Sub-element Indexing and Probabilistic Retrieval in the POSTGRES Database System

Anne Fontaine

May 23, 1995

1 Introduction

Most current information retrieval systems use boolean search methods to request and retrieve documents. While effective for precise query specifications, boolean search systems suffer when the request is of a more general nature such as in a subject search request. If the query is too precise, the search may fail with no documents returned, however, if the query is too general, too many documents may be returned with only a few actually relevant. Moreover, the documents are not returned in an order to make them easy to evaluate. More advanced retrieval systems, such as probabilistic retrieval, are being explored to address some of these problems.

Probabilistic retrieval (PR) is an approach to information retrieval that attempts to estimate the probability that a particular document is relevant to the user's information need as expressed in a natural language query. Documents are returned to the user ranked in descending order of this probability of relevance. The relevance of a document is determined by an algorithm that considers the terms of the query, the terms contained in the documents and how the terms are distributed within the database collection to produce a rank value. The documents are then ordered according to this rank value with the "correct" documents at the top of the list.

The query is entered as a natural language statement that is then scanned for important search terms. All documents with a term in common with the query (with common words filtered out) are returned to the ranking module which determines the order in which the documents are returned. The natural language statement provides an easy interface for the user to request information.

Our ranking strategy requires that frequency counts be calculated for keywords of a document and for keywords throughout the collection. We developed an indexing mechanism based on sub-element indexing to support this type of retrieval. Sub-element indexing provides the ability to create indexes on components extracted from both builtin and user-defined abstract data types. In conventional indexes the entire contents of a given column are treated as a set of single elemental objects, one per row. This type of indexing is not appropriate when the row/column values are themselves complex objects made up of a collection of many "sub-elements". Examples include text documents, arrays of other datatypes, maps, images, structured drawings, etc., where the user is interested, or able to specify in a query, only some part of the information contained in the object.

The goal of this implementation is to create an environment in which to study various indexing and ranking strategies. In particular, the ideas of Lynch[8] and Cooper[3] are incorporated into the system. The POSTGRES database system was used because it provides a flexible environment in which to experiment with various access methods, Functions, access methods and datatypes can be defined by the user and added more easily than with most relational database systems. POSTGRES also provides support for large object types that can be used to access full-text documents.

The next section summarizes the previous work that has influenced the implementation described in this paper. The following sections describe the implementation of the indexing and retrieval modules, and the retrieval experiments that were performed. Following a discussion of the results obtained from this system is a description of the implementation issues specific to the POSTGRES database system.

In this paper, the words *keyword*, *term*, and *token* are often used interchangeably, as are the words *record* and *tuple*.

2 Related Work

Extensive work has been done in the area of indexing and probabilistic retrieval, however, only the material that is specifically related to this implementation is discussed here.

Lynch and Stonebraker propose a method of extended indexing to support abstract data types and user-defined operators [8]. This scheme allows indexes to be created from a computed value of a column. The new *List* datatype is used to accomplish this extension. Their proposal provides much of the foundation to the implementation discussed in this paper.

Aoki implemented an approach to extended indexing along the ideas of Lynch and Stonebraker with only slight changes [1]. Some of Aoki's work was restored from an earlier version of POSTGRES, though most was modified for this implementation. Aoki created an fbtree (functional B-tree) access method, similar to a regular B-tree, that supports multiple index keys from one column value. This code was recovered and modified to accommodate probabilistic retrieval. Other changes made to the extended indexing code include changes to the postquel interface, changes to the extract routine, and support for external document file types.

Cooper, Gey, and Chen experimented with a probabilistic retrieval method using staged logistic regression[3]. The equations that they derived are incorporated here and form the basis of the ranking module of this implementation.

Larson created the *Lassen* system, an indexing and retrieval system that interfaces with the POSTGRES database systems [6]. Lassen uses the POSTGRES rule system to trigger routines that extract and index terms when a document is added to the database. The interface retrieves and then ranks the documents returned by the database system. Much of the indexing and retrieval functions of the *Lassen* system is written as a front-end database application. Because of this, it uses more storage space and requires a few more processing steps than if it were incorporated directly into the database system. In describing the *Lassen* system, Larson also gives suggestions on how these functions might be included in the POSTGRES backend.

This implementation was designed specifically to incorporate the indexing and ranking functions directly into the POSTGRES database system. The POSTGRES database system was used because it provides a flexible environment in which to experiment with various access methods, abstract datatypes, and user-defined functions. It also provides an environment where the ideas of Lynch[8] and Cooper[3] can be evaluated.

3 POSTGRES

POSTGRES is a relational database system designed to allow easy extensibility [9]. Users can create their own abstract data types, operators, functions and access methods [10] [11]. In this adaptation, a new access method was added to accommodate the new indexing strategy. Although the functions to extract keywords from documents and to rank the documents are built directly into the system, we designed them so that user can also define their own functions and register them with the database.

POSTGRES also supports a wide variety of abstract data types. It has several different implementations of large objects all of which are accessed via typical file operations [12]. Although large objects can include photographs, video streams, and documents, we focus on full-text documents and use the External large object type in this implementation. The External type stores the full path name of the external file in the large object column and does not copy the file into the database system. There is, therefore, no transaction processing and crash recovery support when using the External large object. Because of this, there is some security risk in using this file type.

Extensive changes to the system were made to complete this implementation. Not only was a new access method added, but the parser, optimizer, executor, and system catalogs were modified. With this framework in place, we can use the extensibility feature to experiment with alternative ranking and tokenizing routines.

4 Sub-element Indexing

Most relational indexing methods produce one index value from a column value, which is usually the column value itself. For bibliographic data, however, this value is often a filename or a document identifier which is generally not a useful index key.

To index on a value other that the actual column value, POSTGRES implemented the functional index. The functional index runs a function over the column value to produce another value that is then indexed. Although the index value has changed, there is still only one index value per column.

This implementation combines the idea of the functional index with the ability to index more than one value per column into the indexing module itself. The indexing model has been extended to recognize the type of object that it has been given and index based on that type. When a document type is encountered, the indexing module extracts the keywords from the document, counts the number of occurrences within the document and inserts an index record on each of the unique keywords.

There are three steps to the algorithm for indexing a document:

- Tokenize Keywords
- Index Keywords
- Update Meta Data

Each step is further described in the sections below.

4.1 Tokenize Keywords

The tokenize process breaks up a string value into individual tokens that are then candidates for index keys. A default tokenize routine is provided in the implementation, but the user can also define a more specialized routine and register it with the POSTGRES database.

The default routine extracts alphabetic characters and converts them to lower-case to allow for case-insensitive searching. Hyphenated words are split into separate tokens.

Abbreviations are not expanded. Alternate spellings are not considered and numbers are not extracted. It is a simple routine that considers the end of the term when it encounters any non-alphabetic character. After the token is extracted, it is checked against a list of stopwords and passed through a stemming process. Each of these processes are further described in more detail below.

4.1.1 Stoplist Tokens

Stopwords are common words that are insignificant as index keys. They include prepositions, articles, possessives, some passive verbs and some adverbs. Because they occur in many documents, their discriminatory value is low and they are not useful as index terms.

Stopwords can account for a large fraction of a document's terms. Removing these words as index candidates, can reduce the amount of storage space needed and the amount of retrieval time required. Storage space requirements will be less because fewer terms will be indexed making the size of the index file smaller. Retrieval time will be shorter because fewer terms will match and therefore fewer irrelevant documents will be returned.

Our list of stopwords is maintained in the system table pg_stoplist and currently contains 417 words. Words can be added and deleted to the system table, thereby customizing the stoplist for the documents of a particular database.

The stoplist is loaded into the system table when the database is built. The first time the stoplist routine is accessed, the entire list of stopwords is loaded into an in-memory hash table. This hash table reduces the time needed to stoplist a document and is available for subsequent document insertions.

4.1.2 Stem Tokens

Stemming is the process of finding morphological variants of a term, often based on finding a common root term. The stemming routine used here is an implementation of the Porter stemming algorithm that removes suffixes of a term when certain conditions are met [5]. For example, it looks for a plural ending and when one is found removes it from the end of the word.

Stemming combines common root words of a document into one index record. The frequency count of this index record is the number of occurrences of all the variants of this common root word. The retrieval process is then affected by how the terms are stemmed. Not only are common root terms retrieved, but the ordering of documents is also affected because the ranking equation considers the frequency count associated with the root terms. Given a request *find documents about distributed networks*, the

Word	Tolower	Stop	Stem	Keyword	Freq
Implementation	implementation	implementation	implement	implem	1
of	of	–	–		
Extended	extended	extended	extend	extend	1
Indexing	indexing	indexing	index	index	1
and	and	–	–		
Probabilistic	probabilistic	probabilistic	probabilist	probabilist	1
Retrieval	retrieval	retrieval	retriev	retriev	1
in	in	–	–		
POSTGRES	postgres	postgres	postgr	postgr	1

Table 1: Example: Tokenize Process

term *networks* will be stemmed to *network*. Documents with the words *network*, *networks*, and *networking* will be considered and retrieved.

Stemming also has an affect on the size of the index file. Combining the common root terms produces fewer keyword/frequency index records and reduces the size of the file. In this implementation the stemmed word is stored in the index and the unstemmed word is available only in the document itself.

A term can be stemmed when it is inserted or when it is retrieved. This implementation stems a document term before it is inserted as an index. It also stems the query string when retrieving documents. Relevance ranking is done on the stemmed form of the word, so the search terms must be stemmed to match the indexed document terms. The processing overhead for stemming terms then occurs primarily at document insertion time.

Finally, consider the phrase *Implementation of Extended Indexed in POSTGRES*. Each of the steps to tokenize this phrase are shown in Table 1. The routine first scans the phrase into words and converts them to lower case. The Stop column shows that the words *of*, *and*, and *in* are stoplisted and dropped from the process. The fourth column shows the results of the stemming process. Notice that POSTGRES has been stemmed to the term "postgr". The stemmer first evaluated the final 's' as a plural ending and removed it. Then the stemmer tried to find the root of the word "postgre" and converted this to "postgr". The last two columns show the resulting six keyword/frequency pairs.

Stemming tends to increase the number of documents returned for a query because not only the search term itself but other related terms are retrieved. Although we have not studied the affect that stemming has on the precision and recall, we assume that stemming will have a positive impact on retrieval effectiveness.

4.2 Index Keywords

A new access method, the fbtree, was created to handle the indexing. The fbtree is very similar to the regular btree access method with the insert and the search routines customized to support the new indexing. The fbtree was first developed by Aoki [1] and modified here to incorporate probabilistic retrieval.

The insert routine breaks up the document into "indexable terms" - terms that have successfully passed the tokenize routine, and inserts one index record for each unique term in a document. The index records consists of the index term and the frequency count, the number of times the term occurs in the document. The insert routine is also responsible for collecting information about the indexed terms and documents. This data is later used at retrieval time to calculate the rank value of the document to a query.

In the normal index retrieval process, the index record contains the index key and a reference to the "real" record. Once the "real" record is determined and retrieved, the index tuple is no longer needed. In our implementation, however, the index tuple contains the keyword/frequency values that are essential to the ranking module. The search routine was modified to return the keyword/frequency pair as well as the document record itself.

4.3 Update Meta Data

System catalogs are maintained to accumulate information about the number of terms and documents in the database. The tables are updated as documents are inserted into the collection. The data is used at retrieval time to calculate the relative relevance of the document to the query string (see Section 5.3).

There are three categories of meta data:

- Global - for the entire database
  - total number of documents in the database
  - total number of terms in the database
  - total number of unique terms in the database
- Document - for each document
  - number of terms in document
  - number of unique terms in document
- Term - for each term

- number of references
- number of documents that reference term

There is one Global record for the entire database collection, one Document record for each document in the database, and one Term record for each term in the database. The Document and Term tables are indexed for quick retrieval. The Document table is indexed on document name and the Term table is indexed on term name.

5 Probabilistic Retrieval

Probabilistic retrieval attempts to estimate the probability that a particular document is relevant to a user's search request. A search request is entered as a natural language statement. Documents that contain any of the stemmed terms in the query (excluding stoplisted words) are retrieved and given a relevance ranking value. The documents are returned to the user ranked in descending order of this probability of relevance. The relevance value is based on the terms in the search request, in the documents retrieved, and in the database collection as a whole.

There are five steps to the ranked retrieval module:

- Tokenize Query String
- Indexed Retrieval of Tuples
- Calculate Rank Value
- Sort Tuples
- Return Tuples

5.1 Query String

The syntax of a probabilistic query is more flexible and less complicated than the syntax used in boolean searches. Consider the query *what similarity laws must be obeyed when constructing aeroelastic models of heated high speed aircraft.*

A naive query like:

```
similarity AND (laws OR law) AND (construct OR constructing) AND
    (aeroelastic AND (model OR models) AND (heat OR heated) AND high
     AND speed AND aircraft
```

could be constructed, but a document that talked about "supersonic" aircraft and did not use "high speed" would be excluded, ie – any missing "ANDed" term causes the entire match to fail. Changing ANDs to ORs would mean that most of the database would be retrieved, with no indication of better or worse matches.

By contrast, the probabilistic query is entered as it appears in italics above. As a natural language statement, the format provides an easier interface to enter searches than does the Boolean format. The query string is processed as a document - keywords are extracted, stop-listed, stemmed and counted - and the resulting search terms are connected with implied ORs. Although many documents may satisfy the query, a rank value is calculated on each document to order them in order of probable relevance.

The syntax also allows easier extension of search strings. For example, although the current system does not automatically search for synonyms, it is easy for the searcher to add them to the query string. Also, to emphasize the importance of a term, the user can enter the term multiple times, increasing the weight given to the query term.

5.2 Indexed Retrieval

Each unique keyword of the query string is used as a key value upon which to retrieve documents. Any document that has at least one stem in common with the query string is retrieved and passed to the ranking module. The ranking module determines whether the document is presented to the user and in what order. The basis of this decision depends on the minimum rank weight specified in the query and the relevance of the document to other documents retrieved.

5.3 Calculate Rank Value

The ranking routine determines the probable relevance of a document to a given query. The goal of the ranking is to derive an initial ordering of documents to be presented to the user in descending order of relevance. The current ranking scheme uses a logistic regression method developed by Cooper [2]. The equation calculates the logodds of the relevance of the document to the query.

$$\log O(R|Q,D) \approx c_0 + \sum_{i=1}^{6} c_i X_i$$

where R is the event that the document D is relevant to the query Q and

$X_1 = \frac{1}{M}\sum_{j=1}^{M}\log(QAF_{t_j})$	the mean of the logged absolute frequency of matching stemmed terms in the query.
$X_2 = \sqrt{QL}$	the square root of the number of stemmed terms in the query (with stoplist words removed).
$X_3 = \frac{1}{M}\sum_{j=1}^{M}\log(DAF_{t_j})$	the mean of the logged absolute frequency of the matching stemmed terms in the document.
$X_4 = \sqrt{DL}$	the square root of the number of stemmed terms in the document (with stoplist words removed).
$X_5 = \frac{1}{M}\sum_{j=1}^{M}\log(IDF_{t_j})$	the mean logged inverse document frequency which is the ratio of the number of total documents in the collection to the number of documents with the matching term.
$X_6 = \log M$	the log of the number of matching stemmed terms.

The coefficients derived from the study are

$$c_0 = -3.70, c_1 = 1.269, c_2 = -0.310, c_3 = 0.679, c_4 = -0.0674, c_5 = 0.223, c_6 = 2.01.$$

As is shown above, the equation uses several factors in judging the relevance of a document to a query. It not only considers the frequency of matching terms between the document and the query, it also considers the importance of the search term in the query, the length of the document, and the number of different terms matched. The importance of a search term in a query can be estimated by counting the number of times the term is used in the query, by considering the number of terms in the query itself and by determining the relative frequency of the term in the collection of documents. The length of the document is taken into account to offset some of the advantage that a longer document has over a shorter document. Longer documents will tend to have more matches than a shorter ones simply because they have more terms. The equation gives an added weight to documents with more different matching terms. This reduces the event that a document will be judged relevant based on a few very frequently used matched terms. Together these factors provide a reasonable judgment of relevance as is shown in the experiments that follow.

6 Experiments

Retrieval experiments were run on the Cranfield collection set of 1400 documents and 225 queries. The documents are abstracts of aerodynamic documents and the queries

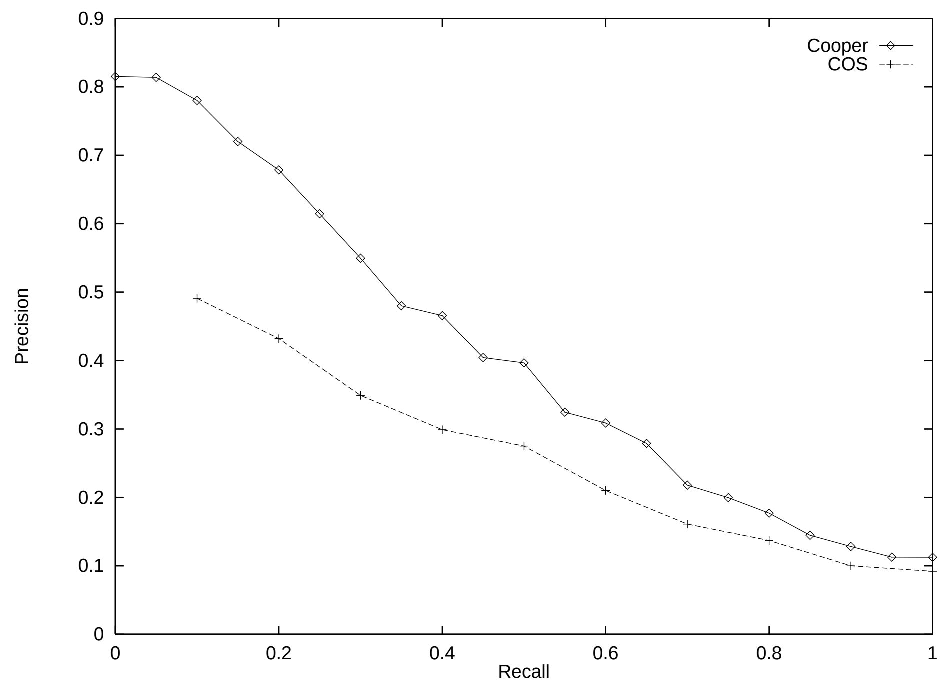

Figure 1: Average precision and recall.

are natural language queries on this information. Query one, for example, asks *what similarity laws must be obeyed when constructing aeroelastic models of heated high speed aircraft*. Included in the collection is a relevance file that lists the relevant documents that should be returned for each of the queries.

The results of the queries were evaluated by calculating the precision, recall, and E measure values which are further described below. As a basis of comparison, the results are shown along with the results from a method using Cosine Correlation(COS), the values of which are given in an analysis of probabilistic models by Croft and Harper[4].

The recall and precision values are measured using the following equations.

$$\text{Recall} = \frac{\text{number of relevant documents retrieved}}{\text{total number of relevant documents for that query}}$$

$$\text{Precision} = \frac{\text{number of relevant documents retrieved}}{\text{total number of documents retrieved}}$$

The recall and precision values are shown in Figure 1. The results show that the Cooper retrieval method performs much better than the COS method at the top section of the return set. Moreover, the precision values are higher in the Cooper method at all levels of recall.

The second experiment calculates the E measure which is a weighted combination of

	E After 10 Docs			E After 20 Docs			E After 30 Docs		
	0.5	1.0	2.0	0.5	1.0	2.0	0.5	1.0	2.0
Cooper	0.7053	0.6833	0.6387	0.7864	0.7359	0.6366	0.8333	0.7803	0.6640
COS	0.79	0.77	0.73	0.84	0.80	0.72			

Table 2: Average E values.

	After 10 Docs	After 20 Docs	After 30 Docs
Precision	0.2884	0.1913	0.1444
Recall	0.4196	0.5323	0.5865

Table 3: Precision and Recall Values for Cooper Method.

recall and precision with lower E values representing more effective retrieval. The E measure is calculated as

$$E = 1 - \frac{1}{\alpha\left[\frac{1}{\text{Precision}}\right] + (1-\alpha)\frac{1}{\text{Recall}}}$$

where

$$\alpha = \frac{1}{\beta^2 + 1}$$

The parameter β is used to evaluate relative effectiveness in terms of recall and precision. A β value of 1.0 gives recall and precision equal importance. A β value equal to 0.5 gives twice as much importance to precision and $\beta = 2.0$ gives twice as much importance to recall.

Because the E value is calculated on a result set, the effectiveness of ranked retrieval can be measured by calculating the E measure at different cutoff points within the result set. Table 2 shows the E values at two cutoff points - 10 documents retrieved and 30 documents retrieved and at three β values - 0.5, 1.0, and 2.0.

The lowest E values, indicating the most effective retrieval, are 0.6366 and 0.6387. These values are with 20 documents and 10 documents returned, respectively, and with recall weighted more heavily than precision. The corresponding value after 30 documents, 0.6640, however, shows a sudden drop in effectiveness. Table 3 further shows that the precision values fall at a greater rate than the recall values rise.

7 Implementation in POSTGRES

The following sections describe specific changes made to three principal modules of the POSTGRES database system:

Parser: parses the command and outputs either a query tree or query list.

Optimizer/Planner: finds all execution plans, calculates the cost of each plan, and selects the cheapest.

Executor: executes the cheapest plan.

7.1 Parser

The parser accepts a command from the user and produces a parsetree that is used by the optimizer/planner in determining an execution plan. The *define index* and the *retrieve* commands were modified to accommodate the changes for this implementation. In the following explanation of the commands, an upper-case word is a Postquel keyword which is a keyword defined by the POSTGRES query language. A lower-case word defines a variable name. In most cases, the name attempts to describe the type of object that it represents. Items in brackets [ ] are optional items. Items in braces { } can be repeated.

7.1.1 Define index

The *define index* statement was changed to include the extract function used in the tokenization process. This function can be the default extract function *kwlist_extract* or it can be a function defined by the user. A user-defined function must be defined and registered with the POSTGRES database (see the POSTGRES user manual for more information about defining functions).

```
DEFINE [ARCHIVE] INDEX index_name
               ON classname USING am_name
               ( attname type_class )
               [WHERE qual]
       --->    [WITH (EXTRACT = extract_func)]
```

index_name is the name of the index assigned by the user. The name is needed to create and remove and index.

classname is the name of the table that contains the attribute that will be indexed.

am_name is the access method that will be used to index the keys. We use the 'fbtree' access method for ranked retrieval.

attname is the name of the attribute that is being indexed.

type_class is the name of the operator class that this attribute uses. The text operator class 'text_ops' in this implementation.

qual is a qualification clause that can restrict when the column will be indexed.

EXTRACT is the Postquel keyword to indicate the function is to used as the extract function.

extract_func is the name of the extract function to be used with this index. Again this is either *kwlist_extract* or a user-defined function.

```
Example:   define index doc_ind on docs using fbtree (doc text_ops)
                   with (extract = kwlist_extract)
```

7.1.2 Retrieve

The *retrieve* command was changed to include the rank clause. This specifies which rank function to use and what query string to evaluate. It also allows the user to specify a threshold value

```
RETRIEVE
        [(INTO classname [ archive_mode] |
            PORTAL portal_name |
            IPORTAL portal_name) ]
            [UNIQUE]
            ([attr_name-1 =] expression-1 {,[attr_name-i =] expression-i})
            [FROM from_list]
            [WHERE
      --->  [RANK [rank_function](attr_name = query_string [, minWt ])]
              [qual]]
            [SORT BY attr_name-1 [ USING operator ]
            {, attr_name-j [ USING operator ] } ]
```

RANK is the Postquel keyword used to introduce the rank clause

rank_function is the name of the rank function to use. This is an optional field. If no value is given the default rank function is used.

attr_name is the attribute name that has an fbtree index.

query_string is a string of words on which to search.

minWt specifies a minimum ranking weight. This is an optional field used to specify a threshold value. Only documents with a rank value greater than or equal to this value are returned to the user. If not minimum weight is given, the default value is zero.

```
Example:  retrieve (docs.all) where rank(docs.doc = ''digital library'',1)
```

7.2 Optimizer/Planner

The optimizer/planner determines how the query will be executed. It finds all the possible execution plans for the particular query and determines which one will be used. In determining the execution plan, the optimizer/planner considers the methods of scanning through the data - sequential scans, index scans - and methods of joining tables - hash join, merge join, nested loop. The optimizer then calculates the cost of each plan and selects the least costly of them.

When a ranked retrieval is requested, the optimizer should select the index scan of the fbtree index to retrieve the keyword/frequency pairs that have been stored in the index. The optimizer was changed to recognize that the rank clause contains an attribute that has an fbtree index and to create a plan based on that index scan. This plan will generally be the least costly one available because the keyword/frequency pairs have already been extracted and calculated. Because of this, the cost of the plan was determined to be zero.

The implementation does not currently account for execution plans other than the one using the index scan. To consider other plans, the executor would need to be modified to read each of the documents as they are retrieved in order to calculate the keyword/frequency pair. The optimizer would also need to be changed to consider a more accurate cost selectivity algorithm than the one being used.

Upon creating the execution plan, the optimizer creates a "rank" node and attaches it at the top level of the plan. This node will control the overall execution of the plan and will perform functions necessary for ranked retrieval. This node is part of the executor and is described further in the next section.

7.3 Executor

The executor carries out the plan that was selected by the optimizer. As mentioned above in section 7.2, the optimizer creates a "rank" node at the top level and somewhere below it is and index scan on the fbtree index. The rank node is very similar to POSTGRES's sort node. That is, both of them retrieve tuples (or records) and store them in a temporary relation. The tuples are then sorted and returned to the user in sorted order.

The rank node uses a temporary hash table to collect the tuples and keyword/frequency pairs. Because index keys will often point to the same tuple, the hash table is used to identify the duplicates and merge the keyword/frequency values. After the tuples are collected each of them is ranked and stored in a temporary relation that is then sorted by rank value.

The following is a outline of the functions performed by the rank node:

1. Tokenize the query string
2. Retrieve tuples
   (a) Set up the next index scan
   (b) Retrieve the tuples from the underlying nodes
   (c) Collect the tuples in a hash table (to identify duplicate tuple records)
3. Calculate Rank
   (a) Read the hash table
   (b) Calculate the relevance of the document to the query
   (c) Store in a temporary relation
4. Sort the temporary relation by rank value.
5. Retrieve tuples from the sorted relation.

8 Storage Space Issues

The method of storage of a fbtree access method is essentially that of a btree. The fbtree access method extracts objects from a larger object and inserts them into a btree. (It also reads objects from the btree and gathers them together.) For each unique keyword in a document, there is one record in the index btree. The index record contains the term and the frequency count of the term within the document.

Other storage requirements are needed to maintain the information used to calculate the relevance ranking. This information includes:

- one record to collect summary information for the entire database
- one record for each document in the database
- one record for each unique keyword in the entire database

The object that we index can be a *text* or an External large object type (*LargeObj*) data type. The *text* type is a regular string and is stored directly into the database. The External large object is used to represent documents that reside outside of the database system. External documents are not copied into the database and do benefit from the transaction management and concurrency control operations provided within a database environment. There are some advantages to using external documents, however. The file is not copied which saves time and may save storage if the file would otherwise be stored in two locations.

Stemming and stoplisting words also reduces storage space requirements. Similar root terms are combined with stemming to reduce the amount of space used for index files. Stoplisting reduces the size by removing the most common words from consideration.

One other storage issue involves main memory usage. An in-memory hash table was used in three situations - to count the number of occurrences of a token within a document or string (used for both document indexing and query string processing); to recognize a duplicate tuple when retrieving; and to store the list of stop words of the system table to reduce the amount of time needed to stoplist terms. There is generally no problem in using the main memory hash table, however, it is possible, although unlikely, that the hash table will grow too large.

9 Other Implementation Issues

The current implementation uses a form of a btree index. There is one entry in the index for each unique keyword of each document. An alternative method would be to use a "doclist" approach that would have one entry per unique keyword in the entire collection. The index would then point to a list of documents that contain the particular keyword.

The optimizer/planner currently considers only the plan with the fbtree index scan. Although this will usually be the most effective plan, the optimizer and executor could be changed to consider other plans. When a plan other than the fbtree index plan is selected, the rank node must read the contents of each document and calculate the keyword/frequency pairs. The optimizer would also need to calculate the cost of using each plan more accurately.

Also, the implementation does not support deleting and changing documents. Documents can be deleted but the meta data associated with it is not updated. This affects the global count of the terms within the collection. Changes to documents are also not recorded in the indexes or in the meta data.

10 Conclusion

This implementation of sub-element indexing and probabilistic retrieval includes specific routines to perform keyword extraction and relevance ranking. Within the POSTGRES environment, these routines can be user-defined allowing us to test and combine alternative methods. The combination of indexing and ranking routines that we used - Porter stemming and Cooper ranking - works well on the collection set of 1400 documents (or document excerpts) and performs better than the Cosine Correlation method. We expect that it would also work well on a larger set and would like to test this hypothesis. We are also interested in evaluating how this method compares to other ranking strategies. Finally, the indexing scheme can be changed while keeping the ranking module in place. We can test a different stemming strategy and the affects that it has on the ranking results.

Together the indexing and ranking routines form a specialized retrieval model that can increase retrieval effectiveness in specific cases. The specialized model discussed here concentrates on retrieving full-text documents, another may concentrate on retrieving pictures. We are interested in investigating ways to combine various indexing and retrieval methods such as these and evaluating their performance.

Acknowledgements

Paul Aoki and Andrew Yu graciously provided expert technical advice with regard to the POSTGRES database system. Prof. Ray Larson provided invaluable guidance throughout this project and Prof. Robert Wilensky added helpful suggestions for writing this paper. This work was supported in part by the Sequoia 2000 project at the University of California, a project principally funded by the Digital Equipment Corporation and by The Environmental Electronic Library project of the NSF under Grant No. IRI-9411334.

References

[1] Aoki, P. "Implementation of Extended Indexes in POSTGRES," *SIGIR Forum*, Spring 1991, vol.25, (no.1):2-9.

[2] Cooper, Wm.S., Chen, A, and Gey, F.C. "Experiments in the Probabilistic Retrieval of Full Text Documents," *Text Retrieval Conference (TREC-3) Draft Conference Papers*, Gaithersburg, MD : National Institute of Standards and Technology.

[3] Cooper, Wm. S., Gey, F. C., and Chen, A. "Probabilistic Retrieval in the TIPSTER Collections: An Application of Staged Logistic Regression," *Proceedings of the 15th ACM-SIGIR Conference on Research and Development in Information Retrieval*, Copenhagen, Denmark, June 1992.

[4] Croft, W.B, and Harper,D.J. "Using Probabilistic Models of Document Retrieval without Relevance Information," *Journal of Documentation*, 35,285-295.

[5] Frakes, W.B. and Baeza-Yates, R. "Stemming Algorithms," *Information Retrieval: Data Structures and Algorithms*, C-8 (pp. 131-160), Englewood Cliffs, New Jersey: Prentice Hall.

[6] Larson, R. "Design and Development of a Network-Based Electronic Library" *Navigating the Networks: Proceedings of the ASIS Midyear Meeting*, Portland, Oregon, May 21-25, 1994 (pp. 95-114). Medford, NJ: Learned Information, Inc., 1994.

[7] Larson, R. "Evaluation of Advanced Retrieval Techniques in an Experimental Online Catalog" *JASIS: Journal of the Americal Society for Information Science*, January 1992.

[8] Lynch, C. and Stonebraker, M. "Extended User-Defined Indexing with Application to Textual Databases" *Proceedings of the 14th VLDB Conference*, Los Angeles, CA, August 1988.

[9] Rowe, L. and Stonebraker, M. "The Design of POSTGRES" *Proceedings of the 1986 ACM-SIGMOD Conference on Management of Data*, Washington, DC, June 1986.

[10] Stonebraker, M. "Inclusion of New Types in Relational Data Base Systems" *Proceedings of 2nd IEEE Data Engineering Conference*, Los Angeles, CA, February 1986.

[11] Stonebraker, M, Anton, J. and Hanson, E. "Extending a Data Base System with Procedures"

[12] Stonebraker, M. and Olson, M. “Large Object Support in POSTGRES” *Proceedings of 9th International Conference on Data Engineering*, Vienna, Austria, April 1993.

Appendix A - POSTGRES System Changes

10.1 System Catalog Tables Added

pg_rank contains meta data over entire collection.

pg_docs contains meta data per document.

pg_kw contains meta data per term.

pg_stoplist contains stopwords, words excluded from indexing.

Routines to support new system catalog tables:

```
catalog/pg_stoplist.c
catalog/pg_rank.c
catalog/pg_docs.c
catalog/pg_kw.c
```

10.2 Nodes

Node structures are created for the planner and the executor. A rank node was created in *execnodes.h* and plannodes.h and the indexpath node was changed in relation.h.

10.3 Optimizer/Planner

The optimizer/planner was changed to create and rank node and to set up an index path. Changes were made to several files, the most notable changes are in *orindxpath.c* and *allpaths.c*.

10.4 Executor

Most of the changes to the executor can be found in the file *n_rank.c*. This file contains routines to create and execute the "Rank" node.

10.5 Access Method

The fbtree access method routines are located in POSTGRES/src/backend/access/fbtree. Most of the differences between the btree and fbtree access methods can be found in the file *fbtree.c*.

10.6 Ranking & Tokenize

The ranking and tokenizing routines are located in POSTGRES/src/backend/access/index-fbtree.

rank.c has actual ranking equation.

extract.c has default extract routine.

stem.c contains a version of the Porter stemming algorithm.

10.7 Example

```
define index kwind on pg_kw using btree (kw text_ops) \g
define index docsind on pg_docs using btree (docId int4_ops) \g

create docs (docId = int4, doc = text) \g

define index docs_ind on docs using fbtree (doc text_ops) with (extract = kwlist_extract) \g
```