1. AltaVista™ – an example for inverted index

AltaVista™ can be used as an example application of inverted index.

In the year of 1998, AltaVista was able to index about 0.8 Tera-Bytes of text, and it indexed every word, ie, there are no stop words. The server needed to support about 37 million queries on weekdays. However, the mean response time and the median response times are still kept as low as 0.6 second and 0.9 second respectively as observed in the experiments in New Jersey.

The reason that AltaVista is so quick attributes to the conceptually simple algorithms (mainly inverted index) that it uses besides its advanced hardware composition. It concatenates all the documents into one document and special token is used in this concatenated document to separate the original documents. Flat inverted file is used as the basic data structure. For each word, a delta-encoded list of all the locations where that word occurs is stored in this file. Lexicon is front-coded to saving the storage space, and this makes the size of full-text index is only around 30% of the input size.

The basic operators of AND, OR, AND NOT, NEAR, PHRASE are manipulated to provide different match modes for the query, which includes exact match, prefix match and phrase match. In order to achieve this, they use only one abstract data type. Thus, for any query, all the locations that match that query are ordered into one list. This list is called *index stream reader* (ISR). The basic operations then can be used to process these ISR. These basic operations include:

- loc( ): return current location
- next( ): advance to the next location
- seek(int k): advance to first location $\geq k$

Since ISR is the only abstract data type, query can be done by translating it into an ISR and then manipulate it using the following elementary operations:

- CREATE: WORD → ISR …….Create a list
- OR: ISR x ISR → ISR …….Merge lists
- AND: ISR x ISR → ISR …….Intersect lists
- AND NOT: ISR x ISR→ ISR …….
- NEAR: ISR x ISR → ISR ……
- PHRASE: ISR x ISR x … x ISR → ISR ……...

Finally, the resulting ISR is translated back into list of documents and ranked as normal query results.

The hardware that they used in AltaVista as of year 1998 included about 20 64-bits computers, each computer has 10 CPUs working at 625MHz, 12 GB RAM and 300 GB RAID storage.

2. Signature files

Signature files is an important alternative method for inverted index. Although it's rarely used in web search engines, it has been used in a variety of applications, ranging from textual databases such as news databases to multimedia office filing to chemical

databases for DNA matching [1]. One feature of signature file is that it does not need lexicon so that it need not keep the huge lexicon in the memory.

Using this method, each document will have one data entry (signature) index structure (the signature file). This data entry is calculated as following. First, for all the words (term) in a document, a hash function is used to hash these words into a series of bit-vectors. Than these bit-vectors, each representing one word of that document, are logically or-ed to form that document's signature in the signature file.

There are three main issues related to using signature files:

1. Generating signatures
2. Boolean logic on signatures
3. Accessing signatures

2.1 Generating signatures

Let's use *W* to represent the width of the signature of each document. Since the signature of the document is formed by the logical-or of the hashing string of all the terms in it, it's naturally that people also set the hashing string's width to be *W*. Out of these *W* bits of every hash string, *b* bits need to be set to 1 while others are set to 0. Typically, people set W to be the range of 1,000 to 10,000. (In the following text we will explain that the value of *W* actually relates directly to the probability of false matching. So there is a tradeoff between the storage space and the accuracy of the match.)

To generate the hash string of a term, follow function is called to set the *b* bits to be 1.

```
for i = 1 to b
    sig[hash_i(term)%W] = 1;
```

Each $hash_i(\ )$ in the above pseudocode represents one hash function.

The signature of the document can then be generated by logically or the hash strings of all its terms.

As an example, consider the list of terms from the nursery rhyme "Pease Porridge Hot". The corresponding hash strings of all the terms in the document is shown in the following table where *W*=16 and *b*=3.

Term	Hash String
cold	1000 0000 0010 0100
days	0010 0100 0000 1000
hot	0000 1010 0000 0000
in	0000 1001 0010 0000
it	0000 1000 1000 0010
like	0100 0010 0000 0001
nine	0010 1000 0000 0100
old	1000 1000 0100 0000
pease	0000 0101 0000 0001
porridg	0100 0100 0010 0000
pot	0000 0010 0110 0000
some	0100 0100 0000 0001
the	1010 1000 0000 0000

The reason that for certain hash string (such as the one for "hot"), the number of bits that are equal to 1 is less than *b* is because a term may get hashed to the same location by two hash functions.

Assume that each line of the rhyme is a document, then using the above table, it easy to get the descriptors of all the documents by doing the logical-or operation of all the hash strings of its terms. The table is shown here.

Document	Text	Descriptor
1	Pease porridge hot, pease porridge cold,	1100 1111 0010 0101
2	Pease porridge in the pot,	1110 1111 0110 0001
3	Nine days old.	1010 1100 0100 1100
4	Some like it hot, some like it cold,	1110 1110 1010 0111
5	Some like it in the pot,	1110 1111 1110 0011
6	Nine days old.	1010 1100 0100 1100

2.2 Query logic on signature files

From the formation of the signature file of a document, it's obvious that a document contains term *T* only if all the bits that are set by *T*'s hash string are also set in the document's signature. However, a document's signature has all these bits set does not necessarily mean that *T* appears in that document since the set bits can be combined by one or more other terms. The following table shows the relationship between whether T appears in the document and the corresponding bits in the signature.

Query	All bits set	Some bits missing
T	Maybe	No
Not T	Maybe	Yes

This also means that we have to use three-value logic to manipulate the query. Let *N*, *M* and *Y* to represent *No*, *Maybe* and *Yes* respectively. And the resulting AND, OR and NOT operation over this three-value logic is shown in the following table:

AND	N	M	Y
N	N	N	N
M	N	M	M
Y	N	M	Y

OR	N	M	Y
N	N	M	Y
M	M	M	Y
Y	Y	Y	Y

NOT	
N	Y
M	M
Y	N

Using these operations, we can now do basic query on the signature files. For example, if we want to query "some OR NOT hot) AND pease", we will have the following results:

Doc.	S	H	P	NOT H	S OR NOT H	(S OR NOT H) AND P
1	M	M	M	M	M	M
2	M	M	M	M	M	M
3	N	N	N	Y	Y	N
4	M	M	N	M	M	N
5	M	M	M	M	M	M
6	N	N	N	Y	Y	N

As shown in the above, the query result contains not only "*yes*" and "*no*", but also "*maybe*". This means that the result will probably introduce some false matches. It turns out increasing W can decrease the expected number of false matches, and the following calculation explains it.

Let

b = bit slices perquery

q = terms per query

z = expected number of false matches

N = number of documents

p = probability that a random bit in a document's descriptor is on

f = number of (term, document) pairs

W = width of the signature

B = average on-bits per document.

It is obvious that B can be calculated as $B = \frac{f}{N} \times \frac{b}{q}$. Since we assume that the bit is randomly chosen, the probability of that randomly chosen bit to be 0 is $(\frac{W-1}{W-1})^B$. Thus, we have

$$p = 1 - (\frac{W-1}{W})^B \quad (1)$$

A false match requires the bit slices of that query are all corresponding to on-bit in the document's descriptor, and there exist N documents. Thus we have

$$z = p^b \times N \quad (2)$$

Substitute equation (1) to (2), we can have calculate W based using p as shown in equation (3).

$$W = \frac{1}{1 - (1-p)^{\frac{1}{B}}} \quad (3)$$

Equation 3 can be used to calculate how long the signature should be if we want to keep the query result above certain accuracy.

2.3 Accessing signatures

One naïve way to check the whether a document contains a query term is to fetch the descriptors of all the documents. For each document, the descriptor is compared with the or-ed hash string of the query to see whether for each on-bit of that hash string, the descriptor has its corresponding bits set. This actually implies to read and search the entire signature files thoroughly, which is not affordable in practice.

A way to improve the efficiency is based on the observation that in most of cases, the number of on-bits is by far smaller than the length of the signature. This means that if we can access columns of the signature files instead of rows, only the columns that corresponding to the on-bits of the query need to be checked. This can save huge accessing time. This technique is called bit-slicing. In order to achieve this technique, signature files have to be stored on disk in transposed form. That is, the signature file is

stored in column-major order. Suppose that W is the length of the signature and q is the average on-bit number of a query. It's very clear that by using this technique, the data to be accessed is only about $\frac{q}{w}$ of the naïve method.

Still take the above rhyme as an example. Suppose that we want to query the presence of "the". As the hash string of "the" is 1010 1000 0000 0000, only the first, third and fifth columns of the signature files need to be checked. It will return a vector $[011111]^T$, which gives the possible match.

2.4 Example: query on TREC

As an example, let's consider the TREC (Text Retrieval Conference) database. There are about 750,000 documents in this database. The total (document, term) pairs are 137M. Suppose b=8 and W=7200. Then in order to query on a term, it will read 8 slices from the signature files, each slice is 750,000 bits. The total data to be accessed for each one-term query is thus about 750,000 bytes.

Using the above number, we can calculate the expected false hit for the query. In this case, f=137*10^6, N=0.75*10^6, W=7200, b=8. Thus, we have

$$B = \frac{f}{N} \times \frac{b}{q} = \frac{137 \times 10^6}{0.75 \times 10^6} \times \frac{8}{1} = 1470$$

$$p = 1 - (\frac{W-1}{W})^B = 1 - (\frac{7200-1}{7200})^{1470} = 0.185$$

Thus, the expected false hit

$$z = p^b \times N = 0.185^8 \times 0.75 \times 10^6 = 1$$

Note that the total space required for TREC is:

$$7200 \times 750000 / 8 = 675 \times 10^6$$

3. Latent Sementic Indexing (LSI) and Singular Value Decomposition (SVD)

LSI uses SVD to project documents into lower (k) dimensional subspace whereby one can perform queries more efficiently. The Goal of LSI is to find documents that have similar content but not necessarily the same keywords. (*e.g.* query with *automobile* will pick up documents with *car* or perhaps *driver*).

	Document
T	
e	
r	
m	

Our matrix representation, denoted by A, where $a_{i,j}$ is the weight of term I in document j. As of 2001, there are around two hundred million documents and about three hundred thousand terms on the Web.

Now to build an *Index*]we use SVD on the matrix formed by all the term vectors on which we drop all but the highest *k* rows of SVD. This effectively reduces dimensionality of space to *k* dimensions. To perform a *query* we first transform the query to a k-dimension space using the precomputed matrix (from SVD), and then find the closest mach to documents in reduced space, typically using the cosine measure.

3.1 Singular value decomposition (SVD)

Given a m *x n* matrix A with $m \geq n$, SVD(A) is $A = U \sum V^T$ where $U = U^T = VV^T = I$ and $\Sigma = diag(\sigma_1, \sigma_2, ..., \sigma_n)$. Example

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 \\ 0 & 1 \\ \frac{1}{\sqrt{2}} & 0 \end{bmatrix} \begin{bmatrix} \sqrt{2} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Note if we require $\alpha_1 \geq \alpha_2 \geq ... \geq \alpha_n$, then the SVD is unique for unequal σ.

3.1.1 Properties of SVD

The key property of SVD is given in the following Theorem.

Theorem: Let $A = U \sum V^T$ be the SVD of A. The matrix $A_k = \sum_{i=0}^{k} u_i . \sigma_i . v_i^T$ is the closest matrix of rank *k* (number of independent columns is *k*) to A.

Notes: A_k is a *m x n* matrix, u_k is denoted the first *k* columns of U and is *m x k.* We also define σ_k to be the upper *k x k* submatrix of Σ and v_k to be the first *k* row of V. By definition the rank (B)=*k,* if the maximum number of independent columns of B is *k* and the metric for measuring "closest" is $\sum_{i,j} [A_{i,j} - (A_k)_{i,j}]^2$.

Some other properties are that the columns of U, V are the orthonormal eigenvectors of AA^T, $A^T A$ respectively and are referred to as the *left singular vectors, right singular vectors*. The diagonal values of Σ are the nonnegative square roots of the eigenvalues of AA^T and are referred to as *singular values.*

In our example above $AA^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$ and $A^T A = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$.

Lastly calculation the SVD has similar techniques as for calculation eigenvectors which takes $O(n^3)$ for dense matrices and $O(n^2 m + nm^2)$ for sparse matrices using Lanczos Algorithm

3.1.2 Queries

Now that are documents are described in a *k*-dimensional subspace of the original *n* dimensional space via u_k, σ_k,and v_k we define the following $\hat{q} = q^T u_k \sigma_k^{-1}$ and $q = U \sum \hat{q}$. Now q is the query and $\hat{q}$ is the query transformed to the lower dimension space.

3.2 Application of LSI

Of the first is traditional indexing and searching. Other applications include *matching people*, where for example one can assign reviewers to papers by first training the application using papers written by potential reviews. Another is *modeling human memory,* one such app trained on an encyclopedia and was then tested on the synonym test of the TOEFL exam in which the app did as well as the average student.

The last app we will discuss is *Cross Language retrieval,* where one groups words that have similar meanings in different languages. So our app is "trained" on databases of documents which are composed of the same text in multiple languages appended together. This has been done for English-French, English-Japanese Kanji, and English-Greek.

3.3 Performance of LSI

A study was done using the TREC database around 1990 which has over 1,000,000 documents from AP News wire, Wall street Journal, journal Abstracts, and U.S Patents among others. Queries tend to be between 10 and 100 keywords with an average of 300 relevant documents per query. The relevance judgements were hand checked.

Now LSI was used by creating a index built with a sample of around 70,000 documents instead of the full database. The full database was not used because of the cost of SVD, $nk^2 \approx 10^6 10^6$. Now when LSI was compared to the best keyword searches, it fared 31% better on filtering tasks and 16% better on retrieval task. So it fared better in both precision and recall in which one usually finds a trade off between the two.

3.4 Example from [3]

keywords appear in more than one book title. Of course, alternative parsing strategies can increase or decrease the number of indexing keywords (or terms).

TABLE 2
Database of titles from books reviewed in SIAM Review. *Underlined keywords appear in more than one book title.*

Label	Titles
B1	A Course on Integral Equations
B2	Attractors for Semigroups and Evolution Equations
B3	Automatic Differentiation of Algorithms: Theory, Implementation, and Application
B4	Geometrical Aspects of Partial Differential Equations
B5	Ideals, Varieties, and Algorithms – An Introduction to Computational Algebraic Geometry and Commutative Algebra
B6	Introduction to Hamiltonian Dynamical Systems and the N-Body Problem
B7	Knapsack Problems: Algorithms and Computer Implementations
B8	Methods of Solving Singular Systems of Ordinary Differential Equations
B9	Nonlinear Systems
B10	Ordinary Differential Equations
B11	Oscillation Theory for Neutral Differential Equations with Delay
B12	Oscillation Theory of Delay Differential Equations
B13	Pseudodifferential Operators and Nonlinear Partial Differential Equations
B14	Sinc Methods for Quadrature and Differential Equations
B15	Stability of Stochastic Differential Equations with Respect to Semi-Martingales
B16	The Boundary Integral Approach to Static and Dynamic Contact Problems
B17	The Double Mellin-Barnes Type Integrals and Their Applications to Convolution Theory

Corresponding to the text in Table 2 is the 18×17 term-document matrix shown in Table 3. The elements of this matrix are the frequencies in which a term occurs in a document or book title (see Section 4). For example, in book title B3, the third column of the term-document matrix, *algorithms*, *theory*, *implementation*, and *application* all occur once. For simplicity, term weighting is not used in this example matrix. Now compute the truncated SVD (with $k = 2$) of the 18×17 matrix in Table 2 to obtain the rank-2 approximation A_2 as defined in Figure 1.

Using the first column of U_2 multiplied by the first singular value, σ_1, for the x-coordinates and the second column of U_2 multiplied by the second singular value, σ_2, for the y-coordinates, the terms can be represented on the Cartesian plane. Similarly, the first column of V_2 scaled by σ_1 are the x-coordinates and the second column of V_2 scaled by σ_2 are the y-coordinates for the documents (book titles). Figure 4 is a two-dimensional plot of the terms and documents for the 18×17 sample term-document matrix.

Notice the documents and terms pertaining to *differential equations* are clustered around the x-axis and the more general terms and documents related to *algorithms* and *applications* are clustered around the y-axis. Such groupings suggest that the subset of book titles {B2, B4, B8, B9, B10 B13, B14, B15} contains titles similar in meaning, for example.

TABLE 3
The 16 × 17 term-document matrix corresponding to the book titles in Table 2.

Terms	Documents																
	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15	B16	B17
algorithms	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0
application	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
delay	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
differential	0	0	0	1	0	0	0	1	0	1	1	1	1	1	1	0	0
equations	1	1	0	1	0	0	0	1	0	1	1	1	1	1	1	0	0
implementation	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
integral	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
introduction	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
methods	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0
nonlinear	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
ordinary	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
oscillation	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
partial	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
problem	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0
systems	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0
theory	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	1

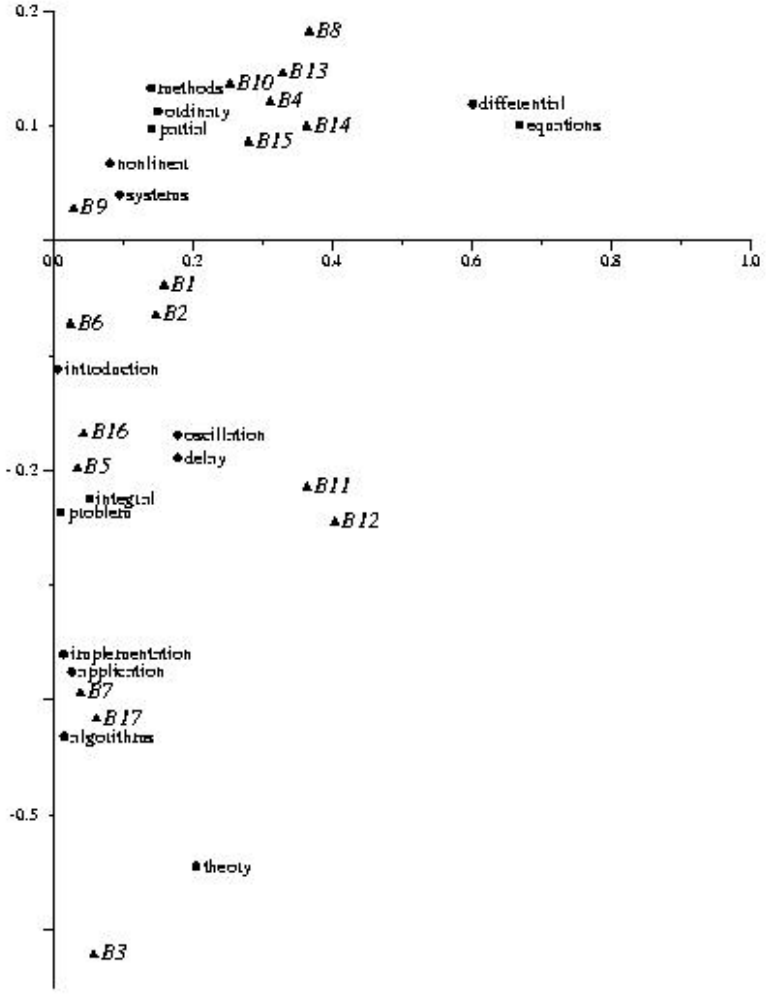

FIG. 1. *Two-dimensional plot of terms and documents for the* 16 × 17 *example.*

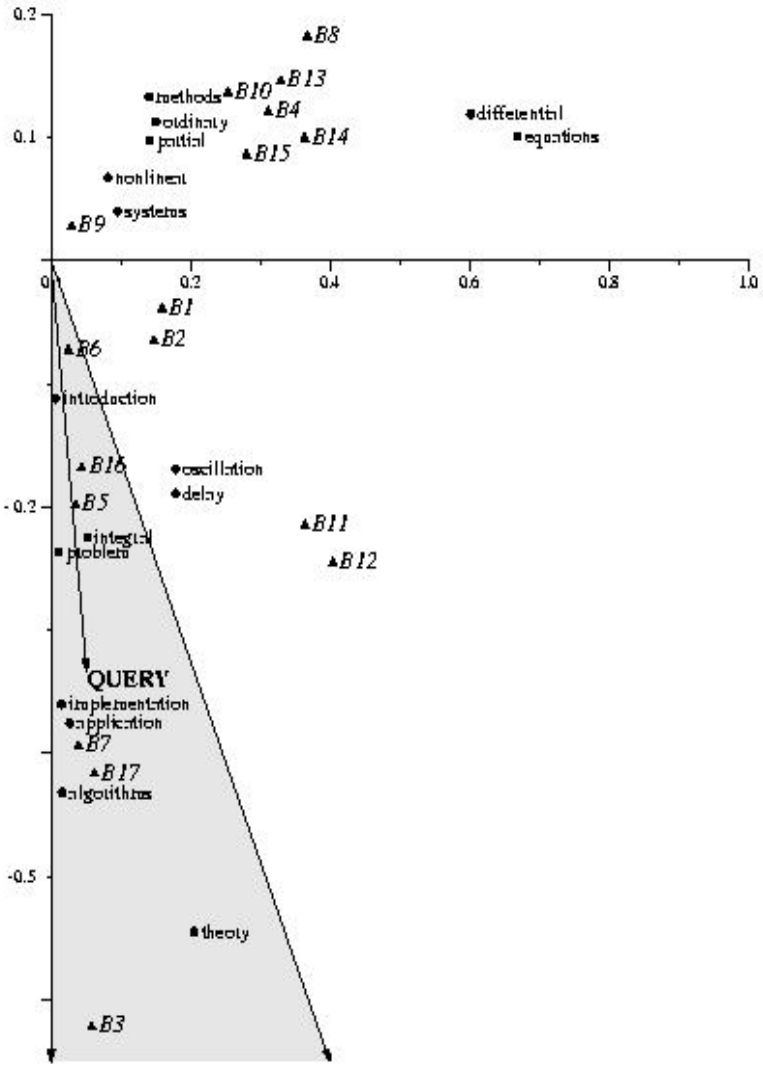

FIG. 6. *A Two-dimensional plot of terms and documents along with the query* application theory.

References:

[1] D. L. Lee, Y. M. K, G. P, "*Efficient signature file methods for text retrieval*", IEEE Transactions on Knowledge and Data Engineering, Volume: 7 Issue: 3,June 1995, p.p.423-435

[2] B. Horowitz, G. Blelloch, "*Scribing notes for algorithms in the real world – lecture 22*", 1998

[3] M Berry, S. Dumais, and G. O'Brien. *Using Linear Algebra for Intelligent Information Retrieval*, SIAM Review 37:4 (1995), pp. 573-595.