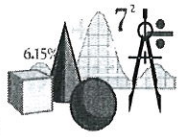
IndexRefGuide



White Paper

Investment Science Corporation

www.investmentsciencecorp.com

Table of Contents

1. INDEXREFGUIDE OVERVIEW 6

1.1 Introduction to IndexRefguide 6

1.2 Running IndexRefGuide and its Associated Agents 7

Figure 1: View Application Page 7
Figure 2: Flowchart 8
Figure 3: Sample Window 9

1.3 Design Goals 9

2. ALGORITHM DEVELOPMENT 10

2.1 The Classic Vector Space Model 10

2.2 Using Salton's Formula for Word Search 10

Figure 4: Sample Computation 11

2.3 Extending Salton's Formula 12

2.3.1. Term Weighting 12
Figure 5: KnowledgeBase in Document Template 12
2.3.2. Query Term Weighting 13

3. SYSTEM ANATOMY 14

3.1 IndexRefGuide Architecture 14

Figure 6: IndexRefGuide Architecture 14

3.2 Helper Agents 15

3.3 AIS Data Structure Used 15

Figure 7: ObjectRepository vs. Index Object 16

3.4 IndexRefGuide Data Storage 16

Figure 8: Sample Documents 16
3.4.1 wordIndex 16
Figure 9: Algorithm to create wordIndex 17
3.4.2 baseFile.txt 18
3.4.3 freqREPOS.db 18
Figure 12: Sample freqREPOS.db 18
3.4.4 wordIndex 19
Figure 13: Statistics for CoreContent 19
Figure 14: Word To Posting List in a Sparse Repository Implementation 20
Figure 15: Two Possible Matrix Implementations 20
3.4.5 docIndex 22
Figure 16: Sample docIndex 22

4. SEARCHING CORECONTENT 23

4.1 searchRefGuide Agent 23
Figure 17: searchRefGuide Algorithm 23

4.2 queryHtml 23
Figure 18: Sample Query 24

5. EVALUATION 25

5.1. IndexRefGuide Evaluation 25
Figure 19: Load Statistics 25
5.2. Other Metrics 26

REFERENCES 27

1. INDEXREFGUIDE OVERVIEW

1.1 Introduction to IndexRefguide

Information retrieval within the context of this project pertains to the search of relevant documents in response to user queries. IndexRefGuide is the AIS agent that indexes the documents prior to user queries. Relevant documents from our static document collection are then retrieved and returned to the user in HTML format. Our static document collection is contained in the CoreContent folder. The CoreContent folder consists of 2,113 XML files.

The algorithm used in IndexRefGuide ranks the resulting documents based on document relevance. Document relevance is computed using the Classic Vector Space Model integrated with a word weighting scheme that is unique to the structure of our AIS documents. This paper will discuss in the following chapters the algorithms and heuristics used by IndexRefGuide store and retrieve document information in an efficient and effective manner.

There are two other agents that run together with IndexRefGuide. These agents are:

- aisRefGuide

This agent acts as a semi-intelligent document manager agent which is able to generate cross referenced and cross indexed AIS HTML documents from the documents initially found in the CoreContent folder to its specified output folder, the wwwroot folder

- alice

This agent is the AIS LISP implementation of the Alice Chatbot originally written by Dr. Wallace. The AIS Lisp version allows Alice to respond to mathematical queries. Such feature is not found in the original Alice version.

There are other agents that are needed to run IndexRefGuide, aisRefGuide and alice. These agents are:

- xml

The XML compiler is a low-level Lisp agent that takes an XML input string and returns a completed XML document tree model.

- index

The index object provides a persistant index object similar to a directory with the several advanced features that includes unique and non-unique index management

- porterStemmer

The porterStemmer is a Lisp agent designed to remove suffixes from a list of words. This agent is based on the aggressive Porter Algorithm.

- browseAgent

The browseAgent agent manages all Agent source code stored in the file cabinet.

1.2 Running IndexRefGuide and its Associated Agents

All the agents mentioned in the previous section reside in the Alice folder within the RefGuide folder. You can download the files through StarTeam. In StarTeam, you can go to the _RefGuide project then proceed to the AliceFolder in the _RefGuide project to download all the files. The Alice folder contains:

- Files:
  - AliceStartup.sl
    Script that loads files to Agent Cabinets and sets the context memory.
  - aisapp.ini and context.ini
    These are the configuration files.

- Subfolders:
  - Bin
    Contains the precompiled binaries of some agents. AliceStartup.sl has already precompiled the xml, index, alice, porterStemmer and browseAgent agents.
  - CoreContent
    Contains AIS Documentation. Users of aisRefGuide put all the new and modified documents in this folder. This folder is the static document collection used by IndexRefGuide.
  - Foundry
    Documents from CoreContent and Templates are moved to Foundry. Documents in Foundry are evaluated by the aisRefguide agent.
  - Foundry
    Documents from CoreContent and Templates are moved to Foundry. Documents in Foundry are evaluated by the aisRefguide agent.
  - Source
    Contains the source code for all the agents in the Alice folder.
  - Templates
    Contains HTML templates to used by IndexRefGuide and aisRefGuide.
  - wwwroot
    Where the output HTML documents and images are placed.

To run these agents, please refer to Figure 2: Flowchart in the next page. After going through all these steps, the user can clidk the View Application button in the webide console as shown below in Figure 1: View Application Page Button .

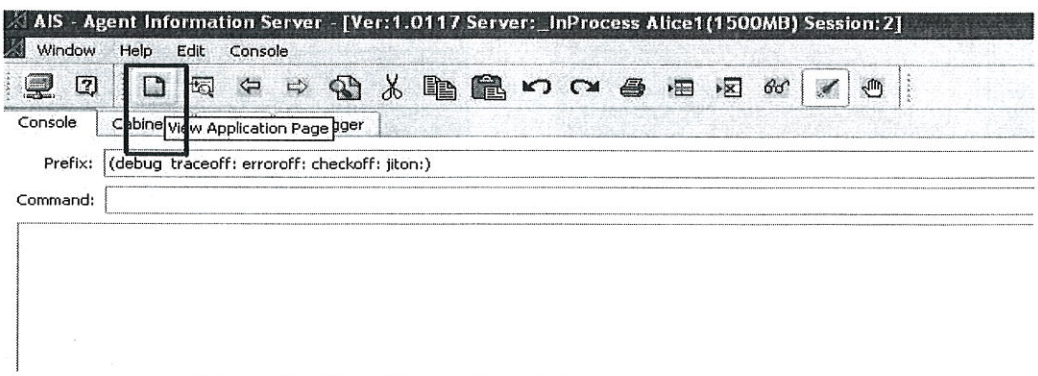


Figure 1: View Application Page

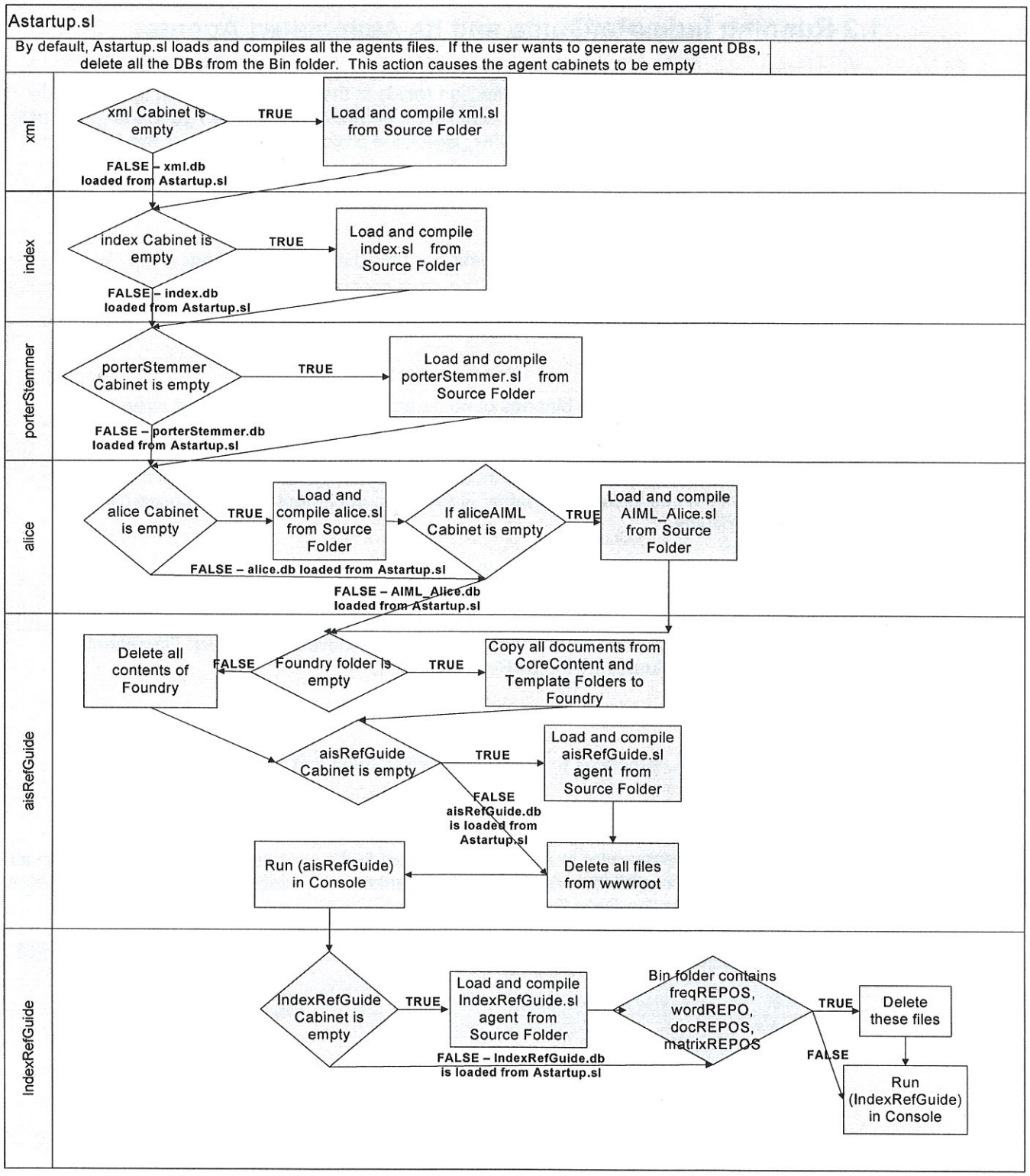


Figure 2: Flowchart

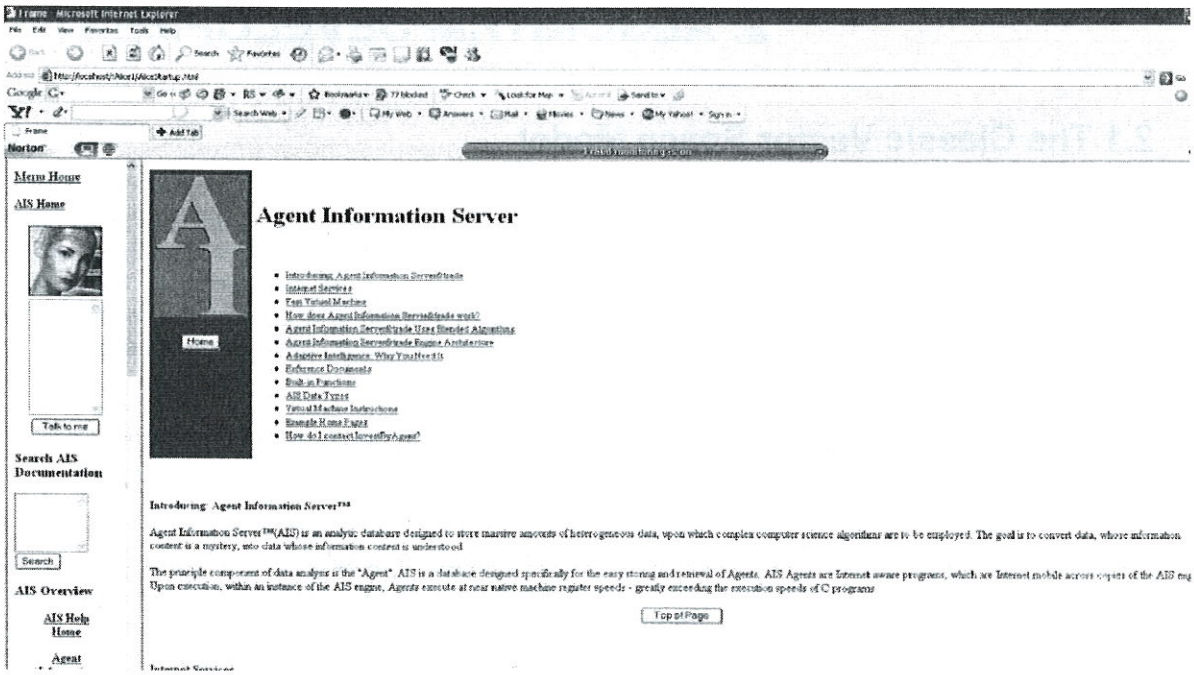


Figure 3: Sample Window

1.3 Design Goals

The design goals of any Information Retrieval System can be summed up in two points: Effectiveness and Efficiency. Algorithms have been used to improve the effectiveness of query processing in terms of precision and recall. There are various retrieval strategies and utilities focused on increasing document relevance. In our previous implementation of RefGuide Indexing as seen in our InvertIndex Agent, we conducted a successful experiment on the use of the Classic Vector Space model to represent documents and queries and the use of Salton's Formula to compute for similarity measures between documents and queries. This algorithm integrated with a word weighting function based on our unique XML Document structure and a word's polysemy count has increased search relevance. Out InvertIndex Agent has given us the frameworK to develop an agent that can effectively search a static collection of documents

We experimented with 2113 documents in CoreContent. These documents contains 10,000 words with 3,000 unique words. IndexRefGuide required 18,624 KB memory on disk and 1500MB Context memory and four hours to index CoreContent. Search time is negligible as InvertIndex was able to search documents fast and effectively.

Efficiency is the issue of run-time performance in terms of index generation and query processing. To address this, the new indexing agent should:
- Compress data saved in memory and on disk;
- Allow for faster access to data saved on disk; and
- Considerably speed up index generation.

To do this we have to:
- Use Integer representation for words
- Minimize looping and checking in of document. Try to get all the necessary information from each document in the least number of passes.

2. ALGORITHM DEVELOPMENT

2.1 The Classic Vector Space Model

The Vector space model computes a measure of similarity by defining a vector that represents each document, and a vector that represents the query. The document vector consists of the terms making up the document and the query vector consists of the terms in the query. Documents whose content, as measured by the terms in the document, correspond most closely to the content of the query are judged to be most relevant, and thus, have the highest similarity coefficient (SC).

Each word is represented by an axis in the vector space.. Each document is treated as a vector, where each coordinate is represented by the frequency of a word.
For example:

> Universal U set of documents consists only of 3 words: "cat", "dog", "mouse"
> Therefore, U is represented as a 3-tuple ("cat", "dog", "mouse")
> Each word is an axis in the vector space Vector (3,1,4) for a document X, will mean that at document X "cat" has a frequency of 3; "dog" has frequency of 1; "mouse" has frequency of 4.

The magnitude for each vector is computed using Phytagorean theorem. If a search word is an element of U, then it can be represented as a unit vector.
For example:

> The search word "cat" will be represented as (1,0,0)

The cosine of the angles formed by the search word unit vector and each of the document vectors are compared. The smaller the cosine of the angle, the more prioritized is the document that will be returned during a search.

2.2 Using Salton's Formula for Word Search

Salton's Formula is based on the classic vector space model. In this formula, document relevance is measured in terms of word weights. Word weights may just be simply word frequency or one can extend the model to incorporate heuristics, probability or any method to make word weight more description so as to return more relevant documents.

$$w_i = tf_i * log\left(\frac{D}{df_i}\right)$$

Where:
w_i = word weight
tf_i= term frequency
D = Total number of documents
df_i = document frequency

Salton's model incorporates local word frequency (or otherwise known as keyword densities) with global information.

Salton's model also includes the similarity function. The higher the similarity, the more relevant the document is to the query, or in other words, the nearer the document vector is to the query vector.

$$\therefore \text{Cosine}\,\theta_{D_i} = \text{Sim}(Q, D_i)$$

$$\therefore \text{Sim}(Q, D_i) = \frac{\sum_i w_{Q,j} w_{i,j}}{\sqrt{\sum_j w_{Q,j}^2}\sqrt{\sum_i w_{i,j}^2}}$$

Below is a sample computation of word weights based on the sample documents. This computation assumes that word weights are based on word frequency.

TERM VECTOR MODEL BASED ON $w_i = tf_i * IDF_i$

Query, Q: "gold silver truck"
D_1: "Shipment of gold damaged in a fire"
D_2: "Delivery of silver arrived in a silver truck"
D_3: "Shipment of gold arrived in a truck"
D = 3; IDF = log(D/df_i)

		Counts, tf_i						Weights, $w_i = tf_i * IDF_i$			
Terms	Q	D_1	D_2	D_3	df_i	D/df_i	IDF_i	Q	D_1	D_2	D_3
a	0	1	1	1	3	3/3 = 1	0	0	0	0	0
arrived	0	0	1	1	2	3/2 = 1.5	0.1761	0	0	0.1761	0.1761
damaged	0	1	0	0	1	3/1 = 3	0.4771	0	0.4771	0	0
delivery	0	0	1	0	1	3/1 = 3	0.4771	0	0	0.4771	0
fire	0	1	0	0	1	3/1 = 3	0.4771	0	0.4771	0	0
gold	1	1	0	1	2	3/2 = 1.5	0.1761	0.1761	0.1761	0	0.1761
in	0	1	1	1	3	3/3 = 1	0	0	0	0	0
of	0	1	1	1	3	3/3 = 1	0	0	0	0	0
silver	1	0	2	0	1	3/1 = 3	0.4771	0.4771	0	0.9542	0
shipment	0	1	0	1	2	3/2 = 1.5	0.1761	0	0.1761	0	0.1761
truck	1	0	1	1	2	3/2 = 1.5	0.1761	0.1761	0	0.1761	0.1761

Figure 4: Sample Computation

2.3 Extending Salton's Formula

2.3.1. Term Weighting

Salton's Fomula can be extended to incorporate heuristics to word weights making word weights a more descriptive measure for document relevance. In the previous implementation of invertIndex agent, the extension of the formula included both XML Document information and word polysemy count. This implementation did not include word polysemy count since the Thesaurus Agent has not been incorporated in the IndexRefGuide agent. IndexRefGuide only includes XML Document information in the word weights.

$$w_i = tf_i * log\left(\frac{D}{df_i}\right) * \text{wXML}$$

Where:
wXML = Added word weight due to Hierarchy of the word in the XML Structure

Document terms are weighted in wordWeight Agent. Each document in the CoreContent folder follows a template. A Document may follow the Document, Essay, Function, Example or VMInstruction Template. These Templates are found in the Templates folder. All templates have a Knowledge Base where pertinent information are stored to make document searching easier. Words found inside a document's knowledgebase are given more weight than terms not found inside the knowledge base.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE Document SYSTEM "../DTD/Document.dtd">
<Document>
    <KnowledgeBase>
        <Title>Index Agent</Title>
        <Topic>Lisp</Topic>
        <SubTopic>Data Management</SubTopic>
        <HumanKeywords>Database Data Management Agent Programming</HumanKeywords>
    </KnowledgeBase>
    <Essay>Essay_Index_Object</Essay>
</Document>
```

Figure 5: KnowledgeBase in Document Template

The wordWeight agent will give the following additional points for words found inside the KnowledgeBase:

MaxWeight of 80 points is given to the first child.
MaxWeight is decremented by 10 points for each child hereafter.

Therefore, "Index" "Agent" in the Title gets an additional 80 points each."Lisp" gets 70. "Data" "Management" gets 60 and so on and so forth.

2.3.2. Query Term Weighting

Query terms are normalized before additional weights are added to them. The formula for normalized query term frequency is a variation of the query term normalization formula. The Original query term normalization formula is:

$$qf_{q,i} = 0.5 + 0.5 * tf_{q,i} / max\ tf_{q,l}$$

Where
$qf_{q,l}$ = normalized query term frequency
$tf_{q,l}$ = frequency of term i in query q
$max\ tf_{q,l}$ = maximum frequency of term i in query q

This formula is modified to be:

$$qf_i = tf_i / total\ tf_i$$

Where
$qf_{i,}$ = normalized query frequency
$tf_{i,}$ = frequency of query term i in document collection
$total\ tf_i$ = total frequency of all the query terms in the document collection

The modification is based on the frequency of the query term in the document collection an not on the query itself. Then we penalize query terms that occur more in the document collection. This is based on the heuristic that less commonly occurring terms can result to more relevant documents.

$$qw_i = ((qf_i + tf_i) - 1)$$

Where
$qw_{i,}$ = query term weight
$qf_{i,}$ = normalized query frequency
$tf_{i,}$ = frequency of query term i in document collection

3. SYSTEM ANATOMY

3.1 IndexRefGuide Architecture

Words in the document collection are pre-processed by the child agents: defaultlexer, porter Stemmer and wordWeight. After pre-processing each of the words in each of the documents IndexRefGuide then goes to its two major agents. This child agent generateLexicon is responsible for generating a word-to-integer mapping. All the words in this mapping are stemmed and lexed. This means that these words do not contain suffixes and do not contain special characters. This word-to-integer mapping is saved in an index object that is in turn saved on disk in wordREPO.db. This mapping is used throughout the IndexRefGuide agent to reference words. The agent generateLexicon also generates the inverted index saved in a file called baseFile.txt. The same agent also generates a freqREPOS.db that saves a word and the number of times that word appears in the document collection.

The second main agent generateMatrix is responsible for preparing the information gotten from generateLexicon for query computation. To do that, generateMatrix opens baseFile.txt and converts the file format of the invert index into an AIS Record data type. This Record data type is a matrix representation of docID x wordID. This AIS Record contains the document IDS as rows and the words as columns and its word weight per word in a document as values. To compute the values, generateMatrix makes use of freqREPOS.db. The agent generateMatrix also generates a docIndex that saves each word and a list of the documents where that word appears.

SearchRefGuide is a child agent of IndexRefGuide. This agent is responsible for computing for the Similiarity Coeffifient of all the documents based on the query terms. The higher the SC, the more relevant the documents. To do this, searchRefGuide accesses wordIndex, freqREPOS, docIndex and matrixREPOS.

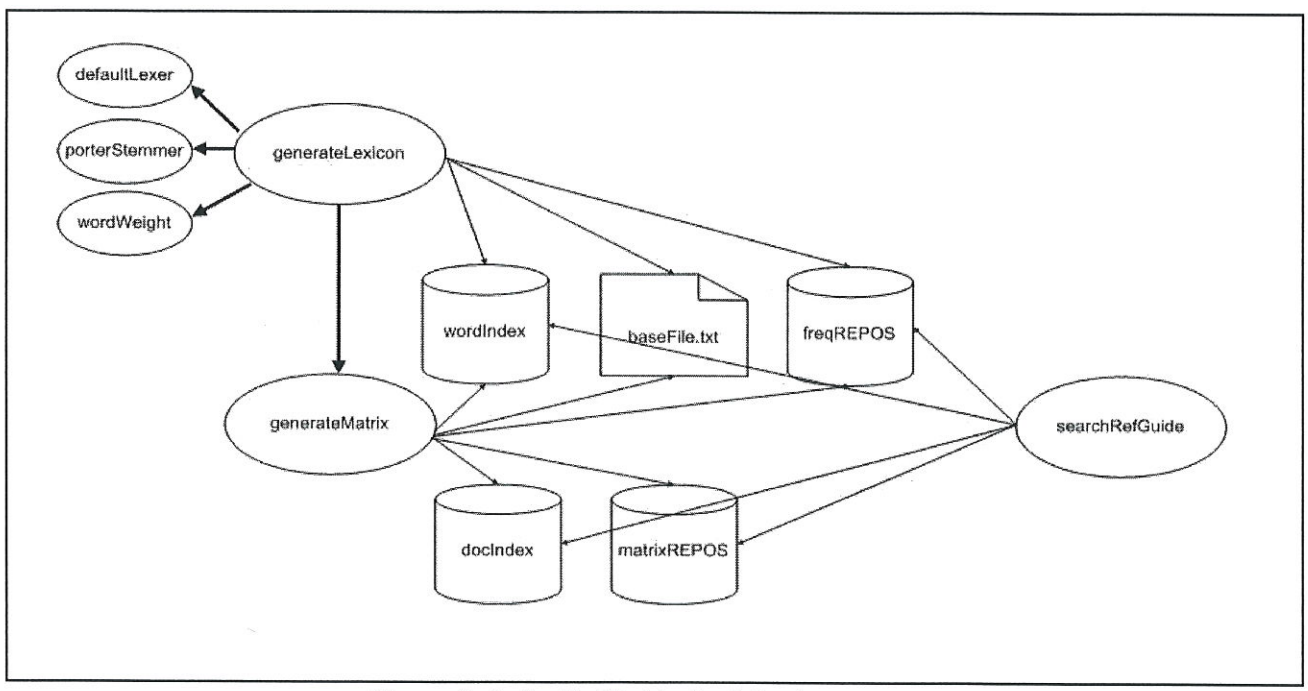


Figure 6: IndexRefGuide Architecture

3.2 Helper Agents

There are two main helper agents used in IndexRefGuide. The first on is the poreterStemmer Agent. It is based on the porterStemming algorithm. To run this agent, see syntax below:
(porterStemmer outputOption)

The outputOption may be 0 or 1. The ouputOption 0 returns a vector of stemmed string. While outputOption 1 returns a strring of stemmed strings delimited by whitespaces. IndexRefGuide uses ouputOption 1.

The porterStemmer also has the option to include the HTML tags in stemming. This option can be set as: (setq porterStemmer.htmlTagsOn TRUE/FALSE). By default htmlTagsOn is FALSE. If set to TRUE, the html tags are stemmed. If set to FALSE, only contents between html tags are stemmed.
Example: <Name> Test </Name>
If set to FALSE: Output: "test"
If set to TRUE: Output: "<Name> test </Name>")

In the IndexRefGuide agent documents are stemmed without removing the HTML tags. And the result is saved as a string in its pvars.

The other helper agent is the child agent defaultLexer contained in IndexRefGuide. This is based on the parseAgent.defaulLexer agent. This agent converts an input string into a vector of recognized lexemes. But to minimize external agent calls which slows down performance time, a local defaultLexer is made. This returns a directory of unique words and its frequency in the document.

3.3 AIS Data Structure Used

The types of data structure are used in IndexRefGuide, namely: files, index object and repositories.

• Data in **files** are stored as bytes. The data can also be reference though in a file through a charpointer. This makes data access fast. Offset address computation. Read sequentially but for retrival of a specific value , an Index Object or an Object Repository is faster.

• The **index object** is an instance of the index agent. An index object is saved in a repository and is created though the commands:
;; To create the repository where the Index Agent is saved
(setq newIndexREPO (new ObjectRepository: "newIndexREPO.db"))
(clear repos)
;; To create a non-unique Index Object on disk
(setq newIndex (new index newIndex: newIndexREPO create:))

• An **Object Repository** is a Heap object containing zero or more bindings. Each binding is composed of an AIS Word data value and followed by an object reference key, allowing ObjectRepository values to be referenced and modified by object key. Unlike an Index Object where an index key allows you to reference an index value that is a directory of values, an Obect Repository can allow you to directly store and reference a key-value pair of any type and complexity.

It is noteworthy to mention that an Object Repository allows for faster retrieval of keys and values but loading or opening an object repository is expensive in terms of context memory. An Index Object, on the other hand, can save you memory but has slower retrieval compared to object repositories. Figure 4 below shows the result of the experiment on using Index Object or ObjectRepository to store Word-To-Integer Mapping.

Total Keys: 457
Total Number of words: 4,812

	Index Object	Object Repository
Size on Disk	260KB	1,998KB
Query Retrieval	1.001 seconds	.808 seconds

Figure 7: ObjectRepository vs. Index Object

The decision on whether to use an Object Repository or an Index Object depends on the current needs of the program. If the program need speed and can afford to sacrifice memory, the an Object Repository is ideal. But if the program cannot afford to sacrifice memory but still can take a performance hit, then an Index Object may be used.

3.4 IndexRefGuide Data Storage

In the previous implementation of InvertIndex agent, the invert index was represented as a vector of vectors stored in an index object. This representation is straightforward and easy to implement but the drawbacks are: it is memory intensive and it takes a long time to generate the Object Repositories. To remedy this, here are the 5 data storage variables used in IndexRefGuide.
To better illustrate the algorithms, we will use a set of sample documents in all our examples. This is the same set of sample documents used in Figure 4 in the previous chapter.

Doc0	Shipment of gold damaged in a fire.
Doc1	Delivery of silver arrived in a silver truck.
Doc2	Shipment of gold arrived in a truct

Figure 8: Sample Documents

3.4.1 wordIndex

wordIndex is an Index Object saved on disk as wordREPO.db. This contains the word-to-integer mapping of all the unique words in the documents. The keys contain the first two letters or symbols of the word the directory value contains the word and its integer wordID. Words are numbered sequentially from 0. This sequential number of each word is its wordID. Its wordID is called its absolute word index (AWI).

Below is the algorithm to construct wordIndex

```
1. Create wordIndex
2. Loop through all the documents: get new document
3.      Loop through all the words in the document: get new word
4.              Stem the word
5.              Lex the word
6.              Check if the word is a junkword
7.              If junkword go to 3.
8.              If not junkword
9.                      Save the first  two letters of the word  as its wordKey
10.                     If wordKey exists in wordIndex check if word is a member of
                        the wordIndex directory
                                If word exist, goto 12.
                                If word does not exist, add word as directory key
                                and 0 as directory value, goto 12
11.                     If wordKey does not exist in wordIndex, create a Directory and assign
                        the word as the directory key      and 0 as directory value, goto 12
12.             If there are still word in the document, goto 3.
13.             If there are no more word in document, goto 2.
14. If there are no more documents, goto 15.
15. Set wordCount=0
16. Loop through all the keys in wordIndex, get new key
17.     Loop thgouh all the directory keys in wordKey
18.             Assign wordCount as directory value for the word
19.             Increment wordCount
20.             If there are still directory keys, goto 17
21.             If there are no more directory keys, goto 16.
22.     If there are no more keys in wordIndex, end loop
END.
```

Figure 9: Algorithm to create wordIndex

After passing and lexing, our documents would look like:
D0 = {damage:1 fire:1 gold:1 ship:1}
D1 = {arrive:1 deliver:1 silver:2 truck:2}
D2 = {arrive:1 gold:1 ship:1 truck:1}

Applying the algorithm above, our wordIndex would look like this:

wordIndex Key	wordIndexValue
ar	#{dir|| arrive: 0 }
da	#{dir|| damage: 1}
fi	#{dir|| fire: 2}
go	#{dir|| gold: 3}
sh	#{dir|| ship: 4}
si	#{dir|| silver: 5}
tr	#{dir|| truck: 6}

Figure 10: Sample wordIndex

3.4.2 baseFile.txt

The baseFile.txt is the file representation of our forward index. This data structure is used to save the stemmed and lexed words an their occurrences in document. This will prepare for the construction of the inverted index. By storing data in baseFile.txt will cut down execution time since we do not have to traverse each document repeatedly agent to get the necessary information when we jump to next agent.

Using the previous Sample document in Figure 7, here is the generated baseFile.txt

```
29      4       damage 1 fire 1  gold 1 ship 1
35      6       arrive 1 deliver 1 silver 2 truck 2
30      4       arrive 1 gold 1 ship 1 truck 1
```

Figure 11: Sample baseFile.txt

The first integer is the byte count of the words and their weights. The second integer is the total number of word weights in a document. Then the forward index is represented in the file as a word with its corresponding word weight.

Please note that unlike figure: 6, the contents of baseFile.txt is not the word frequency but the word weight. This means that the word frequency has been added with preassigned number. More of word weighting will be discussed in wordWeights.

3.4.3 freqREPOS.db

This contains the word and the total weight of the word based on the whole document collection. If we assume that in our Sample Documents that our word frequency is equal to our word weight, then our freqREPOS based on the Sample Documents in Figure 7 will look like this:

freqREPOS key	freqREPOS value
arrive	2
damage	1
fire	1
gold	2
ship	2
silver	1
truck	2

Figure 12: Sample freqREPOS.db

3.4.4 wordIndex

The document and word information should be organized as an inverted index. An inverted index contains the word as a key and a posting list as its value. A posting list contains the list of all the documents that a word appears in and the number of times the word appears in that document.

An experimented was conducted to find out what representation of the inverted index is more efficient in terms of memory and performance. But before that experiment can be done, some information on the kind of documents and words contained in CoreContent is needed. Below are some statistics on CoreContent. The table below is a histogram containing the number of documents under the column ***Document Interval*** and the ***Frequency*** column means the number of times a word appears in a give document Interval.

Document Interval	*Bin*	*Frequency*	*Percentage*
1	1	0	
2-10	10	4123	85.60%
11-50	50	525	10.90%
51-150	150	120	2.50%
151-300	300	33	
301-500	500	5	
501-750	750	3	
751-1000	1000	2	
1001-1250	1250	0	
1251-1500	1500	0	
1501-1750	1750	0	
1751-2111	2000	0	

Total Number of Documents: 2,109
Total Number of unique words: 4,812
Total Number of word excluding junkwords and unstemmed words: 10,000

Figure 13: Statistics for CoreContent

The statistics show that there are no same words that appear only in a single document or in a thousand or more documents. Majority of the words appear in 2-10 documents.

This means that our inverted index is sparse in nature. Even with our Sample Documents in Figure 7, we can see that the inverted index is sparse as well. Below is a representation of the Sample Documents as a sparse repository.

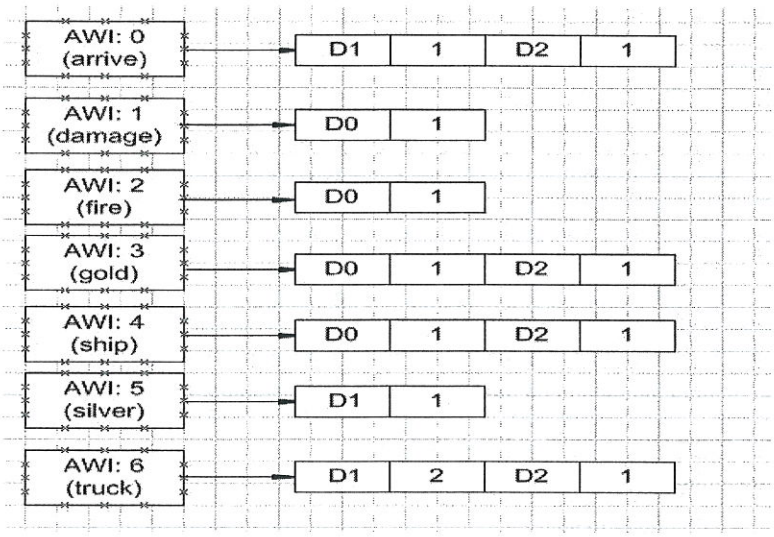


Figure 14: Word To Posting List in a Sparse Repository Implementation

One of the advantages of a sparse repository is that we can conserve memory, in theory. After implementing the sparse repository, the total size on disk of sparseREPOS.db is 40,000KB. One obvious drawback of this implementation is the complex and time-consuming retrieval processing for retrieving an entry in the posting list. The program has to access sparseREPOS.db and traverse the posting list multiple times to get the desired data.

Another option is to save the inverted index as a matrix. Thoeretically, the size of the matrix on disk would be more or less 40,000KB given 4,812 unique words multiplied with 2,109 document and given that each integer value is 4 bytes. This will be the same storage requirement needed for sparseREPOS.db, in theory. Upon constructing the matrix whether it be Implementation A: or B: as seen in Figure below, the matrixREPOS.db size is 10,803KB.

There are two ways we can represent an inverted index matrix. We can have it the words as rows and documents as columns and vice versa.

	D0	D1	D2
AWI:0 (arrive)	0	1	2
AWI:1 (damage)	1	0	0
AWI:2 (fire)	1	0	0
AWI:3 (gold)	1	0	1
AWI:4 (ship)	1	0	1
AWI:5 (silver)	0	1	0
AWI:6 (truck)	0	2	1

A: ***word x document***

	AWI:0 (arrive)	AWI:1 (damage)	AWI:2 (fire)	AWI:3 (gold)	AWI:4 (ship)	AWI:5 (silver)	AWI:6 (truck)
D0	0	1	1	1	1	0	0
D1	1	0	0	0	0	1	2
D2	2	0	0	1	1	0	1

B: ***document x word***

Figure 15: Two Possible Matrix Implementations

To store and retrieve a matrix, the matrix information is first stored in a record. The Record data type is chosen as the intermediate data storage since a record is a lean data structure as compared to object Repositories and Directories. Moreover, accessing an element in a record is easy and fast since you do not have to do offset address calculation as you will have to do if the information is stored in a file. After all the data is in the record, the record is read and the information is stored in a repository row by row. In implementation A: the record data type will look as follows:

```
(setq newRecord (new Record: totalWords  wWeight:Float:totalDocs ))
```

Where:
totalWords = number of columns
totalDocs = number of column repetition

In Implementation B:, the record data type will look as follows:

```
(setq newRecord (new Record: totalDocs  wWeight:Float:totalWords ))
```

Where:
totalDocs = number of columns
totalWords = number of column repetition

There are two important facts to consider in choosing the kind of matrix implementation. The first one is that we can only do row retrieval of a matrix efficiently since we are storing and saving our matrix row by row in an object repository. We can only retrieve our information by row. We cannot do column retrieval. Another important fact to consider are the needs of our search algorithm. The search algorithm needs:

Given a) the list of all the documents and all the words and its corresponding frequency found in the document

Given b) the list of all the words and the documents where the word can be found.

Based on these facts, we can see that neither the matrix implementation A: nor B: in Figure above is enough fro our data requirement. Implementation A does not have Given a) above but it has Given b). Given a) can be obtained by doining a matrix transpose. Implementation B: has Given a) but it does not have Given b). Given b) can be obtained by also doing a matrix transpose or by storing Given b) in another data structure since the data in Given b) are all integers so it will not be that expensive to create a new structure.

Doing a matrix transpose is a complex and not to mention an expensive procedure in terms of memory and performance. Therefore, Implemnetation B: is chosen and a docIndex index object will be created to store Given b).

3.4.5 docIndex

Since docIndex is an IndexObject, its value is automatically a vector. But its key can be any AIS Data type. In IndexRefGuide, the docIndex key is an integer reprenting a word's AWI and its value is a Vector of DocIDs reprenting the documents where the word is locatedIn our Sample Documents our docIndex will look like this:

docIndex key	docIndex value
0 arrive	#(1 2)
1damage	#(0)
2 fire	#(0)
3 gold	#(0 2)
4 ship	#(0 2)
5 silver	#(1)
6 truck	#(1 2)

Figure 16: Sample docIndex

4. SEARCHING CORECONTENT

4.1 searchRefGuide Agent

The agent searchRefGuide is a child agent of IndexRefGuide. This agent accesses the indices and repositories generated by generateLexicon and generateMatrix to compute for the document similarity for each query term. It returns a directory ranking the documents in ascending order based on the similarity coefficient.

Below is the algorithm used in searchRefGuide:

```
1. Loop though all the words in the query
2.      Stem the word
3.      Lex the word
4.      If word does not exist in wordREPO goto 1
5.      If word exists in wordREPO
6.              Normalize word.
7.              Compute for wordWeight = tf * IDF
8.              Get all the docs containg the word from docIndex
9.              Save wordWeight and docs for the word in queryDirectory
10.             Save all the docs in a docVector
10.     If there are still words in the query, goto 1.
11.     If there are no more words in the query goto 12.
12. Loop through all the docs in docVector
13.     Loop though all the keys in queryDirectory
14.             Get the word weight of the query term in the doc though matrixREPOS
15.             Get the document magnitude of the doc in matrixREPOS
16.             Compute for document similarity
17.             Save document and document similarity in simDirectory.
18.             If there are still keys in queryDirectory, goto 13.
19.             If there are nore more  keys in queryDirecotry, goto 12.
20.     If there are still docs in docVector, goto 12.
21.     If there are no more  docs in docVector, return simDirectory.
END
```

Figure 17: searchRefGuide Algorithm

4.2 queryHtml

The user can run (IndexRefGuide.searchRefGuide {seach terms}) in the console. Or the user can use the Web interface to search RefGuide. Figure 2 in the first chapter has showed us in detail how to construct the indices for IndexRefGuide. After going though all these steps in the flowchart, the user can click the View Application Page button to lauch IndexRefGuide and its other associated agents.

The IndexRefGuide examples demonstrates how webide communicates with the AIS (also known as the Smartbase Engine) using HTML and JavaScript. IndexRefGuide uses an HTML web page containing forms that require the user to enter an search terms and press a "Submit"

key. The Internet Explorer web browser detects the key press action and triggers a **navigate** event. The HTML language has provisions to permit a developer to interrupt the **navigate** event and redirect the browser to run an application, such as **IndexRefGuide**. Once **IndexRefGuide** receives the input query terms, it calls on **IndexRefGuide.queryHtml** to perform the search in CoreContent. The output webpages containing the result of IndexRefGuide.queryHtml are then placed in the output folder, wwwroot. The agents write the result to a file as HTML tagged instructions and returns the URL of the output to the web browser and the web browser navigates to it.

Below is the window for the aisRefGuide, Alice and IndexRefGuide.

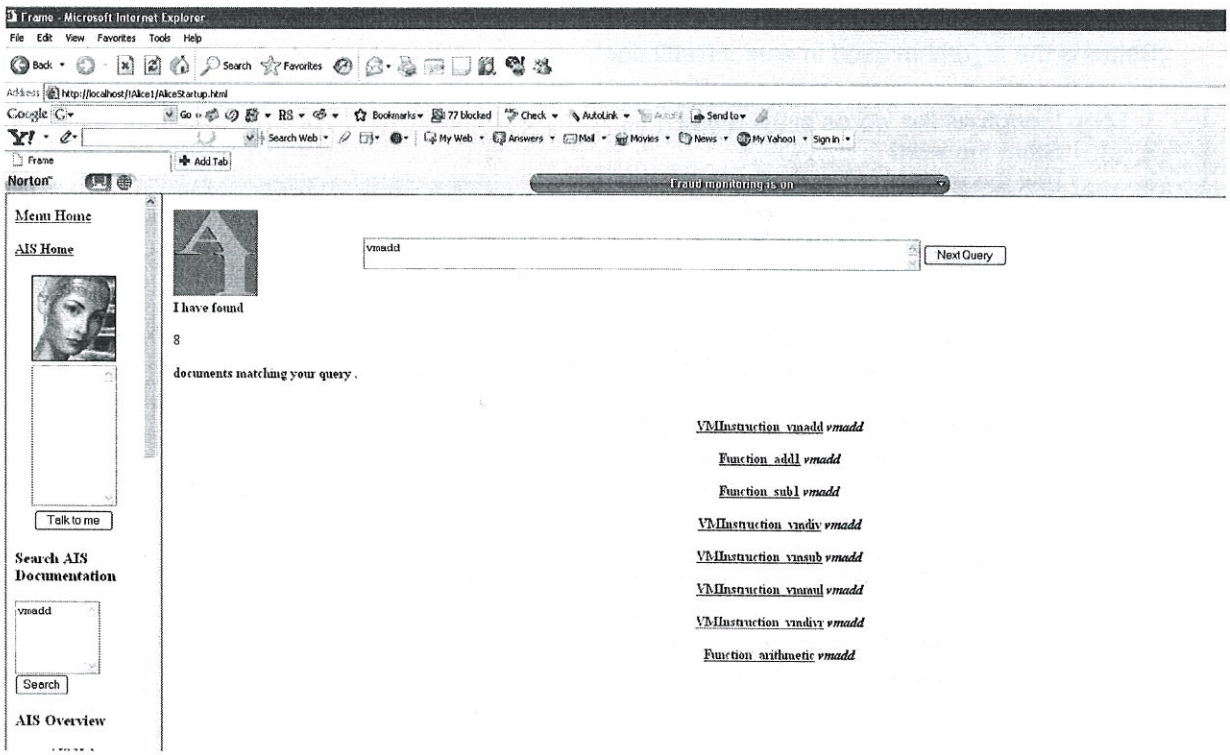


Figure 18: Sample Query

5. EVALUATION

5.1. IndexRefGuide Evaluation

The first step in evaluating RefGuide is to evaluate the scalability of its design. Scalability refers to the ability to either handle growing amounts of data in a graceful manner, or to be readily enlarged so as to accommodate additional amounts of data.

To measure Load Scalability as number of words and documents increases, load statistics are computed using the simple Ratio and Proportion Formula.

$$\frac{\text{RefGuide Document Size} \times \text{RefGuide Unique Words}}{\text{Data Structure StorageSize}} = \frac{\text{TestData Document Size} \times \text{TestData Unique Words}}{\text{Data Structure StorageSize}}$$

Using this formula, we are assuming that the relationships between all the variables are line To come up with the figures for testData, we assume for testData that there are on the average 14 unique words per document, that the increase in repository size is directly proportional to the increase in the number of words at a constant number.

	RefGuide	TestData
Document Size	2,113	1,000,000
Total Word Size	10,000	20,000,000
Number of Unique Words	4,812	14,000,000
wordIndex size	259 KB	35 GB
freqREPOS size	231 KB	31 GB
docREPOS size	831KB	114 GB
matrixREPOS size	10,803 KB	1500 GB
baseFile.txt	866KB	119 GB

Figure 19: Load Statistics

If we reach these numbers, another consideration is our context memory of 1500MB. We cannot load all the indices and repositories of these sizes all at once and still run on 1500MB memory. A solution is to divide repositories and indices into barrels of smaller sizes and load these barrels as needed.

To evaluate IndexRefGuide, we also need to look into the System Performance. System performance can be divided into Index Generation and Query Performance. The total time it takes for IndexRefGuide to run and generate all the files and repositories from CoreContent is 10 minutes in an AMD Turion FX60 X2 Machime with 4GB of RAM. The average time it takes for IndexRefGuide to return a webpage of query results is 1.2 Seconds still using the same machine.

5.2. Other Metrics

Both Scalability and System Performance pertain to a system's efficiency. To measure a system's effectiveness, precision and recall should be computed to get the F-value. An F-value measures a test accuracy. To get the F-value of IndexRefGuide, we need to construct a series of query words, categorize all the documents in CoreContent as either relevant or irrelevant in terms of the query word and get the F-value based on these formulas:

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved document}}{|\{\text{retrieved documents}\}|}$$

$$\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

$$F = 2 \cdot (\text{precision} \cdot \text{recall})/(\text{precision} + \text{recall}).$$

References

[1] Brin, Sergey and Page, Lawrence. The Anatomy of a Large-Scale Hypertextual Web Search Engine.
[2] Grossman, David and Frieder, Ophir. Information Retrieval:Algorithms and Heuristics (2^{nd} Edition).
[3] Petralba, Felipe Jr. CoreContent Indexing.
[4] Term Vector Theory and Keyword Weights
http://www.miislita.com/term-vector/term-vector-1.html
[5] Wikipedia
http://en.wikipedia.org/