G.PULLAIAH COLLEGE OF ENGINEERING AND TECHNOLOGY,KURNOOL
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

IV B.TECH I SEM (R13)

INFORMATION RETRIEVAL SYSTEMS

UNIT-5

UNIT-V

5.1 A Historical Progression

Previous work can be partitioned into systems that combine information retrieval and DBMS together, or systems that extend relational DBMS to include information retrieval functionality.

5.1.1 Combining Separate Systems

Integrated solutions consist of writing a central layer of software to send requests to underlying DBMS and information retrieval systems. Queries are parsed and the structured portions are submitted as a query to the DBMS, while text search portions of the query are submitted to an information retrieval system. The results are combined and presented to the user. The key advantage of this approach is that the DBMS and information retrieval system are commercial products that are continuously improved. Additionally, software development costs are minimized. The disadvantages include poor data integrity, portability, and run-time performance.

5.1.1.1 Data Integrity

Data integrity is sacrificed because the DBMS transaction log and the information retrieval transaction log are not coordinated. If a failure occurs in the middle of an update transaction, the DBMS will end in a state where the entire transaction is either completed or it is entirely undone. It is not possible to complete half of an update. The information retrieval log (if present) would not know about the DBMS log. Hence, the umbrella application that coordinates work between the two systems must handle all recovery. Recovery done within an application is typically error prone and, in many cases, applications simply ignore this coding. Hence, if a failure should occur in the information retrieval system, the DBMS will not know about it. An update that must take place in both systems can succeed in the DBMS, but fail in the information retrieval system. A partial update is clearly possible, but is logically flawed.

5.1.1.2 Portability

Portability is sacrificed because the query language is not standard. Presently, a standard information retrieval query language does not exist. However, some work is being done to develop standard information retrieval query languages. If one existed, it would require many years for widespread commercial acceptance to occur. The problem is that developers must be retrained each time a new DBMS and information retrieval system is brought in. Additionally, system administration is far more difficult with multiple systems.

5.1.1.3 Performance

Run-time performance suffers because of the lack of parallel processing and query optimization. Although most commercial DBMS have parallel implementations, most information retrieval systems do not. Query optimization exists in every relational DBMS. The optimizer's goal is to

choose the appropriate access path to the data. A rule-based optimizer uses pre-defined rules, while a cost-based optimizer estimates the cost of using different access paths and chooses the cheapest one. In either case, no rules exist for the unstructured portion of the query and no cost estimates could be obtained because the optimizer would be unaware of the access paths that may be chosen by the information retrieval system. Thus, any optimization that included both structured and unstructured data would have to be done by the umbrella application. This would be a complex process. Hence, run-time performance would suffer due to a lack of parallel algorithms and limited global query optimization.

5.1.1.4 Extensions to SQL

Proposals have been made that SQL could be modified to support text. The SMART information retrieval prototype developed in the 1980's used the INGRES relational database system to store its data. In this relevance ranking as well as Boolean searches were supported. The focus was on the problem of efficiently searching text in a RDBMS. RDBMS would store the inverted index in another table thereby making it possible to easily view the contents of the index. An information retrieval system typically hides the inverted index as simply an access structure that is used to obtain data. By storing the index as a relation, the authors pointed out that users could easily view the contents of the index and make changes if necessary. Extensions such as RELEVANCE(*) were mentioned, that would compute the relevance of a document to a query using some pre-defined relevance function. More recently, a language called SQLX was used to access documents in a multimedia database. SQLX assumes that an initial cluster based search has been performed based on keywords. SQLX extensions allow for a search of the results with special connector attributes that obviate the need to explicitly specify joins.

5.1.2 User-defined Operators

User-defined operators that allow users to modify SQL by adding their own functions to the DBMS engine was described. Commercialization of this idea has given rise to several products including the Teradata Multimedia Object Manager, Oracle Cartridges, ruM DB2 Text Extender, as well as features in Microsoft SQL Server. An example query that uses the user-defined area function is given below. Area must be defined as a function that accepts a single argument. The datatype of the argument is given as rectangle. Hence, this example uses both a user-defined function and a user-defined datatype.

Ex: 1 SELECT MAX(AREA(Rectangle)) FROM SHAPE

In the information retrieval domain, an operator such as proximity() could be defined to compute the result set for a proximity search. In this fashion the "spartan simplicity of SQL" is preserved, but users may add whatever functionality is needed. A few years later user-defined operators were defined to implement information retrieval.

The following query obtains all documents that contain the terms termI, term2,and term3:

Ex: 2 SELECT DocJd
FROM DOC
WHERE SEARCH-TERM(Text, Terml, Term2,Term3)

This query can take advantage of an inverted index to rapidly identify the terms. To do this, the optimizer would need to be made aware of the new access method. Hence, user-defined functions also may require user-defined access methods.

The following query uses the proximity function to ensure that the three query terms are found within a window of five terms.

Ex: 3 SELECT DocJd
FROM DOC
WHERE PROXIMITY(Text, 5, Terml, Term2, Term3)

The advantages of user-defined operators are that they not only solve the problem for text, but also solve it for spatial data, image processing, etc. Users may add whatever functionality is required. The key problems with user-defined operators again are integrity, portability, and run-time performance.

5.1.2.1 Integrity

User-defined operators allow application developers to add functionality to the DBMS rather than the application that uses the DBMS. This unfortunately opens the door for application developers to circumvent the integrity of the DBMS. For user-defined operators to be efficient, they must be linked into the same module as the entire DBMS, giving them access to the entire address space of the DBMS. Data that reside in memory or on disk files that are currently opened, can be accessed by the user-defined operator. It is possible that the user-defined operator could corrupt these data.

To protect the DBMS from a faulty user-defined operator, a remote procedure call (RPC) can be used to invoke the user-defined operator. This ensures the operator has access only to its address space, not the entire DBMS address space. Unfortunately, the RPC incurs substantial overhead, so this is not a
solution for applications that require high performance.

5.1.2.2 Portability

A user-defined operator implemented at SITE A may not be present at SITE B. Worse, the operator may appear to exist, but it may perform an entirely different function. Without user-defined operators, anyone with an RDBMS may write an application and expect it to run at any site that runs that RDBMS. With user-defined operators, this perspective changes as the application is limited to only those sites with the user-defined operator.

5.1.2.3 Performance

Query optimization, by default, does not know much about the specific user defined operators. Optimization is often based on substantial information about the query. A query with an EQUAL operator can be expected to retrieve fewer rows than a LESS THAN operator. This knowledge assists the optimizer in choosing an access path.

knowing the semantics of a user-defined operator, the optimizer is unable to efficiently use it. Some user-defined operators might require a completely different access structure like an inverted index. Unless the optimizer knows that an inverted index is present and should be included in path selection, this path is not chosen.

For user-defined operators to gain widespread acceptance, some means of providing information about them to the optimizer is needed. Additionally, parallel processing of a user-defined operator would be something that must be defined inside of the user-defined operator. The remainder of the DBMS would have no knowledge of the user-defined operator, and as such, would not know how to parallelize the operator.

5.1.3 Non-first Normal Form Approaches

Non-first normal form (NFN) approaches have been proposed. The idea is that many-many relationships are stored in a cumbersome fashion when 3NF (third normal form) is used. Typically, two relations are used to store the entities that share the relationship, and a separate relation is used to store the relationship between the two entities.

For an inverted index, a many-many relationship exists between documents and terms. One term may appear in many documents, while one document may have many terms. This, as will be shown later, may be modelled with a DOC relation to store data about documents, a TERM relation to store data about individual terms, and an INDEX relation to track an occurrence of a term in a document.

Instead of three relations, a single NFN relation could store information about a document, and a nested relation would indicate which terms appeared in that document.

Although this is clearly advantageous from a run-time performance standpoint, portability is a key issue. No standards currently exist for NFN collections. Additionally, NFN makes it more difficult to implement ad hoc queries. Since both user-defined operators and NFN approaches have deficiencies, we describe an approach using the unchanged, standard relational model to implement a variety of information retrieval functionality. This support integrity and portability while still yielding acceptable runtime performance. Some applications, such as image processing or CAD/CAM may require user-defined operators, as their processing is fundamentally not set-oriented and is difficult to implement with standard SQL.

5.1.4 Bibliographic Search with Unchanged SQL

The potential of relational systems to provide typical information retrieval functionality included queries using structured data (e.g., affiliation of an author) with unstructured data (e.g., text found in the title of a document). The following relations model the document collection.

• DIRECTORY(name, institution)-identifies the author's name and the institution the author is affiliated with.
• AUTHOR(name, Doc/d)-indicates who wrote a particular document.
• INDEX(term, Doc/d)-identifies terms used to index a particular document

The following query ranks institutions based on the number of
publications that contain input_term in the document.

```
Ex: 4 SELECT UNIQUE institution, COUNT(UNIQUE name)
        FROM DIRECTORY WHERE name IN
            (SELECT name FROM AUTHOR WHERE DocId IN SELECT
                DocId FROM INDEX WHERE term = inputterm
                    ORDER BY 2 DESCENDING)
```

There are several benefits for using the relational model as a foundation for document retrieval. These benefits are the basis for providing typical information retrieval functionality in the relational model, so we will list some of them here.
• Recovery routines
• Performance measurement facilities
• Database reorganization routines
• Data migration routines
• Concurrency control
• Elaborate authorization mechanisms
• Logical and physical data independence
• Data compression and encoding routines
• Automatic enforcement of integrity constraints
• Flexible definition of transaction boundaries (e.g., commit and rollback)
• Ability to embed the query language in a sequential applications language

5.3 Information Retrieval as a Relational Application

Initial extensions are based on the use of a QUERY(term) relation that stores the terms in the query, and an INDEX (DocId, term)relation that indicates which terms appear in which documents. The following query lists all the identifiers of documents that contain at least one term in QUERY:

```
Ex: 5 SELECT DISTINCT(i.DocId) FROM INDEX i, QUERY q WHERE i.term = q.term
```

Frequently used terms or stop terms are typically eliminated from the document collection. Therefore, a STOP _TERM relation may be used to store the frequently used terms. The STOP TERM relation contains a single attribute (term). A query to identify documents that contain any of the terms in the query except those in the STOP _TERM relation is given below:

Ex: 6 SELECT DISTINCT(i.DocId) FROM INDEX i, QUERY q, STOP]ERM
s WHERE i.term = q.term AND i.term i= s.term

Finally, to implement a logical AND of the terms InputTerml, InputTerm2 and InputTerm3,

Ex: 7 SELECT DocId FROM INDEX WHERE term = InputTermI INTERSECT
SELECT DocId FROM INDEX WHERE term = InputTerm2 INTERSECT
SELECT DocId FROM INDEX WHERE term = InputTerm3

The query consists of three components. Each component results in a set of documents that contain a single term in the query. The INTERSECT keyword is used to find the intersection of the three sets. After processing, an AND is implemented.

The key extension for relevance ranking was a corr() function-a built-in function to determine the similarity of a document to a query.

The SEQUEL (a precursor to SQL) example that was given was:

Ex: 8 SELECT DocId FROM INDEX i, QUERY q WHERE i.term =
q.term GROUP BY DocId HAVING CORRO > 60

Other extensions, such as the ability to obtain the first n tuples in the answer set, were given. We now describe the unchanged relational model to implement information retrieval functionality with standard SQL. First, a discussion of preprocessing text into files for loading into a relational DBMS is required.

5.3.1 Preprocessing

Input text is originally stored in source files either at remote sites or locally on CD-ROM. For purposes of this discussion, it is assumed that the data files are in ASCII or can be easily converted to ASCII with SGML markers. SGML markers are a standard means by which different portions of the document are marked. The markers in the working example are found in the TIPSTER collection which was in previous years as the standard dataset for TREC. These markers begin with a < and end with a > (e.g., <TAG).A preprocessor that reads the input file and outputs separate flat files is used. Each term is read and checked against a list of SGML markers. The main algorithm for the preprocessor simply parses terms and then applies a hash function to hash them into a small hash table. If the term has not occurred for this document, a new entry is added to the hash table. Collisions are handled by a single linked list associated with the hash table. If the term already exists, its term frequency is updated. When an end-of-document marker is encountered, the hash table is scanned. For each entry in the hash table a record is generated. The record contains the document identifier for the current document, the

term, and its term frequency. Once the hash table is output, the contents are set to NULL and the process repeats for the next document.

After processing, two output files are stored on disk. The output files are then bulk-loaded into a relational database. Each file corresponds to a relation. The first relation, DOC, contains information about each document. The second relation, INDEX, models the inverted index and indicates which term appears in which document and how often the term has appeared.

The relations are:

INDEX(DocId, Term, TermFrequency)
DOC(DocId, DocName, PubDate, Dateline)

These two relations are built by the preprocessor. A third TERM relation tracks statistics for each term based on its number of occurrences across the document collection. At a minimum, this relation contains the document frequency (df) and the inverse document frequency (idf). The term relation is of the form: TERM(Term, Idf). It is possible to use an application programming interface (API) so that the preprocessor stores data directly into the database. However, for some applications, the INDEX relation has one hundred million tuples or more. This requires one hundred million separate calls to the DBMS INSERT function. With each insert, a transaction log is updated. All relational DBMS provide some type of bulk-load facility in which a large flat file may be quickly migrated to a relation without significant overhead. Logging is often turned off (something not typically possible via an on-line API) and most vendors provide efficient load implementations. For parallel implementations, flat files are loaded using multiple processors. This is much faster than anything that can be done with the API.

For all examples in this chapter, assume the relations were initially populated via an execution of the preprocessor, followed by a bulk load. Notice that the DOC and INDEX tables are output by the preprocessor. The TERM relation is not output. In the initial testing of the preprocessor, it was found that this table was easier to build using the DBMS than within the preprocessor. To compute the TERM relation once the INDEX relation is created, the following SQL statement is used:

Ex: 9 INSERT INTO TERM SELECT Term, log(N / COUNT(*)) FROM INDEX GROUP BY Term

N is the total number of documents in the collection, and it is usually known prior to executing this query. However, if it is not known then SELECT COUNT(*) FROM DOC will obtain this value. This statement partitions the INDEX relation by each term, and COUNT(*) obtains the number of documents represented in each partition (i.e., the document frequency). The idf is computed by dividing N by the document frequency.

Consider the following working example. Input text is provided, and the preprocessor creates two files which are then loaded into the relational DBMS to form DOC and INDEX. Subsequently, SQL is used to populate the TERM relation.

5.3.2 A Working Example

Two documents are taken from the TIPSTER collection and modelled using relations. The documents contain both structured and unstructured data and are given below.

<DOC>
<DOCNO> WSJ870323-0180 </DOCNO>
<HL> Italy's Commercial Vehicle Sales
</HL> <DD> 03/23/87 </DD>
<DATELINE> TURIN, Italy </DATELINE>
<TEXT>
Commercial-vehicle sales in Italy rose 11.4% in February from a year earlier, to 8,848 units, according to provisional figures from the Italian Association of Auto Makers.
</TEXT>
</DOC>
<DOC>
<DOCNO> WSJ870323-0161
</DOCNO> <HL> Who's News: Du Pont
Co. </HL> <DD> 03/23/87 </DD>
<DATELINE> Du Pont Company, Wilmington, DE
</DATELINE> <TEXT>
John A. Krol was named group vice president, Agriculture Products department, of this diversified chemicals company, succeeding Dale E. Wolf, who will retire May 1. Mr. Krol was formerly vice president in the Agricultural Products department.
</TEXT>
</DOC>

The preprocessor accepts these two documents as input and creates the two files that are then loaded into the relational DBMS. The corresponding DOC and INDEX relations are given in following Tables.

Table DOC

DocId	DocName	PubDate	Dateline
1	WSJ870323-0180	3/23/87	TURIN, Italy
2	WSJ870323-0161	3/23/87	Du Pont Company, Wilmington, DE

Table INDEX

DocId	Term	TermFrequency
1	commercial	1
1	vehicle	1
1	sales	1
1	italy	1
1	february	1
1	year	1
1	according	1
...	...	...
2	krol	2
2	president	2
2	diversified	1
2	company	1
2	succeeding	1
2	dale	1
2	products	2
...	...	...

INDEX models an inverted index by storing the occurrences of a term in a document. Without this relation, it is not possible to obtain high performance text search within the relational model. Simply storing the entire document in a Binary Large OBject (BLOB) removes the storage problem, but most searching operations on BLOB's are limited, in that BLOB's typically cannot be indexed. Hence, any search of a BLOB involves a linear scan, which is significantly slower than the O(log n) nature of an inverted index. In a typical information retrieval system, a lengthy preprocessing phase occurs in which parsing is done and all stored terms are identified. A posting list that indicates, for each term, which documents contain that term is identified . A pointer from the term to the posting list is implemented. In this fashion, a hashing function can be used to quickly jump to the term, and the pointer can be followed to the posting list.

The fact that one term can appear in many documents and one document contains many terms indicates that a many-many relationship exists between terms and documents. To model this, document and term may be thought of as entities (analogous to employee and project), and a linking relation that describes the relationship EMP_PROJ must be modeled. The INDEX relation described below models the relationship. A tuple in the INDEX relation is equivalent to an assertion that a given term appears in a given document. Note that the term frequency (tf) or number of occurrences of a term within a document, is a specific characteristic of the APPEARS-IN relationship; thus, it is stored in this table. The primary key for this relation is (DocId,Term), hence, term frequency is entirely dependent upon this key. For proximity searches such as "Find all documents in which the phrase vice president exists," an additional offset attribute is required. Without this, the INDEX relation indicates that vice and president co-occur in the same document, but no information as to their location is given. To indicate that vice is adjacent to president, the offset attribute identifies the current term offset in the document. The first term is given an offset of zero, the second an offset of one, and, in general, the n^{th} is given an offset of n - 1. The INDEX_PROX relation given in Table contains the necessary offset attribute required to implement proximity searches.

Several observations about the INDEX_PROX relation should be noted. Since stop words are not included, offsets are not contiguously numbered. An offset is required for each occurrence of a term. Thus, terms are listed multiple times instead of only once, as was the case in the original INDEX relation.

Table INDEX_PROX

DocId	Term	Offset
1	commercial	0
1	vehicle	1
1	sales	2
1	italy	4
1	rose	5
1	february	8
1	year	11
...	...	...
1	makers	26
...	...	...
2	krol	2
...	...	...

To obtain the INDEX relation from INDEX..PROX, the following statement can be used:

Ex: 10 INSERT INTO INDEX SELECT DocId, Term, COUNT(*) FROM
INDEX..PROX GROUP BY DocId, Term

Finally, single-valued information about terms is required. The TERM relation contains the idf for a given term. To review, a term that occurs frequently has a low idf and is assumed to be relatively unimportant. A term that occurs infrequently is assumed very important. Since each term has only one idf, this is a single-valued relationship which is stored in a collection-wide single TERM relation.

Table Term

Term	Idf
according	0.9031
commercial	1.3802
company	0.6021
dale	2.3856
diversified	2.5798
february	1.4472
italy	1.9231
krol	4.2768
president	0.6990
products	0.9542
...	...
...	...
sales	1.0000
succeeding	2.6107
vehicle	1.8808
year	0.4771
...	...

To maintain a syntactically fixed set of SQL queries for information retrieval processing, and to reduce the syntactic complexities of the queries themselves, a QUERY relation is used. The QUERY relation contains a single tuple for each query term. Queries are simplified because the QUERY relation can be joined to INDEX to see if any of the terms in QUERY are found in INDEX. Without QUERY, a lengthy WHERE clause is required to specifically request each term in the query.

Finally, STOP_TERM is used to indicate all of the terms that are omitted during the parsing phase. This relation illustrates that the relational model can store internal structures that are used during data definition and population.

Table query

Term	tf
vehicle	1
sales	1

Table STOP_TERM

Term
a
an
and
...
the
...

The following query illustrates the potential of this approach. The SQL satisfies the request to "Find all documents that describe vehicles and sales written on 3/23/87." The keyword search covers unstructured data, while the publication date is an element of structured data. This example is given to quickly show how to integrate both structured data and text. Most information retrieval systems support this kind of search by making DATE a "zoned field"-a portion of text that is marked and always occurs in a particular section or zone of a document. These fields can then be parsed and stored in a relational structure. Example illustrates a sequence of queries that use much more complicated unstructured data, which could not easily be queried with an information retrieval system.

Ex: 11 SELECT d.Doc1d FROM DOC d, INDEX I WHERE i.Term IN ("vehicle", "sales")
AND d.PubDate = "3/23/87" AND d.DocId = i.DocId

5.3.3 Boolean Retrieval

A Boolean query is given with the usual operators-AND, OR, and NOT. The result set must contain all documents that satisfy the Boolean condition. For small bibliographic systems (e.g., card catalog systems), Boolean queries are useful. They quickly allow users to specify their information need and return all matches. For large document collections, they are less useful because the result set is unordered, and a query can result in thousands of matches. The user is then forced to tune the Boolean conditions and retry the query until the result is obtained. Relevance ranking avoids this problem by ranking documents based on a measure of relevance between the documents and the query. The user then looks at the top-ranked documents and determines whether or not they fill the information need. We start with the use of SQL to implement Boolean retrieval. We then show how a proximity search can be implemented with unchanged SQL, and finally, a relevance ranking implementation with SQL is described. The following SQL query returns all documents that contain an arbitrary term, InputTerm.

Ex: 12 SELECT DISTINCT(i.DocId) FROM INDEX i WHERE i.Term = InputTerm

Obtaining the actual text of the document can now be performed in an application specific fashion. The text is found in a single large attribute that contains a BLOB or CLOB (binary or character large object), possibly divided into separate components (i.e., paragraphs, lines, sentences, phrases, etc.). If the text is
found in a single large attribute (in this example we call it Text), the query can be extended to execute a subquery to obtain the document identifiers. Then the identifiers can be used to find the appropriate text in DOC.

Ex: 13 SELECT d.Text FROM DOC d WHERE d.DocId IN
(SELECT DISTINCT(i.DocId) FROM INDEX i WHERE i.Term = InputTerm)

It is natural to attempt to extend the query in Example 12 to allow for n terms. If the Boolean request is an OR, the extension is straightforward and does not increase the number of joins found in the query.

Ex: 14 SELECT DISTINCT(i.DocId) FROM INDEX I WHERE i. Term = InputTerml OR i.
Term = InputTerm2 OR
i.Term = InputTerm3 OR
….
i. Term = InputTermN

Unfortunately, a Boolean AND results in a dramatically more complex query. For a query containing n input terms, the INDEX relation must be joined n times. This results in the following query.

Ex: 15 SELECT a.DocId FROM INDEX a, INDEX b, INDEX c, ... INDEX n - 1, INDEX
n WHERE a.Term = InputTerml AND
b.Term = InJYUtTerm2 AND
c.Term = InputTerm3 AND
…..
n.Term = $InputTerm_n$ AND
a.DocId = b.DocId AND
b.DocId = c.DocId AND
…
n - l.DocId = n.DocId

Multiple joins are expensive. The order that the joins are computed affects performance, so a cost-based optimizer will compute costs for many of the orderings . Pruning is expensive. In addition to performance concerns, the reality is that commercial systems are unable to implement more than a fixed number of joins. Although it is theoretically possible to execute a join of n terms, most implementations impose limits on the number of joins (around sixteen is common). It is the complexity of this simple Boolean AND that has led many researchers to develop extensions to SQL or user-defined operators to allow for a more simplistic SQL query.
An approach that requires a fixed number of joins regardless of the number of terms found in the input query reduces the number of conditions in the query. However, an additional sort is needed (due to a GROUP BY) in the query where one previously did not exist.

The following query computes a Boolean AND using standard syntactically fixed SQL:

Ex: 16 SELECT i.DocId FROM INDEX i, QUERY q WHERE i.Term = q.Term

GROUP BY i.DocId HAVING COUNT(i.Term) = (SELECT COUNT(*) FROM QUERY)

The WHERE clause ensures that only the terms in the query relation that match those in INDEX are included in the result set. The GROUP BY specifies that the result set is partitioned into groups of terms for each document. The HA VING ensures that the only groups in the result set will be those whose cardinality is equivalent to that of the query relation. For a query with k terms (t1, t2, ... , tk), the tuples as given in Table are generated for document d_i containing all k terms.

Table Result Set

DocId	*term*
d_i	t_1
d_i	t_2
...	...
d_i	t_k

The GROUP BY clause causes the cardinality, k, of this document to be computed. At this point, the HAVING clause determines if the k terms in this group matches the number of terms in the query. If so, a tuple di appears in the final result set.

Until this point, we assumed that the INDEX relation contains only one occurrence of a given term for each document. This is consistent with our example where a term frequency is used to record the number of occurrences of a term within a document. In proximity searches, a term is stored multiple times in the INDEX relation for a single document. Hence, the query must be modified because a single term in a document might occur k times which results in d_i being placed in the final result set, even when it does not contain the remaining k - 1 terms.

The following query uses the DISTINCT keyword to ensure that only the distinct terms in the document are considered. This query is used on INDEX relations in which term repetition in a document results in term repetition in the INDEX relation.

Ex: 17 SELECT i.DocId FROM INDEX i, QUERY q WHERE i. Term = q. Term
GROUP BY i.DocId HAVINGCOUNT(DISTINCT(i.Term)) = (SELECT COUNT(*) FROM QUERY)

This query executes whether or not duplicates are present, but if it is known that duplicate terms within a document do not occur, this query is somewhat less efficient than its predecessor. The DISTINCT keyword typically requires a sort.

Using a set-oriented approach to Boolean keyword searches results in the fortunate side-effect that a Threshold AND (TAND) is easily implemented. A partial AND is one in which the condition is true if k subconditions are true. All of the subconditions are not required. The following query returns all documents that have k or more terms matching those found in the query.

```
Ex: 18 SELECT i.DocId FROM INDEX i,QUERY q WHERE i. Term = q. Term
       GROUP BY i.DocId HAVING COUNT(DISTINCT(i.Term)) ≥k
```

5.3.4 Proximity Searches

To briefly review, proximity searches are used in information retrieval systems to ensure that the terms in the query are found in a particular sequence or at least within a particular window of the document. Most users searching for a query of "vice president" do not wish to retrieve documents that contain the sentence, "His primary vice was yearning to be president of the company." To implement proximity searches, the INDEXYROX given in our working example is used. The offset attribute indicates the relative position of each term in the document.

The following query, albeit a little complicated at first glance, uses unchanged SQL to identify all documents that contain all of the terms in QUERY within a term window of width terms. For the query given in our working example, "vice" and "president" occur in positions seven and eight, respectively.

Document two would be retrieved if a window of two or larger was used.

```
Ex: 19 SELECT a.DocId FROM INDEXYROX a, INDEXJ>ROX b
       WHERE a.Term IN (SELECT q.Term FROM QUERY q) AND
             b. Term IN (SELECT q. Term FROM QUERY q) AND
             a.DocId = b.DocId AND
            (b. Offset - a.Offset) BETWEEN 0 AND (width - 1)
       GROUP BY a.DocId, a.Term, a.Offset
        HAVING COUNT(D/STINCT(b.Term)) =(SELECT COUNT(*) FROM QUERY)
```

The INDEX_PROX table must be joined to itself since the distance between each term and every other term in the document must be evaluated. For a document di that contains k terms (t1, t2, ... , tk) in the corresponding term offsets of (01, 02, ... , Ok), the first two conditions ensure that we are only examining offsets for terms in the document that match those in the query. The third condition ensures that the offsets we are comparing do not span across documents.

The following tuples make the first three conditions evaluate to TRUE.

In following Table, we illustrate the logic of the query. Drawing out the first step of the join of INDEX_ PROX to itself for an arbitrary document di yields tuples in which each term in INDEX_TERM is matched with all other terms. This table shows only those terms within

document di that matched with other terms in document di. This is because only these tuples evaluate to TRUE when the condition "a.DocId = b.DocId" is applied. We also assume that the terms in the table below match those found in the query, thereby satisfying the condition "b.term IN (SELECT q.term FROM QUERY)."

Table Result of self-join of INDEX_PROX

a.DocId	*a.Term*	*a.Offset*	*b.DocId*	*b.Term*	*b.Offset*
d_i	t_1	o_1	d_i	t_1	o_1
d_i	t_1	o_1	d_i	t_2	o_2
d_i	t_1	o_1	d_i	t_k	o_k
d_i	t_2	o_2	d_i	t_1	o_1
d_i	t_2	o_2	d_i	t_2	o_2
d_i	t_2	o_2	d_i	t_k	o_k
d_i	t_k	o_k	d_i	t_1	o_1
d_i	t_k	o_k	d_i	t_2	o_2
d_i	t_k	o_k	d_i	t_k	o_k

The fourth condition examines the offsets and returns TRUE only if the terms exist within the specified window. The GROUP BY clause partitions each particular offset within a document. The HAVING clause ensures that the size of this partition is equal to the size of the query. If this is the case, the document has all of terms in QUERY within a window of size offset. Thus, document di is included in the final result set. For an example query with "vehicle" and "sales" within a two term window, all four conditions of the WHERE clause evaluate to TRUE for the following tuples. The first three have eliminated those terms that were not in the query, and the fourth eliminated those terms that were outside of the term window. The GROUP BY clause results in a partition in which "vehicle", at offset one, is in one partition and "sales", at offset two, is in the other partition. The first partition has two terms which match the size of the query, so document one is included in the final result set (see Table).

Table Result after all four conditions of the WHERE clause

a.DocId	*a.Term*	*a.Offset*	*b.DocId*	*b.Term*	*b.Offset*
1	vehicle	1	1	vehicle	1
1	vehicle	1	1	sales	2
1	sales	2	1	sales	2

5.3.5 Computing Relevance Using Unchanged SQL

Relevance ranking is critical for large document collections as a Boolean query frequently returns many thousands of documents. Numerous algorithms exist to compute a measure of similarity between a query and a document. The vector-space model is commonly used. Systems

based on this model have repeatedly performed well at the Text REtrieval Conference (TREC). Recall, that in the vector space model, documents and queries are represented by a vector of size t, where t is the number of distinct terms in the document collection. The distance between the query vector Q and the document vector Di is used to rank documents. The following dot product measure computes this distance:

$$SC(Q, D_i) = \sum_{j=1}^{t} w_{qj} \times d_{ij}$$

where Wqj is weight of the j^{th} term in the query q, and d_{ij} is the weight of the j^{th} term in the i^{th} document. In the simplest case, each component of the vector is assigned a weight of zero or one (one indicates that the term corresponding to this component exists). Numerous weighting schemes exist, an example of which is tf-idf Here, the term frequency is combined with the inverse document frequency.
The following SQL implements a dot product query with the tf-idf weight.

```
Ex: 20 SELECT i.DocId, SUM(q.tf*t.idf*i.tf*t.idf) FROM QUERY q, INDEX i, TERM
        t WHERE q.Term = t.Term AND i.Term = t.Term
            GROUP BY i.DocId ORDER BY 2 DESC
```

The WHERE clause ensures that only terms found in QUERY are included in the computation. Since all terms not found in the query are given a zero weight in the query vector, they do not contribute to the summation. The idf is obtained from the TERM relation and is used to compute the tf-idf weight in the select-list. The ORDER BY clause ensures that the result is sorted by the similarity coefficient. At this point, we have used a simple similarity coefficient. Unchanged SQL can be used to implement these coefficients as well. Typically, the cosine coefficient or its variants is commonly used. The cosine coefficient is defined as:

$$SC(Q, D_i) = \frac{\sum_{j=1}^{t} w_{qj} d_{ij}}{\sqrt{\sum_{j=1}^{t} (d_{ij})^2 \sum_{j=1}^{t} (w_{qj})^2}}$$

The numerator is the same as the dot product, but the denominator requires a normalization which uses the size of the document vector and the size of the query vector. Each of these normalization factors could be computed at query time, but the syntax of the query becomes overly complex. To simplify the SQL, two separate relations are created: DOC_WT (DocId, Weight) and QUERY _WT (Weight). DOC_WT stores the size of the document vector for each document and QUERY _WT contains a single tuple that indicates the size of the query vector.
These relations may be populated with the following SQL:

```
Ex: 21 INSERT INTO DOC_WT SELECT DocId, SQRT(SUM(i.tf* t.idf* i.tf* t.idf))
            FROM INDEX i, TERM t WHERE i.Term = t.Term
```

GROUP BY DocId

Ex: 22 INSERT INTO QRLWT SELECT SQRT(SUM(q.tf* t.idf * q.tf* t.idf))
FROM QUERY q, TERM t WHERE q.Term = t.Term

For each of these INSERT-SELECT statements, the weights for the vector are computed, squared, and then summed to obtain a total vector weight. The following query computes the cosine.

Ex: 23 SELECT i.DocId, SUM(q.tf* t.idf* i.tf* t.idf)/(dw. Weight * qw. Weight)
FROM QUERY q, INDEX i, TERM t, DOCWT dw, QRY_WT qw
WHERE q. Term = t. Term AND i. Term = t. Term AND i.DocId = dw.DocId
GROUP BY i.DocId, dw. Weight, qw. Weight ORDER BY 2 DESC

The inner product is modified to use the normalized weights by joining the two new relations, DOC_W T and QRY _W T. An additional condition is added to the WHERE clause in order to obtain the weight for each document. To implement this coefficient, it is necessary to use the built-in square root function which is often present in many SQL implementations. We note that these queries can all be implemented without the non-standard square root function simply by squaring the entire coefficient. This modification does not affect the document ranking as as $a \leq b \rightarrow a^2 \leq b^2$ for $a, b \geq 0$. For simplicity of presentation, we used a built-in sqrt function (which is present in many commercial SQL implementations) to compute the square root of an argument. Modifications to the SUM() element permit implementation of other similarity measures. For instance, with the additional computation and storage of some document statistics, (log of the average term frequency), some collection statistics (average document length and the number of documents) and term statistics (document frequency), pivoted normalization and a probabilistic measure can be implemented. Essentially, the only change is that the SUM operator is modified to contain new weights. The result is that fusion of multiple similarity measures can be easily implemented in SQL.

5.3.6 Relevance Feedback in the Relational Model

Relevance feedback can be supported using the relational model. Recall, relevance feedback is the process of adding new terms to a query based on documents presumed to be relevant in an initial running of the query. Separate SQL statements were used for each of the following steps:
Step 1: Run the initial query. This is done using the SQL we have just described.
Step 2: Obtain the terms in the top n documents. A query of the INDEX relation given a list of document identifiers (these could be stored in a temporary relation generated by Step 1) will result in a distinct list of terms in the top n documents. This query will run significantly faster if

the DBMS has the ability to limit the number of tuples returned by a single query (many commercial systems have this capability). An INSERT-SELECT can be used to insert the terms obtained in this query into the QUERY relation.
Step 3: Run the modified query. The SQL remains the same as used in Step 1.

5.3.7 A Relational Information Retrieval System

The need to integrate structured and unstructured data led to the development of a scalable, standard SQL-based information retrieval prototype engine called SIRE. The SIRE approach, initial built for the National Institutes of Health National Center for Complementary and Alternative Medicine, leverages the investment of the commercial relational database industry by running as an application of the Oracle DBMS. It also includes all the capabilities of the more traditional customized information retrieval approach. Furthermore, additional functionality common in the relational database world, such as concurrency control, recovery, security, portability, scalability, and robustness, are provided without additional effort. Such functionality is not common in the traditional information retrieval market. Also, since database vendors continuously improve these features and likewise incorporate advances made in hardware and software, a SIRE-based solution keeps up with the technology curve with less investment on the part of the user as compared to a more traditional (custom) information retrieval system solution. To demonstrate the applicability and versatility of SIRE, key information retrieval strategies and utilities such as leading similarity measures, proximity searching, n-grams, passages, phrase indexing, and relevance feedback were all implemented using standard SQL. By implementing SIRE on a host of relational database platforms including NCR DBC-1012, Microsoft SQL Server, Sybase, Oracle, IBM DB2 and SQLIDS, and even mySQL, system portability was demonstrated. Efficiency was enhanced using several optimization approaches and some specific to relational database technology. These included the use of a pruned index and query thresholds as well as clustered indexes. All of these optimizations reduced the I/O volume, hence significantly reduced query execution time. More recent related efforts have focused on scaling the SIRE-based approach using parallel technology and incorporating efficient document updating into the paradigm. With the constant changes to text available particularly in Web environments, updating of the documents is becoming a necessity. Traditionally, information retrieval was a "read only" environment. Early parallel processing efforts used an NCR machine configured with 24 processors and achieved a speedup of 22-fold.

5.4 Semi-Structured Search using a Relational Schema

Numerous proprietary approaches exist for searching eXtensible Markup Language (XML) documents, but these lack the ability to integrate with other structured or unstructured data. Relational systems have been used to support XML by building a separate relational schema to map to a particular XML schema or DTD (Document-type Definitions) .

5.4.1 Background

XML has become the standard for platform-independent data exchange. There were a variety of methods proposed for storing XML data and accessing them efficiently .One approach is a customized tree file structure, but this lacks portability and does not leverage existing database technology . Other approaches include building a database system specifically tailored to storing semi-structured data from the ground or using a full inverted index . There are several popular XML query languages.

5.4.2 Static Relational Schema to support XML-QL

We describe a static relational schema that supports arbitrary XML schema to provide support for XML query processing. Later, IIT Information Retrieval Laboratory (www.ir.iit.edu) shown that a full XML-QL query language could be built using this basic structure. This is done by translating semi-structured XML-QL to SQL. The use of a static schema accommodates data of any XML schema without the need for document-type definitions or XSchemas.

The static relational storage schema stores each unique XML path and its value from each document as a separate row in a relation. This is similar to the edge table named for the fact that each row corresponds to an edge in the XML graph representation. This static relational schema is capable of storing an arbitrary XML document. The hierarchy of XML documents is kept in tact such that any document indexed into the database can be reconstructed using only the information in the tables. The relations used are:

TAG_NAME (TagId, tag) ATTRIBUTE (AttributeId, attribute)
TAG_PATH (TagId, path) DOCUMENT (DocId, fileName)
INDEX (Id, parent, path, type, tagId, attrId, pos, value)

5.4.3 Storing XML Metadata

These tables store the metadata (data about the data) of the XML files. TAG_NAME and TAG_PATH together store the information about tags and paths within the XML file. TAG-
.NAME stores the name of each unique tag in the XML collection. TAG_PATH stores the unique paths found in the XML documents. The ATTRIBUTE relation stores the names of all the attributes. In our example, we have added an attribute called LANGUAGE which is an attribute of the tag TEXT. In the TAG_NAME and TAG_PATH relations, the tagld is a unique key assigned by the preprocessing stage. Similarly, attributeId is uniquely assigned as well. As with our examples earlier in the chapter, these tables are populated each time a new XML file is indexed. This process consists of parsing the XML file and extracting all of this information and storing it into these tables.

5.4.4 Tracking XML Documents

Since XML-QL allows users to specify what file(s) they wish to query, many times we do not want to look at each record in the database but only at a subset of records that correspond to that file. Each time a new file is indexed,

Table TAG_NAME

tagId	tag
10	DOC
11	DOCNO
12	HL
13	DD
14	DATELINE
15	TEXT

Table TAG_PATH

tagId	path
10	[DOC]
11	[DOC, DOCNO]
12	[DOC, HL]
13	[DOC, DD]
14	[DOC,DATELINE]
15	[DOC,TEXT]

Table ATTRIBUTE

AttributeId	attribute
7	LANGUAGE

It receives a unique identifier that is known as the pin value. This value corresponds to a single XML file. The DOCUMENT relation contains a tuple for each XML document. For our example, we only store the actual file name that contains this document. Other relevant attributes might include the length of the document - or the normalized length .

Table DOCUMENT

docId	fileName
2	doc_0.xml
3	doc_1.xml

5.4.5 INDEX

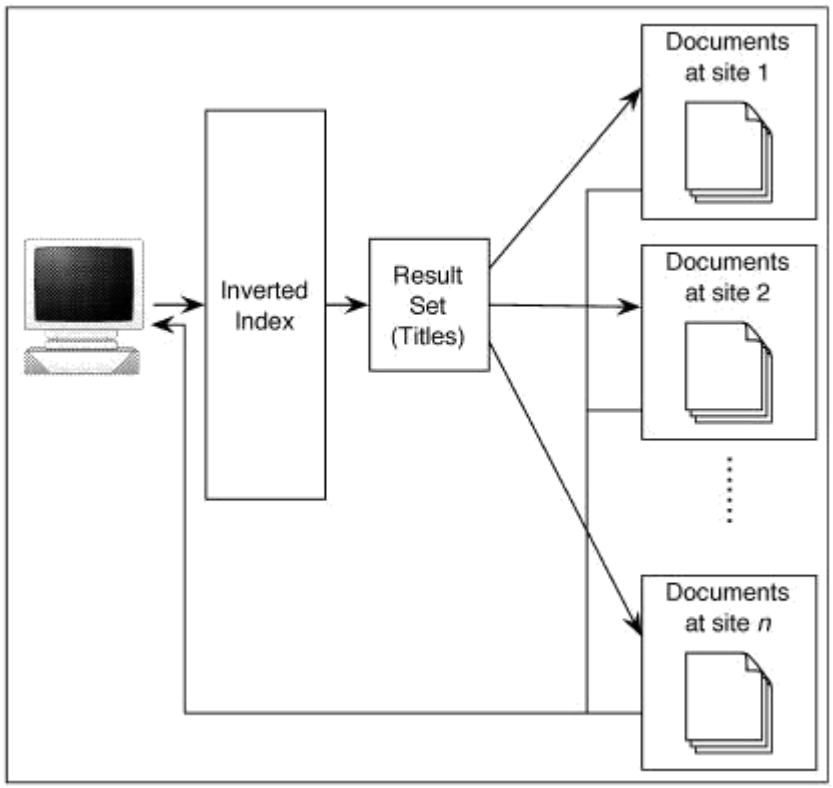
The INDEX table models an XML index. It contains the mapping of each tag, attribute or value to each document that contains this value. Also, since the order of attributes and tags is important in XML (e.g.;
there is a notion of the first occurrence, the second occurrence, etc.), the position or order of the tags is also stored. The id column is a unique integer assigned to each element and attribute in a document. The parent attribute indicates the id of the tag that is the parent of the current tag. This is needed to preserve the inherently hierarchical nature of XML documents. The path corresponds to the primary key value in the TagPath. The type indicates whether the path terminates with an element or an attribute (E or A). The Tagld and Attrld is a foreign key to the Tagld in the TagName and Attribute tables. The DocId attribute indicates the XML document for a given row. The pos tracks the order of a given tag and is used for queries that use the index expression feature of XML-QL and indicates the position of this element relative to others under the same parent (starting at zero). This column stores the original ordering of the input XML for explicit usage in users' queries. Finally, value contains the atomic unit in XML - the value inside the lowest level tags. Once we have reached the point of value, all of the prior means of using relations to model an inverted index for these values apply.

Table INDEX

id	parent	path	type	tagId	AttrId	docId	pos	value
41	0	10	E	10	1	6	0	NULL
42	41	11	E	11	1	6	0	WSJ870323-0180
43	41	12	E	12	1	6	0	Italy's Commercial...
44	41	13	E	13	1	6	0	03/23/87
45	41	14	E	14	1	6	0	TURIN, Italy
46	41	15	E	15	1	6	0	Commercial-vehicle ...
47	46	15	A	15	7	6	0	English
48	0	10	E	10	1	7	0	NULL
49	48	11	E	11	1	7	0	WSJ870323-0161
50	48	12	E	12	1	7	0	Who's News...
51	48	13	E	13	1	7	0	03/23/87
52	48	14	E	14	1	7	0	Du Pont Co...DE
53	48	15	E	15	1	7	0	John A. Krol ...
54	53	15	A	15	7	7	0	English

5.5 DISTRIBUTED INFORMATION RETRIEVAL

Until now, we focused on the use of a single machine to provide an information retrieval service and the use of a single machine with multiple processors to improve performance. Although efficient performance is critical for user acceptance of the system, today, document collections are often scattered across many different geographical areas. Thus, the ability to process the data where they are located is arguably even more important than the ability to efficiently process them. Possible constraints prohibiting the centralization of the data include data security, their sheer volume prohibiting their physical transfer, their rate of change, political and legal constraints, as well as other proprietary motivations. One of the latest popular processing infrastructures is the "Grid". The grid is named after the global electrical power grid. In the power grid, appliances (systems in our domain) simply "plug in" and immediately operate and become readily available for use or access by the global community. A similar notion in modem search world is the use of Distributed Information Retrieval Systems (DIRS). DIRS provides access to data located in many different geographical areas on many different machines. In the early 1980's, it was already clear that distributed information retrieval systems would become a necessity. Initially, a theoretical model was developed that described some of the key components of a distributed information retrieval system.



5.5.1 A Theoretical Model of Distributed Retrieval

We first define a model for a centralized information retrieval system and then expand that model to include a distributed information retrieval system.

5.5.1.1 Centralized Information Retrieval System Model

Formally, an information retrieval system is defined as a triple, I = (D, R, δ) where D is a document collection, R is the set of queries, and $\delta j : Rj \rightarrow 2^{Dj}$ is a mapping assigning the j^{th} query to a set of relevant documents.

Many information retrieval systems rely upon a thesaurus, in which a user query is expanded, to include synonyms of the keywords to match synonyms in a document. Hence, a query that contains the term curtain will also include documents containing the term drapery. To include the thesaurus in the model, The triple be expanded to a quadruple as:

$$I= (T,D,R, \delta)$$

where T is a set of distinct terms and the relation $\rho \cap T \times T$ such that ρ (tl, t2) implies that tl is a synonym of t2. Using the synonym relation, it is possible to represent documents as a set of descriptors and a set of ascriptors. Consider a document D I , the set of descriptors d consists of all terms in DI such that:

• Each descriptor is unique

• No descriptor is a synonym of another descriptor

An ascriptor is defined as a term that is a synonym of a descriptor. Each ascriptor must be synonymous with only one descriptor. Hence, the descriptors represent a minimal description of the document. The generalization relation assumes that it is possible to construct a hierarchical knowledge base of all pairs of descriptors. Construction of such knowledge bases was attempted both automatically and manually, but many terms are difficult to define. Relationships pertaining to spatial and temporal substances, ideas, beliefs, etc. tend to be difficult to represent in this fashion. However, this model does not discuss how to construct such a knowledge base, only some interesting properties that occur if one could be constructed.

The motivation behind the use of a thesaurus is to simplify the description of a document to only those terms that are not synonymous with one another. The idea being that additional synonyms do not add to the semantic value of the document. The generalization relation is used to allow for the processing of a query that states "List all animals" to return documents that include information about dogs, cats etc. even though the term dog or cat does not appear in the document.

The generalization can then be used to define a partial ordering of documents. Let the partial ordering be denoted by ≤ and let t(di) indicate the list of descriptors for document di . Partial ordering,≤, is defined as:

$$t(d_1) \preceq t(d_2) \Leftrightarrow (\forall t' \in t(d_1))(\exists t'' \in t(d_2))(\gamma(t', t''))$$

Hence, a document d1 whose descriptors are all generalizations of the descriptors found in d2 will have the ordering d1 ≤ d2. For example, a document with the terms animal and person will precede a document with terms dog and John. Note that this is a partial ordering because two documents with terms that have no relationship between any pairs of terms will be unordered. To be inclusive, the documents that correspond to a general query q1 must include (be a superset ot) all documents that correspond to the documents that correspond to a more specific query q2, where q1 ≤ q2. Formally:

$$(q_1, q_2 \in Q) \wedge (q_1 \preceq q_2) \rightarrow (\delta(q_1) \supset \delta(q_2))$$

This means that if two queries, q1 and q2, are presented to a system such that q1 is more general than q2, it is not necessary to retrieve from the entire document collection for each query. It is only necessary to obtain the answer set for ɣ(q1) to obtain the ɣ(q2).

5.5.1.2 Distributed Information Retrieval System Model

The centralized information retrieval system can be partitioned into n local information retrieval systems s1,s2 , ... ,sn . Each system sj is of the form: sj = (Tj, D j , Rj , δj), where Tj is the thesaurus; D j is the document collection; Rj the set of queries; and δj : Rj $\rightarrow$ 2^{Dj} maps the

queries to documents. By taking the union of the local sites, it is possible to define the distributed information retrieval system as:

$$s= (T,D,R, \delta)$$

where:

$$T = \bigcup_{j=1}^{n} T_j$$

$$s_j = s \bigcap (T_j \times T_j), R_j = R \bigcap (d_j \times d_j)$$

This states that the global thesaurus can be reconstructed from the local thesauri, and the queries at the sites j will only include descriptors at site j. This is done so that the terms found in the query that are not descriptors will not retrieve any documents.

$$D = \bigcup_{j=1}^{n} D_j$$

The document collection, D, can be constructed by combining the document collection at each site.

$$R \supset \bigcup_{j=1}^{n} R_j, \preceq_j = \preceq \bigcap (R_j \times R_j)$$

The queries can be obtained by combining the queries at each local site. The partial ordering defined at site j will only pertain to queries at site j.

$$(\forall r \in R)(\delta(r) = d : d \in D \wedge r \preceq t(d))$$

For each query in the system, the document collection for that query contains documents in the collection where the documents are at least as specific as the query. The hierarchy represented by I is partitioned among the different sites. A query sent to the originating site would be sent to each local site and a local query would be performed. The local responses are sent to the originating site where they are combined into a final result set. The model allows for this methodology if the local sites satisfy the criteria of being a subsystem of the information retrieval system.

$$S_1 = (T_1, D_1, R_1, \delta_1) \text{ is a subsystem of } S_2 = (T_2, D_2, R_2, \delta_2) \text{ if:}$$

$$(T_1 \supset T_2) \wedge (R_1 = R_2) \bigcap (d_1 \times d_2) \wedge (s_1 = s_2) \bigcap (T_1 \times T_2)$$

The thesaurus of T1 is a superset of T2

$$D_1 \supset D_2$$

The document collection at site s1 contains the collection D2

$$R_1 \in R_2 \wedge \preceq_1 = \preceq_2 \bigcap (R_1 \times R_2)$$

The queries at site s1 contain those found in s2.

$$\delta_1(r) = \delta_2(r) \bigcap D_1 for\ r \in R$$

The document collection returned by queries in s1 will include all documents returned by queries in s2 The following example illustrates that an arbitrary partition of a hierarchy may not yield valid subsystems.

Consider the people hierarchy:

ɣ(people, Harold), ɣ(people, Herbert), ɣ (people, Mary)

and the second animal hierarchy:

ɣ(animal, cat), ɣ(animal, dog), ɣ,(cat, black-cat), ɣ(cat, cheshire), ɣ(dog, doberman), ɣ(dog, poodle)

Assume that the hierarchy is split into sites s1 and s2. The hierarchy at s1 is:

ɣ(people, Harold), ɣpeople, Mary)

ɣ(animal, cat), ɣ(animal, dog), ɣ(dog, doberman), ɣ(dog, poodle)

The hierarchy at s2 is:

ɣ(people, Herbert), ɣ(people, Harold), ɣ(animal, cat), ɣ(animal, doberman), ɣ(cat, cheshire), ɣ(cat, black-cat)

Consider a set of documents with the following descriptors:

D1 = (Mary, Harold, Herbert)

D2 = (Herbert, dog)

D3 = (people, dog)

D4 = (Mary, cheshire)

D5 = (Mary, dog)

D6 = (Herbert, black-cat, doberman)

D7 = (Herbert, doberman)

A query of the most general terms (people, animal) should return all documents 2 through 7 (document 1 contains no animals, and the query is effectively a Boolean AND). However, the hierarchy given above as s1 will only retrieve documents D3 and D5 , and s2 will only retrieve documents D6 and D7. Hence, documents D2 and D4 are missing from the final result if the local results sets are simply concatenated. Since, the document collections cannot simply be concatenated, the information retrieval systems at sites s1 and s2 fail to meet the necessary criterion to establish a subsystem.

In practical applications, there is another problem with the use of a generalization hierarchy. Not only are they hard to construct, but also it is non-trivial to partition them. This distributed model was expanded to include weighted keywords for use with relevance.

5.6 Web Search

Search tools that access Web pages via the Internet are prime examples of the implementation of many of the algorithms and heuristics. These systems are, by nature, distributed in that they access data stored on Web servers around the world. Most of these systems have a centralized index, but all of them store pointers in the form of hypertext links to various Web servers. These systems service tens of millions of user queries a day, and all of them index several Terabytes of Web pages. We do not describe each search engine in vast detail because search engines change very. We note that sixteen different Web search engines are listed at www.searchenginewatch.com while www.searchengineguide.com lists over 2,500 specialized search engines.

5.6.1 Evaluation of Web Search Engines

The traditional information retrieval environments, individual systems are evaluated using standard queries and data. In the Web environment, such evaluation conditions are unavailable. Furthermore, manual evaluations on any grand scale are virtually impossible due to the vast size and dynamic nature of the Web. To automatically evaluate Web search engines, a method using online taxonomies that were created as part of Open Directory Project (ODP) . Online directories were used as known relevant items for a query. If a query matches either the title of the item stored or the directory file name containing a known item then it is considered a match. The authors compared the system rankings achieved using this automated approach versus a limited scale, human user based system rankings created using multiple individual users. The two sets of rankings were statistically identical.

5.6.2 High Precision Search

Another concern in evaluating Web search engines is the differing measures of success as compared to traditional environments. Traditionally, precision and recall measures are the main evaluation metrics, while response time and space requirements are likely addressed. However, in the Web environment, response time is critical. Furthermore, recall estimation is very difficult, and precision is of limited concern since most users never access any links that appear beyond the first answer screen (first ten potential reference links). Thus, Web search engine developers focus on guaranteeing that the first results screen is generated quickly, is highly accurate, and that no severe accuracy mismatch exists. Text is efficiently extracted from template generated Web documents; the remainder of the frame or frames are discarded to prevent identifying a document as relevant as a result of potentially an advertisement frame matching the query. Efficient, high-precision measures are used to quickly sift and discard any item that is not with great certainty relevant as a top-line item to display in a current news listing service.

5.6.3 Query Log Analysis

In summary, although similar and relying on much the same techniques as used in traditional information retrieval system domains, the Web environment provides for many new opportunities to revisit old issues particularly in terms of performance and accuracy optimizations and evaluation measures of search accuracy. In that light, recently, an hourly analysis of a very large topically categorized Web query log was published. Using the results presented, it is possible to generate many system optimizations. For example, as indicated in the findings presented, user request patterns repeat according to the time of day and day of week. Thus, depending on the time of day and day of week, it is possible to pre-cache likely Web pages in anticipation of a set of user requests. Thus, page access delays are reduced increasing system throughput. Furthermore, in terms of accuracy optimization, it is likewise possible to adjust the ranking measures to better tune for certain anticipated user subject requests. In short, many optimizations are possible.

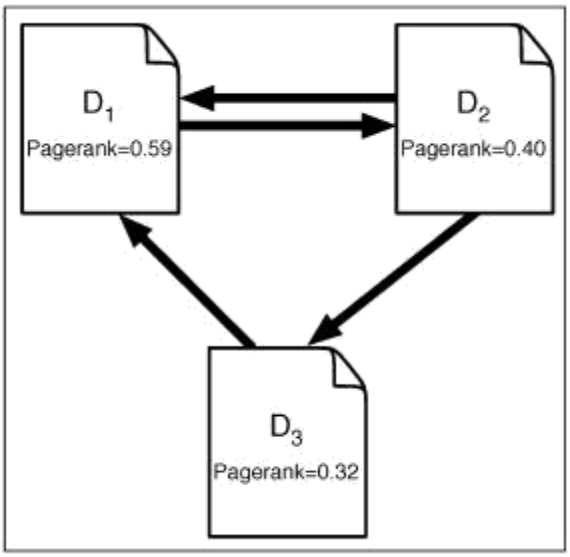
5.6.4 Page Rank

The most popular algorithm for improving Web search is PageRank algorithm (named after Page) was first. It extends the notion of hubs and authorities in the Web graph. PageRank is at the heart of the popular Web search engine, Google. Essentially, the Page Rank algorithm uses incoming and outgoing links to adjust the score of a Web page with respect to its popularity, independent of the user's query. Hence, if a traditional retrieval strategy might have previously ranked two documents equal, the PageRank algorithm will boost the similarity measure for a popular document. Here, popular is defined as having a number of other Web pages link to the document. This algorithm works well on Web pages, but has no bearing on documents that do not have any hyperlinks. The calculation of PageRank for page A over all pages linking to it D1 ... Dn is defined as follows:

$$PageRank(A) = (1-d) + d \sum_{D_1 \ldots D_n} \frac{PageRank(D_i)}{C(D_i)}$$

where C(Di) is the number of links out from page Di and d is a dampening factor from 0-1. This dampening factor serves to give some non-zero PageRank to pages that have no links to them. It also smooths the weight given to other links when determining a given page's PageRank. This significantly affects the time needed for PageRank to converge. The calculation is performed iteratively. Initially all pages are assigned an arbitrary PageRank. The calculation is repeated using the previously calculated scores until the new scores do not change significantly. The example in above Figure, using the common
dampening factor of 0.85 and initializing each PageRank to 1.0, it took 8 iterations before the scores converged.

Simple pagerank calculation



5.6.5Improving Effectiveness of Web Search Engines

Using a Web server to implement an information retrieval system does not dramatically vary the types of algorithms that might be used. For a single machine, all of the algorithms given in Chapter 5 are relevant. Compression of the inverted index is the same, partial relevance ranking is the same, etc.

However, there were and are some efforts specifically focused on improving the performance of Web-based information retrieval systems. In terms of accuracy improvements, it is reasonable to believe that by sending a request to a variety of different search engines and merging the obtained results one could improve the accuracy of a Web search engine. This was proposed as early as TREC-4 . Later, the CYBERosetta prototype system developed by the Software Productivity Consortium (SPC) for use by DARPA, identical copies of a request were simultaneously sent to multiple search servers. After a timeout limit was reached, the obtained results were merged into a single result and presented to the user.