S. K. M. Wong, W. Ziarko, V. V. Raghavan & P. C. N. Wong. *On modeling of information retrieval concepts in vector space.* TODS, 12(2), 299-321, 1987.

Appendix A

Vector Text user documentation

A.1 Installation

Vector Text cartridge was created as the extension to Oracle Text, so it can be installed only on machines with existing installation of Oracle Text in version 9.2 (9.2.0.1.0 or higher)[1]. The following three steps have to be done to install Vector Text cartridge: The following three steps has to be done to install Vector Text cartridge:

1. Create schema for Vector Text - Run script `dr0csys` from `source` directory in SQL*Plus as user with DBA role granted. This script will create schema `VECTXSYS` and role `VECTXAPP` for Vector Text cartridge. Role `VECTXAPP` has granted right `EXECUTE` for all public packages of Vector Text cartridge and it has also granted role `CTXAPP` for using public packages from Oracle Text. You will have to specify three values during run of the script `dr0csys`: `VECTXSYS` user password, tablespace and temporary tablespace.

2. Install the shared library - For a Windows platform, run a script `install_lib.bat` which will copy the shared library `vectxx9.dll` to the `%ORACLE_HOME%\bin` directory. For a Unix platform, run the script `install_lib.sh` which will copy the shared library `vectxx9.so` to the `$ORACLE_HOME/bin` directory.

3. Install the Vector Text database objects - Connect as `VECTXSYS` user to the database using SQL*Plus and run `dr0install` script to install Vector Text cartridge. You will have to specify ORACLE_HOME value as the input parameter of the script `dr0install`.

[1]Vector Text was not tested with Oracle Text version 10.1 and higher, the important changes in privileges principles were done there and so the proper functionality of Vector Text is not ensured.

A.2 SQL Statements and Operators

This section describes the SQL statements and Vector Text operators used for creating and managing Vector Text indexes and performing Vector Text queries.

A.2.1 CREATE INDEX

This subsection describes the `CREATE INDEX` statement as it pertains to creating a Vector Text domain index.

Syntax

```
CREATE INDEX [schema.]index_name ON [schema.]table(column)
  INDEXTYPE IS vectxsys.VECTOR [PARAMETERS(paramstring)];
```

[schema.]index_name

Specify the name of the Vector index to create.

[schema.]table(column)

Specify the name of the table and column to index The column you specify must be one of the following types: `CHAR`, `VARCHAR`, `VARCHAR2`, `BLOB`, `CLOB`, `BFILE`, `XMLType` or `URIType`. Indexes on multiple columns are not supported with the vector index type. You must specify only one column in the column list.

PARAMETERS(paramstring)

Create vector index with given preferences. All preferences as in Context index `CREATE INDEX` statement can be specified and the following preferences are added in Vector Text:

- `SIMILARITY_MEASURE` – Name of similarity measure which is default for queries issued on created index.
- `TOKEN_WEIGHT` – Name of token weight which is default for queries issued on created index.
- `USE_CONTEXT_INDEX` – Name of context index on which is vector index based. If this parameter is set, no other parameters for Context index can be specified and vector index is based on given existing context index. Otherwise it can be specified any parameter for Context index, which is being created as a part of vector index.

- `MIN_TOKEN_FREQUENCY` – Minimal count of occurrences of the token in individual document to include the token frequency value into the resultant index. This option is used for decreasing size of index tables by leaving out the extremely rare data.

Notes

Partitioning is not implemented for Vector Text cartridge!

Examples

The following example creates a vector index called new_idx on the text column in news table. Default preferences are used.

```
CREATE INDEX new_idx ON news(text) INDEXTYPE IS vectxsys.VECTOR;
```

The following example creates a vector index called new_idx on the text column in news table. The index is created with a custom stoplist preference called stop (has to be created using `CTX_DDL.CREATE_PREFERENCE` from Oracle Text) and similarity measure preference called dice_measure. Default preferences are used for the unspecified attributes.

```
CREATE INDEX new_idx ON news(text) INDEXTYPE IS vectxsys.VECTOR
  PARAMETERS('STOPLIST stop SIMILARITY_MEASURE dice_measure');
```

The following example creates a vector index called new_idx on the text column in news table. Created index is based on existing Context index called news_ctx_ind and all index preferences are taken out of it. In that case only `SIMILARITY_MEASURE` and `TOKEN_WEIGHT` preferences can be also specified in parameters list of `CREATE INDEX` statement. The data about tokens with at least 5 occurances in a document are saved into index tables.

```
CREATE INDEX new_idx ON news(text) INDEXTYPE IS vectxsys.VECTOR
  PARAMETERS('USE_CONTEXT_INDEX news_ctx_ind MIN_TOKEN_FREQUENCY 5');
```

A.2.2 CONTAINS

`CONTAINS` operator returns a relevance score for every row selected. This value can be obtained by `SCORE` operator. Returned value depends on used similarity measure and token weight methods.

Syntax

```
CONTAINS([schema.]column,query[,query_params VARCHAR2]
   [,label NUMBER])
```

[schema.]column

Text column to be searched on. This column must have Vector Text index associated with it.

query

Query specification, use one of the following functions is required:

- `TEXT_VECTOR` – text vector
- `INDEXED_TEXT` – identification of text which is allready indexed by Vector Text index
- `NONINDEXED_TEXT` – text abstract which is being searched for

For more detailed specification see following subsections of this chapter.

query_params

Allows specifying of optional `SIMILARITY_MEASURE` and `TOKEN_WEIGHT` parameters of the query.

label

The label that identifies the score generated by the `CONTAINS` operator, optional usage.

Examples

The following example searches for all titles of documents where text column contains the words oracle and database and the word oracle is preffered more.

```
SELECT title
  FROM news
 WHERE CONTAINS(
          text,
          TEXT_VECTOR('oracle:1;database:0.5')
       ) > 0;
```

A.2.3 INDEXED_TEXT

`INDEXED_TEXT` operator prepares specified text to be compared to others by `CONTAINS` operator. This text has to be saved in column which has Vector Text index associated with it.

Syntax

```
INDEXED_TEXT(table_spec VARCHAR2, key_spec VARCHAR2)
```

table_spec

Column specification where the text to be compared is saved. The specification has to be in `[schema.]table(column)` form.

key_spec

Row identification where the text to be compared is saved. The specification has to be in `column(value)[,column2(value2)...]` form.

Examples

The following example searches for all titles in news table having texts similar to text being indexed by vector index. The source text for similarity comparison is saved in text column of history table in current user's schema and the record containing given text is specified by 2005 value in year column and 1123 value in text_id column. The resultant titles are returned sorted starting with the most similar texts.

```
SELECT title
  FROM news
 WHERE CONTAINS(
          text,
          INDEXED_TEXT(
             'history(text)',
             'year(2005),text_id(1123)'),
          1
       ) > 0.5
 ORDER BY SCORE(1) DESC;
```

A.2.4 NONINDEXED_TEXT

`NONINDEXED_TEXT` operator prepares specified text to be compared to others by `CONTAINS` operator.

Syntax

```
NONINDEXED_TEXT(text [,index_name VARCHAR2])
```

text

The text to be compared, it has be one of the following types: `VARCHAR2`, `CLOB`, `BLOB`, `BFILE`, `XMLType`, `URIType`.

index_name

Name of the Vector Text index which preferences to use when preparing text (eg. `LEXER` preference). Usage is optional, default preferences used when not specified. The specification of index name has to be in `[schema.]index_name` form.

Note

Oracle Text thesaurus usage is supported inside the text.

Examples

The following example searches for all titles in news table having texts similar to source text. Assume that this source text for similarity comparison is returned by GetSourceText in CLOB datatype. The resultant titles are returned sorted starting with the most similar texts. Default preferences are used for parsing and indexing of source text.

```
declare
  lclb_text CLOB;
begin
  lclc_text := GetSourceText(...);

  SELECT title
    FROM news
   WHERE CONTAINS(
           text,
           NONINDEXED_TEXT(lclb_text),
           1
         ) > 0.5
   ORDER BY SCORE(1) DESC;
end;
```

The next example searches for all titles in news table having texts similar to source text which is taken from news_history table. The resultant titles are returned sorted starting with the most similar texts. Preferences for parsing and indexing of source text are used the same as for history_vect_ind index in dataminer schema.

```
SELECT n.title
  FROM news n,
       ( SELECT NONINDEXED_TEXT(
                    text,
                    'dataminer.history_vect_ind'
                 ) source_text
           FROM news_history
          WHERE history_id = 1233
        ) s
 WHERE CONTAINS(n.text, s.source_text, 1) > 0.5
 ORDER BY SCORE(1) DESC;
```

A.2.5 TEXT_VECTOR

`TEXT_VECTOR` operator creates query from specified similarity vector for usage by `CONTAINS` operator.

Syntax

```
TEXT_VECTOR(vector_spec VARCHAR2)
```

vector_spec

Similarity vector specification, its value has to be in format `token:weight[;token2:weight2...]`.

Note

Oracle Text thesaurus usage is supported in tokens of similarity vector.

Examples

The following example searches for all titles in news table where text column contains the words oracle and database and the word oracle is preffered more. The resultant titles are returned sorted starting with the most similar texts.

```
SELECT title
  FROM news
 WHERE CONTAINS(
          text,
          TEXT_VECTOR('oracle:1;database:0.5'),
          1
       ) > 0.5
 ORDER BY SCORE(1) DESC;
```

The next example shows usage of Oracle Text thesaurus in query, it searches for all titles in news table where text column contains the word oracle and any synonym of the word delete. The resultant titles are returned sorted starting with the most similar texts.

```
SELECT title
   FROM news
  WHERE CONTAINS(
           text,
           TEXT_VECTOR('oracle:1;SYN(delete):0.5'),
           1
        ) > 0.5
  ORDER BY SCORE(1) DESC;
```

A.2.6 SCORE

`SCORE` operator in a `SELECT` statement returns the score values produced by a `CONTAINS` query.

Syntax

```
SCORE(label NUMBER)
```

label

A number to identify the score produced by the query, it is used to identify the score in the `CONTAINS` clause.

Example

The following example searches for all titles and its score values in news table where abstract column or text column contains the words oracle and database and the word oracle is preferred more. The resultant titles are returned sorted by the most similar abstracts.

```
SELECT title, SCORE(1), SCORE(2)
  FROM news
 WHERE CONTAINS(
          abstract,
          TEXT_VECTOR('oracle:1;database:0.5'),
          1
       ) > 0.5
       OR
       CONTAINS(
          text,
          TEXT_VECTOR('oracle:1;database:0.5'),
          2
       ) > 0.1
 ORDER BY SCORE(1) DESC, SCORE(2) DESC;
```

A.2.7 ALTER INDEX

The `ALTER INDEX` statement is described here as it pertains to managing a Vector Text domain index.

RENAME Syntax

```
ALTER INDEX [schema.]index_name RENAME TO new_index_name;
```

[schema.]index_name

Specify the name of the index to rename.

new_index_name

Specify the new name for `[schema].index`. The `new_index_name` parameter can be no more than 25 characters.

REBUILD Syntax

```
ALTER INDEX [schema.]index_name
   REBUILD [PARAMETERS(paramstring)];
```

PARAMETERS(paramstring)

Rebuild with changed parameters, rebuild to optimize or synchronize.

replace(optional_preference_list)

Rebuilds index with changed given preferences. If the vector index was not created as based on existing Context index (using `USE_CONTEXT_INDEX` parameter in `CREATE INDEX statement`) it can be specified any appropriate preference for Context index `ALTER INDEX` statement. Following preferences added in Vector Text:

- `SIMILARITY_MEASURE` – Name of similarity measure which is default for queries on given index.
- `TOKEN_WEIGHT` – Name of token weight which is default for queries on given index.

Examples

The following statement renames the index old_ind to new_ind:

```
ALTER INDEX old_ind RENAME TO new_ind;
```

Changing the default similarity measure used for new_ind index querying to cosine_measure shows the following example:

```
ALTER INDEX new_ind REBUILD
  PARAMETERS('replace similarity_measure cosine_measure');
```

New_ind index optimization for oracle token can be processed by calling the following statement:

```
ALTER INDEX new_ind REBUILD
  PARAMETERS('optimize token oracle');
```

A.2.8 DROP INDEX

Use `DROP INDEX` to drop a specified Vector Text index. If given index was not created as based on existing Context index (using `USE_CONTEXT_INDEX` parameter) it drops Context index too.

Syntax

```
DROP INDEX [schema.]index_name;
```

Example

The following example drops an index named new_ind in the current user's database schema.

```
DROP INDEX new_ind;
```

A.3 Public packages

This chapter contains description of all public packages which support users usage of Vector Text cartridge.

A.3.1 VECTX_ADM

`VECTX_ADM` package is used to administer Vector Text data dictionary.

SET_PARAMETER

This procedure sets system-level parameters for index creation.

Syntax

```
VECTX_ADM.SET_PARAMETER (
  param_name IN VARCHAR2,
  param_value IN VARCHAR2
);
```

param_name

Specify the name of parameter to set, which can be one of the following:

- `default_similarity_measure` – default similarity measure procedure
- `default_token_weight` – default token weight function
- `log_directory` – default directory for log files

Example

The following example sets similarity measure and token weight parameters used in Vector Text by default. The similarity measure parameter is set to dice measure and token weight parameter is set to normalized token frequency.

```
begin
  VECTX_ADM.SET_PARAMETER (
     'default_similarity_measure',
     'dice_measure');
  VECTX_ADM.SET_PARAMETER (
     'default_token_weight',
     'ntf_weight');
end;
```

A.3.2 VECTX_DDL

`VECTX_DDL` package is used to create and manage the preferences and precomputed values required for Vector Text.

SYNC_INDEX

Use this procedure to synchronize the index. It processes insert, updates and deletes to the base table.

Syntax

```
VECTX_DDL.SYNC_INDEX (
  index_name IN VARCHAR2
);
```

index_name

Specify the name of index to synchronize.

Example

The following example performs synchronization of news_text_ind vector index in dataminer schema.

```
begin
  VECTX_DDL.SYNC_INDEX ('dataminer.news_text_ind');
end;
```

OPTIMIZE_INDEX

Use this procedure to optimize the index. You optimize your index after you synchronize it. Optimizing the index removes old data and minimizes index fragmentation. Optimizing index can improve query response time.

Syntax

```
VECTX_DDL.OPTIMIZE_INDEX (
  index_name IN VARCHAR2,
  opt_level IN VARCHAR2,
  token IN VARCHAR2 DEFAULT NULL
);
```

index_name

Specify the name of index to optimize.

opt_level

Specify optimization of level as a string. You can specify one of the following methods:

- `FAST` – Compacts fragmented rows, old data are not removed
- `FULL` – Compacts fragmented rows, old data are removed
- `TOKEN` – Compacts fragmented rows for specified token, old data are removed

token

Specify the token index to be optimized.

Examples

The following example performs fast optimization of news_text_ind vector index in dataminer schema, which compacts fragmented index rows but doesn't delete old data.

```
begin
  VECTX_DDL.OPTIMIZE_INDEX (
     'dataminer.news_text_ind',
     'FAST');
end;
```

The next example performs optimization of news_text_ind vector index in dataminer schema only for oracle token. It compacts fragmented index rows for oracle token and deletes old data.

```
begin
  VECTX_DDL.OPTIMIZE_INDEX (
     'dataminer.news_text_ind',
     'TOKEN',
     'oracle');
end;
```

CREATE_PREFERENCE

Creates a preference in the Vector Text data dictionary. You specify preferences in the parameter string of `CREATE INDEX`, `ALTER INDEX` or `CONTAINS`.

Syntax

```
VECTX_DDL.CREATE_PREFERENCE (
  preference_name IN VARCHAR2,
  object_name IN VARCHAR2
);
```

preference_name

Specify the name of preference to be created.

object_name

Specify the name of preference type.

Example

The following example creates preference my_new_measure for user_measure object.

```
begin
  VECTX_DDL.CREATE_PREFERENCE (
     'my_new_measure',
     'user_measure');
end;
```

DROP_PREFERENCE

The `DROP_PEFERENCE` procedure deletes the specified preference from the Vector Text data dictionary. Dropping a preference does not affect indexes that have already been created using that preference.

Syntax

```
VECTX_DDL.DROP_PREFERENCE (
  preference_name IN VARCHAR2
);
```

preference_name

Specify the name of preference to be dropped.

Examples

The following example drops preference my_new_measure.

```
begin
  VECTX_DDL.DROP_PREFERENCE (
     'my_new_measure');
end;
```

SET_ATTRIBUTE

This procedure sets a preference attribute. You use this procedure after you have created a preference with `VECTX_DDL.CREATE_PREFERENCE`.

Syntax

```
VECTX_DDL.SET_ATTRIBUTE (
  preference_name IN VARCHAR2,
  attribute_name IN VARCHAR2,
  attribute_value IN VARCHAR2
);
```

preference_name

Specify the name of preference.

attribute_name

Specify the name of attribute.

attribute_value

Specify the attribute value. You can specify boolean values as `TRUE` or `FALSE`, `T` or `F`, `YES` or `NO`, `ON` or `OFF`, or `1` or `0`.

Examples

The following example sets values of two attributes of my_new_measure preference. The procedure attribute is set to dataminer.measures.my_new_measure_proc value and the normalize_procedure attribute is set to drifun.cosine_normalize value. These attributes means which database procedure and function are used for counting similarity measure using my_new_measure preference.

```
begin
  VECTX_DDL.SET_ATTRIBUTE (
     'my_new_measure',
     'procedure',
     'dataminer.measures.my_new_measure_proc');
  VECTX_DDL.SET_ATTRIBUTE (
     'my_new_measure',
     'normalize_procedure',
     'drifun.cosine_normalize');
end;
```

UNSET_ATTRIBUTE

Removes a set attribute from a preference.

Syntax

```
VECTX_DDL.UNSET_ATTRIBUTE (
  preference_name IN VARCHAR2,
  attribute_name IN VARCHAR2
);
```

preference_name

Specify the name of preference.

attribute_name

Specify the name of attribute.

Example

The following example removes value of normalize_procedure attribute from my_new_measure preference.

```
begin
  VECTX_DDL.UNSET_ATTRIBUTE (
     'my_new_measure',
     'normalize_procedure');
end;
```

SET_PRECOMPUTED_VALUE

This procedure saves precomputed value for given index and is designed for usage in user own similarity measure or token weight functions.

Syntax

```
VECTX_DDL.SET_PRECOMPUTED_VALUE (
  index_info IN vectxsys.VectorIndexInfo,
  name IN VARCHAR2,
  precomputed_value IN [[VARCHAR2],[BOOLEAN],[NUMBER]]
  query_id IN PLS_INTEGER DEFAULT NULL
);
```

index_info

Object containing all information about given index.

name

Name of precomputed value.

precomputed_value

Value to be saved.

query_id

Id of the query in case precomputed value should be saved only for given query duration.

Example

```
declare
  lint_docweight NUMBER;
  ind_info vectxsys.VectorIndexInfo;
begin
  VECTX_DDL.SET_PRECOMPUTED_VALUE (
     ind_info,
     'doc_weight',
     lint_docweight);
end;
```

UNSET_PRECOMPUTED_VALUE

This procedure deletes precomputed value for given index and is designed for usage in user own similarity measure or token weight functions.

Syntax

```
VECTX_DDL.UNSET_PRECOMPUTED_VALUE (
  index_info IN vectxsys.VectorIndexInfo,
  name IN VARCHAR2,
  query_id IN PLS_INTEGER DEFAULT NULL
);
```

index_info

Object containing all information about given index.

name

Name of precomputed value.

query_id

Id of the query in case precomputed value should be saved only for given query duration.

Example

```
declare
  ind_info vectxsys.VectorIndexInfo;
begin
  VECTX_DDL.UNSET_PRECOMPUTED_VALUE (
     ind_info,
     'doc_weight');
end;
```

GET_PRECOMPUTED_VALUE

This procedure returns saved precomputed value for given index and is designed for usage in user own similarity measure or token weight functions.

Syntax

```
VECTX_DDL.GET_PRECOMPUTED_VALUE (
  index_info IN vectxsys.VectorIndexInfo,
  name IN VARCHAR2,
  precomputed_value OUT [[VARCHAR2],[BOOLEAN],[NUMBER]]
  query_id IN PLS_INTEGER DEFAULT NULL
);
```

index_info

Object containing all information about given index.

name

Name of precomputed value.

precomputed_value

Resultant saved precomputed value.

query_id

Id of the query in case precomputed value should be saved only for given query duration.

Example

```
declare
  ind_info vectxsys.VectorIndexInfo;
  resultant_weight NUMBER;
begin
  VECTX_DDL.GET_PRECOMPUTED_VALUE (
     ind_info,
     'doc_weight',
     resultant_weight);
end;
```

A.3.3 VECTX_OUTPUT

START_LOG

Begin logging index and document service requests.

Syntax

```
VECTX_OUTPUT.START_LOG (
  logfile IN VARCHAR2
);
```

logfile

Specify the name of the log file. The log is stored in the directory specified by the system parameter `LOG_DIRECTORY`.

Example

The following example starts logging of Vector Text cartridge usage in current session into userlog01.log file (stored in directory specified by the system parameter `LOG_DIRECTORY`).

```
begin
  VECTX_OUTPUT.START_LOG ('userlog01.log');
end;
```

END_LOG

Halt logging index and document service requests.

Syntax

```
VECTX_OUTPUT.END_LOG;
```

Example

The following example ends logging of Vector Text cartridge usage in current session.

```
begin
  VECTX_OUTPUT.END_LOG;
end;
```

LOGFILENAME

Returns the filename for the current log. This procedure looks for the log file in the directory specified by the `LOG_DIRECTORY` system parameter.

Syntax

```
VECTX_OUTPUT.LOGFILENAME RETURN VARCHAR2;
```

Returns

Log file name.

Example

The following example returns the filename for current log. If logging is not started for current session then NULL is returned.

```
SELECT VECTX_OUTPUT.LOGFILENAME FROM dual;
```

A.3.4 VECTX_OUTPUT

START_LOG

Begin logging index and document service requests.

Syntax

```
VECTX_OUTPUT.START_LOG (
  logfile IN VARCHAR2
);
```

logfile

Specify the name of the log file. The log is stored in the directory specified by the system parameter `LOG_DIRECTORY`.

Example

The following example starts logging of Vector Text cartridge usage in current session into userlog01.log file (stored in directory specified by the system parameter `LOG_DIRECTORY`).

```
begin
  VECTX_OUTPUT.START_LOG ('userlog01.log');
end;
```

END_LOG

Halt logging index and document service requests.

Syntax

```
VECTX_OUTPUT.END_LOG;
```

Example

The following example ends logging of Vector Text cartridge usage in current session.

```
begin
  VECTX_OUTPUT.END_LOG;
end;
```

LOGFILENAME

Returns the filename for the current log. This procedure looks for the log file in the directory specified by the `LOG_DIRECTORY` system parameter.

Syntax

```
VECTX_OUTPUT.LOGFILENAME RETURN VARCHAR2;
```

Returns

Log file name.

Example

The following example returns the filename for current log. If logging is not started for current session then NULL is returned.

```
SELECT VECTX_OUTPUT.LOGFILENAME FROM dual;
```

A.3.5 VECTX_REPORT

DESCRIBE_INDEX

Creates a report describing the index. This includes the settings of the index meta-data, the indexing objects used and the settings of the attributes of the objects.

Syntax

```
procedure VECTX_REPORT.DESCRIBE_INDEX(
  index_name IN VARCHAR2,
  report     IN OUT NOCOPY CLOB
);

function VECTX_REPORT.DESCRIBE_INDEX(
  index_name IN VARCHAR2
) return CLOB;
```

index_name

The name of the index to be described.

report

The CLOB locator to which the report is written. It will be truncated before the report is generated.

Example

The following example fills into *rep* CLOB the index report for *new_idx* index.

```
DECLARE
  rep CLOB;
BEGIN
  rep := VECTX_REPORT.DESCRIBE_INDEX(
           'new_idx'
         );
END;
```

CREATE_INDEX_SCRIPT

Creates a SQL*Plus script which will create a text index that duplicates the named text index. The created script will include creation of preferences identical to those used in the named text index, only the names of the preferences will be different.

Syntax

```
procedure CTX_REPORT.CREATE_INDEX_SCRIPT(
  index_name      in varchar2,
  report          in out nocopy clob,
  prefname_prefix in varchar2 default null
);

function CTX_REPORT.CREATE_INDEX_SCRIPT(
  index_name      in varchar2,
  prefname_prefix in varchar2 default null
) return clob;
```

index_name

The name of the index for which the script is created.

report

The CLOB locator to which the script is written. It will be truncated before the script is generated.

prefname_prefix

Specify optional prefix to use for preference names. If not specified the index name is used.

Example

The following example fills into *rep* CLOB the index create script for *new_idx* index. The preference names are created with the prefix *pref*.

```
DECLARE
  rep CLOB;
BEGIN
  rep := VECTX_REPORT.CREATE_INDEX_SCRIPT(
           'new_idx',
           'pref'
         );
END;
```

Appendix B

Queries used in texts

The list of all queries used for tests are presented here. The results of the tests can be found in the Section 6.3. The individual table contains three columns - the first one contains the type of query (CTXT stands for query on context index, VECT stands for pre-defined query on vector index and VOPT stands for optimized query on vector index), the second one specifies the used similarity measure or token weight method and the last one contains the query specification being used.

Type	Parameter	Query specification
CTXT	-	'(COMPUTER ARCHITECTURES) or (ASSOCIATIVE PROCESSORS) or (ASSOCIATIVE STORES) or (ASSOCIATIVE or MEMORY)'
VECT	ALL METHODS	'COMPUTER ARCHITECTURES, ASSOCIATIVE PROCESSORS, ASSOCIATIVE STORES, ASSOCIATIVE MEMORY'
VOPT	TF_WEIGHT	'COMPUTER:0; ARCHITECTURE:2; ASSOCIATIVE:3; PROCESSOR:0; STORE:1; MEMORY:0'
VOPT	LOG_TF_WEIGHT	'COMPUTER:0; ARCHITECTURE:2; ASSOCIATIVE:2; PROCESSOR:0; STORE:1; MEMORY:0'
VOPT	NTF_WEIGHT	'COMPUTER:0; ARCHITECTURE:2; ASSOCIATIVE:2; PROCESSOR:0; STORE:1; MEMORY:0'
VOPT	TF_ITF_WEIGHT	'COMPUTER:0; ARCHITECTURE:2; ASSOCIATIVE:3; PROCESSOR:0; STORE:1; MEMORY:0'
VOPT	NTF_ITF_WEIGHT	'COMPUTER:0; ARCHITECTURE:2; ASSOCIATIVE:2; PROCESSOR:0; STORE:1; MEMORY:0'
VOPT	NORM_NTF_ITF_WEIGHT	'COMPUTER:0; ARCHITECTURE:1; ASSOCIATIVE:1; PROCESSOR:0; STORE:1; MEMORY:0'
VOPT	COSINE	'COMPUTER:0; ARCHITECTURE:2; ASSOCIATIVE:3; PROCESSOR:0; STORE:1; MEMORY:0'
VOPT	JACCARD	'COMPUTER:0; ARCHITECTURE:2; ASSOCIATIVE:3; PROCESSOR:0; STORE:1; MEMORY:0'
VOPT	DICE	'COMPUTER:0; ARCHITECTURE:2; ASSOCIATIVE:3; PROCESSOR:0; STORE:1; MEMORY:0'
VOPT	OVERLAP	'COMPUTER:0; ARCHITECTURE:2; ASSOCIATIVE:13; PROCESSOR:0; STORE:0; MEMORY:0'
VOPT	PSEUDOCOSINE	'COMPUTER:0; ARCHITECTURE:2; ASSOCIATIVE:2; PROCESSOR:0; STORE:1; MEMORY:0'
VOPT	ASYMMETRIC	'COMPUTER:0; ARCHITECTURE:1; ASSOCIATIVE:3; PROCESSOR:0; STORE:1; MEMORY:0'
VOPT	APPROXIMATED_COSINE	'COMPUTER:0; ARCHITECTURE:1; ASSOCIATIVE:3; PROCESSOR:0; STORE:1; MEMORY:0'

Table B.1: Query 1 - settings of CONTAINS operators for all test queries

Type	Method	Query specification
CONTEXT	-	'CHEMISTRY or CHEMICAL or PATENTS'
VECTOR	ALL METHODS	'CHEMISTRY, CHEMICAL, PATENTS'
OPTIM V	TF_WEIGHT	'CHEMISTRY:0; CHEMICAL:1; PATENTS:4'
OPTIM V	LOG_TF_WEIGHT	'CHEMISTRY:0; CHEMICAL:1; PATENTS:4'
OPTIM V	NTF_WEIGHT	'CHEMISTRY:0; CHEMICAL:2; PATENTS:2.5'
OPTIM V	TF_ITF_WEIGHT	'CHEMISTRY:0; CHEMICAL:1; PATENTS:5'
OPTIM V	NTF_ITF_WEIGHT	'CHEMISTRY:0; CHEMICAL:2; PATENTS:2.5'
OPTIM V	NORM_NTF_ITF_WEIGHT	'CHEMISTRY:0; CHEMICAL:3.1; PATENTS:5'
OPTIM V	COSINE	'CHEMISTRY:0; CHEMICAL:2; PATENTS:5'
OPTIM V	JACCARD	'CHEMISTRY:0; CHEMICAL:1; PATENTS:5'
OPTIM V	DICE	'CHEMISTRY:0; CHEMICAL:1; PATENTS:5'
OPTIM V	PSEUDOCOSINE	'CHEMISTRY:0; CHEMICAL:2; PATENTS:3'
OPTIM V	ASYMMETRIC	'CHEMISTRY:0; CHEMICAL:1; PATENTS:3'
OPTIM V	APPROXIMATED_COSINE	'CHEMISTRY:0; CHEMICAL:1; PATENTS:10'

Table B.2: Query 3 - settings of CONTAINS operators for all test queries

Type	Method	Query specification
VECTOR	ALL METHODS	'LIBRARY OUTREACH, BOOK DEPOSIT COLLECTIONS, ETHNIC MINORITIES, CHINESE, RACIAL GROUPS, ETHNIC GROUPS, LIBRARY SERVICES FOR MULTI-CULTURAL SOCIETY'
OPTIM V	TF_WEIGHT	'LIBRARY:2; OUTREACH:0; BOOKS:1; DEPOSIT:0; COLLECTIONS:1; ETHNIC:5; MINORITIES:4; CHINESE:4; RACIAL:0; GROUPS:1; SERVICES:2; MULTI-CULTURAL:0; SOCIETY:0'
OPTIM V	LOG_TF_WEIGHT	'LIBRARY:2; OUTREACH:0; BOOKS:1; DEPOSIT:0; COLLECTIONS:1; ETHNIC:5; MINORITIES:4; CHINESE:4; RACIAL:0; GROUPS:1; SERVICES:2; MULTI-CULTURAL:0; SOCIETY:0'
OPTIM V	NTF_WEIGHT	'LIBRARY:2; OUTREACH:0; BOOKS:1; DEPOSIT:0; COLLECTIONS:0; ETHNIC:6; MINORITIES:6; CHINESE:4; RACIAL:0; GROUPS:2; SERVICES:3; MULTI-CULTURAL:0; SOCIETY:0'
OPTIM V	TF_ITF_WEIGHT	'LIBRARY:2; OUTREACH:0; BOOKS:1; DEPOSIT:0; COLLECTIONS:1; ETHNIC:6; MINORITIES:6; CHINESE:4; RACIAL:0; GROUPS:1; SERVICES:2.5; MULTI-CULTURAL:0; SOCIETY:0'
OPTIM V	NTF_ITF_WEIGHT	'LIBRARY:2; OUTREACH:0; BOOKS:1; DEPOSIT:0; COLLECTIONS:0; ETHNIC:6; MINORITIES:6; CHINESE:4; RACIAL:0; GROUPS:2; SERVICES:3; MULTI-CULTURAL:0; SOCIETY:0'
OPTIM V	NORM_NTF_ITF_WEIGHT	'LIBRARY:3; OUTREACH:0; BOOKS:1; DEPOSIT:0; COLLECTIONS:0; ETHNIC:6; MINORITIES:6; CHINESE:4; RACIAL:0; GROUPS:2; SERVICES:4; MULTI-CULTURAL:0; SOCIETY:0'

Table B.3: Query 6 - settings of CONTAINS operators for the test queries using the different token weight methods

Type	Method	Query specification
CONTEXT	-	'(LIBRARY OUTREACH) or (BOOK DEPOSIT COLLECTIONS) or (ETHNIC MINORITIES) or (CHINESE) or (RACIAL GROUPS) or (ETHNIC GROUPS) or (LIBRARY SERVICES FOR MULTI-CULTURAL SOCIETY)'
OPTIM V	COSINE	'LIBRARY:2; OUTREACH:0; BOOKS:1; DEPOSIT:0; COLLECTIONS:0; ETHNIC:6; MINORITIES:6; CHINESE:6; RACIAL:0; GROUPS:1; SERVICES:2; MULTI-CULTURAL:0; SOCIETY:0'
OPTIM V	JACCARD	'LIBRARY:2.5; OUTREACH:0; BOOKS:1; DEPOSIT:0; COLLECTIONS:0; ETHNIC:6; MINORITIES:6; CHINESE:6; RACIAL:0; GROUPS:1; SERVICES:3.5; MULTI-CULTURAL:0; SOCIETY:0'
OPTIM V	DICE	'LIBRARY:2.5; OUTREACH:0; BOOKS:1; DEPOSIT:0; COLLECTIONS:0; ETHNIC:6; MINORITIES:6; CHINESE:6; RACIAL:0; GROUPS:1; SERVICES:3.5; MULTI-CULTURAL:0; SOCIETY:0'
OPTIM V	PSEUDOCOSINE	'LIBRARY:2.5; OUTREACH:0; BOOKS:1; DEPOSIT:0; COLLECTIONS:0; ETHNIC:6; MINORITIES:6; CHINESE:6; RACIAL:0; GROUPS:1; SERVICES:3.5; MULTI-CULTURAL:0; SOCIETY:0'
OPTIM V	ASYMMETRIC	'LIBRARY:2.5; OUTREACH:0; BOOKS:1; DEPOSIT:0; COLLECTIONS:0; ETHNIC:6; MINORITIES:6; CHINESE:6; RACIAL:0; GROUPS:1; SERVICES:3.5; MULTI-CULTURAL:0; SOCIETY:0'
OPTIM V	APPROXIMATED_COSINE	'LIBRARY:2.5; OUTREACH:0; BOOKS:1; DEPOSIT:0; COLLECTIONS:0; ETHNIC:6; MINORITIES:6; CHINESE:6; RACIAL:0; GROUPS:1; SERVICES:3.5; MULTI-CULTURAL:0; SOCIETY:0'

Table B.4: Query 6 - settings of CONTAINS operators for the test queries using the different similarity measure methods

Type	Method	Query specification
CONTEXT	-	'SDI or (SELECTIVE DISSEMINATION OF INFORMATION) or (CURRENT AWARENESS BULLETINS) or (INFORMATION BULLETINS)'
VECTOR	ALL METHODS	'SDI, SELECTIVE DISSEMINATION OF INFORMATION, CURRENT AWARENESS BULLETINS, INFORMATION BULLETINS'
OPTIM V	TF_WEIGHT	'SDI:10; SELECTIVE:2; DISSEMINATION:1; INFORMATION:1; CURRENT:1.2; AWARENESS:3; BULLETINS:3.5'
OPTIM V	LOG_TF_WEIGHT	'SDI:3; SELECTIVE:1; DISSEMINATION:1; INFORMATION:2; CURRENT:1; AWARENESS:1; BULLETINS:2'
OPTIM V	NTF_WEIGHT	'SDI:10; SELECTIVE:2; DISSEMINATION:1; INFORMATION:1.2; CURRENT:1.2; AWARENESS:4.5; BULLETINS:4.5'
OPTIM V	TF_ITF_WEIGHT	'SDI:10; SELECTIVE:2; DISSEMINATION:1; INFORMATION:1; CURRENT:1.2; AWARENESS:3; BULLETINS:3.5'
OPTIM V	NTF_ITF_WEIGHT	'SDI:10; SELECTIVE:2; DISSEMINATION:1; INFORMATION:1.2; CURRENT:1.2; AWARENESS:4.5; BULLETINS:4.5'
OPTIM V	NORM_NTF_ITF_WEIGHT	'SDI:12; SELECTIVE:2; DISSEMINATION:1; INFORMATION:1.2; CURRENT:1.2; AWARENESS:4.5; BULLETINS:4.5'
OPTIM V	COSINE	'SDI:10; SELECTIVE:2; DISSEMINATION:1; INFORMATION:1; CURRENT:1.3; AWARENESS:3.5; BULLETINS:3.5'
OPTIM V	JACCARD	'SDI:12; SELECTIVE:2; DISSEMINATION:1; INFORMATION:1; CURRENT:1.3; AWARENESS:3.5; BULLETINS:3.5'
OPTIM V	DICE	'SDI:12; SELECTIVE:2; DISSEMINATION:1; INFORMATION:1; CURRENT:1.3; AWARENESS:3.5; BULLETINS:3.5'
OPTIM V	PSEUDOCOSINE	'SDI:12; SELECTIVE:2; DISSEMINATION:1; INFORMATION:1; CURRENT:1.3; AWARENESS:3.5; BULLETINS:3.5'
OPTIM V	ASYMMETRIC	'SDI:12; SELECTIVE:2; DISSEMINATION:1; INFORMATION:1; CURRENT:1.3; AWARENESS:3.5; BULLETINS:3.5'
OPTIM V	APPROXIMATED_COSINE	'SDI:10; SELECTIVE:2; DISSEMINATION:1; INFORMATION:1.2;CURRENT:1; AWARENESS:3.5; BULLETINS:3.5'

Table B.5: Query 10 - settings of CONTAINS operators for all test queries

Type	Method	Query specification
CONTEXT	-	'(CHEMICAL STRUCTURES) or (COMPUTERIZED DATABASES) or CHEMISTRY or CHEMICAL or SUBSTRUCTURE'
VECTOR	ALL METHODS	'CHEMICAL STRUCTURES, COMPUTERIZED DATABASES, CHEMISTRY, CHEMICAL, SUBSTRUCTURE'
OPTIM V	TF_WEIGHT	'CHEMICAL:3; STRUCTURES:4; COMPUTERIZED:0; DATABASES:0; CHEMISTRY:1; SUBSTRUCTURE:10'
OPTIM V	LOG_TF_WEIGHT	'CHEMICAL:3; STRUCTURES:4; COMPUTERIZED:0; DATABASES:0; CHEMISTRY:1; SUBSTRUCTURE:10'
OPTIM V	NTF_WEIGHT	'CHEMICAL:3; STRUCTURES:4; COMPUTERIZED:0; DATABASES:0; CHEMISTRY:1; SUBSTRUCTURE:10'
OPTIM V	TF_ITF_WEIGHT	'CHEMICAL:3; STRUCTURES:4; COMPUTERIZED:0; DATABASES:0; CHEMISTRY:1; SUBSTRUCTURE:10'
OPTIM V	NTF_ITF_WEIGHT	'CHEMICAL:3; STRUCTURES:4; COMPUTERIZED:0; DATABASES:0; CHEMISTRY:1; SUBSTRUCTURE:10'
OPTIM V	NORM_NTF_ITF_WEIGHT	'CHEMICAL:3; STRUCTURES:4; COMPUTERIZED:0; DATABASES:0; CHEMISTRY:1; SUBSTRUCTURE:10'
OPTIM V	COSINE	'CHEMICAL:3; STRUCTURES:4; COMPUTERIZED:0; DATABASES:0; CHEMISTRY:1; SUBSTRUCTURE:10'
OPTIM V	JACCARD	'CHEMICAL:3; STRUCTURES:4; COMPUTERIZED:0; DATABASES:0; CHEMISTRY:1; SUBSTRUCTURE:10'
OPTIM V	DICE	'CHEMICAL:3; STRUCTURES:4; COMPUTERIZED:0; DATABASES:0; CHEMISTRY:1; SUBSTRUCTURE:10'
OPTIM V	PSEUDOCOSINE	'CHEMICAL:3; STRUCTURES:4; COMPUTERIZED:0; DATABASES:0; CHEMISTRY:1; SUBSTRUCTURE:10'
OPTIM V	ASYMMETRIC	'CHEMICAL:3; STRUCTURES:4; COMPUTERIZED:0; DATABASES:0; CHEMISTRY:1; SUBSTRUCTURE:10'
OPTIM V	APPROXIMATED_COSINE	'CHEMICAL:4; STRUCTURES:5; COMPUTERIZED:0; DATABASES:0; CHEMISTRY:1; SUBSTRUCTURE:15'

Table B.6: Query 15 - settings of CONTAINS operators for all test queries

Enclosed CD

The enclosed CD contains the following data:

doc/	programming documentation in the HTML format generated by RoboDoc (http://sourceforge.net/projects/robodoc) from the sources of the Vector Text cartridge
test/	LISA collection with pre-defined queries and lists of relevant documents
thesis/	LaTeX sources of this thesis including used pictures
thesis.ps	this thesis in PostScript format
thesis.pdf	this thesis in PDF format
VectorText/	installation scripts and sources of the Vector Text cartridge