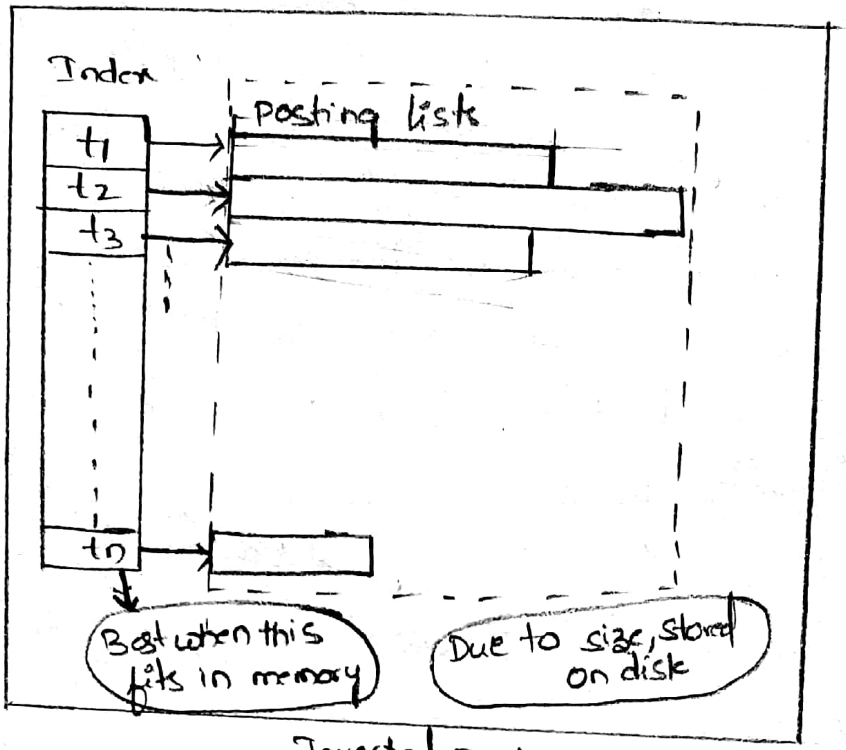
Inverted Index

Since many document collections are reasonably static, it is feasible to build an inverted index to quickly find terms in the document collection. Inverted indexes were used in both early information retrieval and database management systems in the 1960's [Bleir, 1967]. Instead of scanning the entire collection, the text is preprocessed and all unique terms are identified. This list of unique terms is ~~reff~~ referred to as index. For each term, a list of documents that contain the term is also stored. This list is ~~reff~~ referred to as a posting list.

An entry in the list of documents can also contain the location of the term in the document to facilitate proximity searching. Additionally, an entry can obtain a manually or automatically assigned weight for each term in the document. This weight frequently used to computations that generate a measure of relevance to the query. Once this measure is computed, the document retrieval algorithm identifies all the documents that are relevant to the query by sorting the coefficient and presenting a ranked list to the user.

Indexing requires additional overhead since the entire collection is scanned and substantial I/O is required, to generate an efficiently represented inverted index for use in secondary storage. Indexing was shown to ~~dragr~~ dramatically reduce the amount of

I/O required to static satisfy and adhoc query. upon Efficiency.



Inverted Index

receiving a query, the index is consulted, the corresponding posting lists are retrieved and the algorithm ranks the documents based on the contents of the posting lists.

The size of the index is another concern. many Indexes can be equal to the size of the original text. This means that storage requirments are double due to the index. However, compression of the index typically results in a space requirment of less then ten percent of the original text. The terms or phrases stored in the index depend on the parsing algorithms that are employed.

The size of posting list in the inverted index can be approximated by the zipfian distribution - zipf proposed that the term frequency distribution in a natural language is such that if all terms were ordered and assigned a rank, the product of their frequency and their rank

Scanned by CamScanner

would be constant. Table 5.1 illustrates the zipfian distri-bution when this constant is equal to one.

Using $\frac{c}{r}$ wher r is rank and c is the value of the constant. an ~~es~~ estimate can be made for the number of occurrences of a given term. The constant, c is domain-specific and equals to the number of occurances of the most frequent term.

Table 5.1 Top five Terms in Zipfian Distrubtion.

Rank	Frequency	constant
1	1.00	1
2	0.50	1
3	0.33	1
4	0.25	1
5	0.20	1

Building an Inverted Index

An inverted Index consist of two components, a list of each distinct term refferred to as the index and a set of lists refferred to as posting lists. To compute relevance ranking the term frequency or weight must be maintained. Thus, a posting list contains a set of tuples for each distinct term in the collection. The set of tuples is of the form <doc_id, tf> for each distinct term in the collection. A typical uncompressed index spends four bytes on the document identifier and two bytes on the term frequency since a long document can have a term that appears more than 255 times.

consider a document collection inwhich document, one contains two occurrences of sales and one occurrence of veohicle. Document two contains one occurance of veehide. The Index would contain the entries vehicle and sales. The posting list is simply a linked list that is associated with each of these terms. For this example we would have:

$$sales \rightarrow (1,2)$$
$$vehicle \rightarrow (1,1)\ (2,1)$$

The entries in the posting lists are stored in ascending order by document number. clearly the construction of this Inverted index is expensive, but once built, quaries can be efficiently implemented. The algoritms of Underlying the implementation of the query processing and the construction of the inverted index are now described.

A possible approach to index creation is as follows. An inverted index is constructed by stepping through the entire document collection, one term at a time. The output of the index construction algorithm is a set of files written to disk. The files are.

<u>Index file</u> contains the actual posting list for each distinct in the collection. A term 't' that occures in i different documents will have posting list of the form.

$$t \rightarrow (d_1, tf_{1j}), (d_2, tf_{2j}), \ldots, (d_i, tf_{ij})$$

Where d_i indicates the document identifier of document i and tf_{ij} indicates the number of times term j

occurs in document i.
Document file: contains information about each distinct document - document identifier, long document name, data published etc.
weight file: contains the weight for each document. this is the denominator for the cosine coefficient - defined as the cosine of the angle between the query and document vector (see. section 2.1).
the construction of inverted index is implemented by scanning the entire collection, one term at a time. when a term is encountered, a check is made to see if this term is a stop word (if stop word removal is used) or if it is a previously identified term. A hash function is used to quickly locate the term in an array. collosions caused by the hash function are resolved via a linear linked list. Different hashing functions and their relative performance are given in. once the posting list corresponding to this term is identified, the first entry of the list is checked to see if its document identifier matches the current document. If it does, the term frequency is merely incremented. Otherwise, this is the first occurrence of this term in the document, so a new posting list entirely is added to the start of the list.

The posting list is stored entirely in memory. Memory is allocated by dynamically for each new posting list entry. with each memory allocation, a check is made to determine if the memory reserved for indexing has been exceeded. If it has, processing halts while all posting lists resident in memory are written to disk. once processing continues, new posting lists are written. with each output to disk, posting list entries for the same term are chained together.

Processing is completed when all the terms are ~~Chained~~. processed. At this point, the inverse document frequency for each term is computed by scanning the entire list of unique terms. once the inverse document frequency is computed, it is possible to compute the document weight (the denominator for the cosine coefficient). this is done by scanning the entire posting list for each term.

Query processing

Recent work has focused on improving query run time. moffat and zobel have shown that query performance can be improved by modifying the inverted index to support fast scanning of a posting list. Other work has shown that reasonable precision and recall can be obtained by retriering fewer terms in the query. computation can be reduced even further by eliminating some of the complexity found in the vector space model.

Inverted Index Modifications

Moffat and zobel show how an inverted index can be segmented to allow for a quick search of a posting list to locate a particular document. the typical ranking algorithm scans the entire posting list foreach term in the query. An array of document scores is updated for each entry in the posting list. Moffat and zobel suggest that the least frequent terms should be processed first.

The premise is that less frequent terms carry the most meaning and probably have the most significant contribution to ahigh-ranking documents. The entire posting lists for these terms are processed. Some algorithms suggest that processing should stop after 'd' documents are assigned to non zero score. The premise is that at this point, the high-frequency terms in the query will simply be generating scores for documents that will not end up the final top t documents where t is the moven number of documents that are displayed to the user.

A suggested improvement is to continue pro-cessing all the terms in the query, but only updated the weights found in the d documents. In order words, after some threshold of d scores has been reached, the remaining query terms become part of an AND (they only increment documents who contain another term in the query) instead of the usual vector space OR. At this point, it is cheaper to reverse the order of the nested loop that is used to increment scores. prior to reaching d scores, the basic algorithm is

```
For each term t in the Query Q
        Obtain the posting list entries for t
      For each posting list entry that indicates t in
                                       doc i
        Update score for document i.
```

for query terms with small posting lists, the inner loop is small; however when terms that are very frequent are examined, extremely long posting lists are prevalent. Also after d documents are accessed, there is no need to update the score for every document, it is only necessary to update the score for those documents that have a non-zero score.

To avoid scanning very long posting lists, the algorithm is modified to be:

```
for each term t ~~and~~ in the query Q.
        Obtain posting list p, for document that contains t.
    for each document x in the reversed list of d documents.
        Scan posting list p for x
        if x exists
            update score for document x.
```

The key here is that the inverted index must be changed to allow quick access to a posting list entry. It is assumed that the entries in the posting list are stored by a document identifier. As a new document is encountered, its, entry can be appended to the existing posting list. moffat and Zobel propose to change the posting list by partitioning it and adding pointers to each partitions. the position list can quickly be scanned by checking the first partition ~~at~~ pointer (which contains the document identifier of the highest document in the partition and a pointer to the next partition). This check indicates whether or not a jump should be made to the next partition} or if the current partition should be scanned. The process continues until the partition is found, and the document we are looking for is matched against the elements of the partition. A small size, d, of about 1,000 resulted in the best cpu time for a set of TREC queries against the TREC data.

· partial Result set Retrieval

Another way to improve run-time performance to stop processing after some threshold of computational resources is expended. one approach counts disk I/O operations and stop after a threshold of disk I/O operations is reached. The key to this approach is to sort the terms in the query based on some indicator of term goodness and process the terms in this order. By doing this, query processing stops after the important terms have been processed. sorting the terms is analogous to sorting their posting lists. three measures used to characterize a posting list are now described.

cutoff Based on Document Frequency:

The simplest measures of term quality is to rely on document frequency. This was described in which showed that using between twenty-five to seventy five percent of the query terms after they were sorted by document query frequency resulted in almost no degardation in precision and recall for the TREC-4 document collection. In some cases, precision and recall improves with fewer terms because lower ranked terms are sometimes noise terms such as good nice useful etc. These terms have long posting lists that result in scoring thousands of documents and do little to improve the quality of the result. Using term frequency is a means of implementing adynamic Stop word list in which high-frequency terms are eliminated without using a static set of stop words.

Cutoff Based on maximum Estimated weight:

Two other measures of sorting the query terms are described in the first computes the maximum term frequency of a given query term as tfmax and uses the following as a means of sorting the query.

tfmax X Idf .

The idea is that a term appears frequently in all the documents in which it appears, is probably of more importance than a term that appears infrequently in the document that it appears in. The assumption is that the maximum value is a good indicator of how often the term appears in a document.

Cutoff Based on the weight of a Disk page in the posting list.

The cutoffs based on term weights can be used to characterize posting lists and choose which posting list to process first. the problem is that posting list can be quite long and might have substaintial skew. to avoid this problem a new measure sorts disk pages within a posting list instead of entire posting list. At index Creation time, the posting lists are sorted in the decreasing order by term frequency and instead of just a pointer that points to the first entry in the posting list, the index contains an entry for each page of the posting list. The entry indicates the maximum term frequency on a given page. The posting list pages are then sorted by

$$tf_{max} \times idf \times f(I)$$

Vector simplification:

Recent work has shown, in many cases, that specifications to the vector space model can be made only limited degradation in precision and recall. In this work, five variations to the basic cosine measure were tested on five small collections and 10,000 articles from the wall street Journal portion of the TREC collection. To review the baseline cosine coefficient

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

The first variation was to replace the document length normalization that is based on weight with the square root of the number of terms in D_i. The second variation was to simply remove the document length normalization.

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

The third measure drops the Idf, This eliminat-es one entry in the index for each term.

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

The fourth measure drops the t.f but retains the idf This eliminates the need to store tf in each entry of the posting list. This significantly reduces both the computational, storage and I/o costs.

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

The weight w_{qj} is one If term j is in the query and zero if otherwise. The weight w_{ij} is equal to idf_j if term j is in the document and zero otherwise.

The fifth and finial method simply counts matches between the query and the terms is

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

Scanned by CamScanner

Signature Files:

→ The use of signature files lies between a sequential scan of the original text and the construction of an inverted index.

→ A signature is an encoding of a document.

→ The idea is to encode all the documents as relatively small signatures

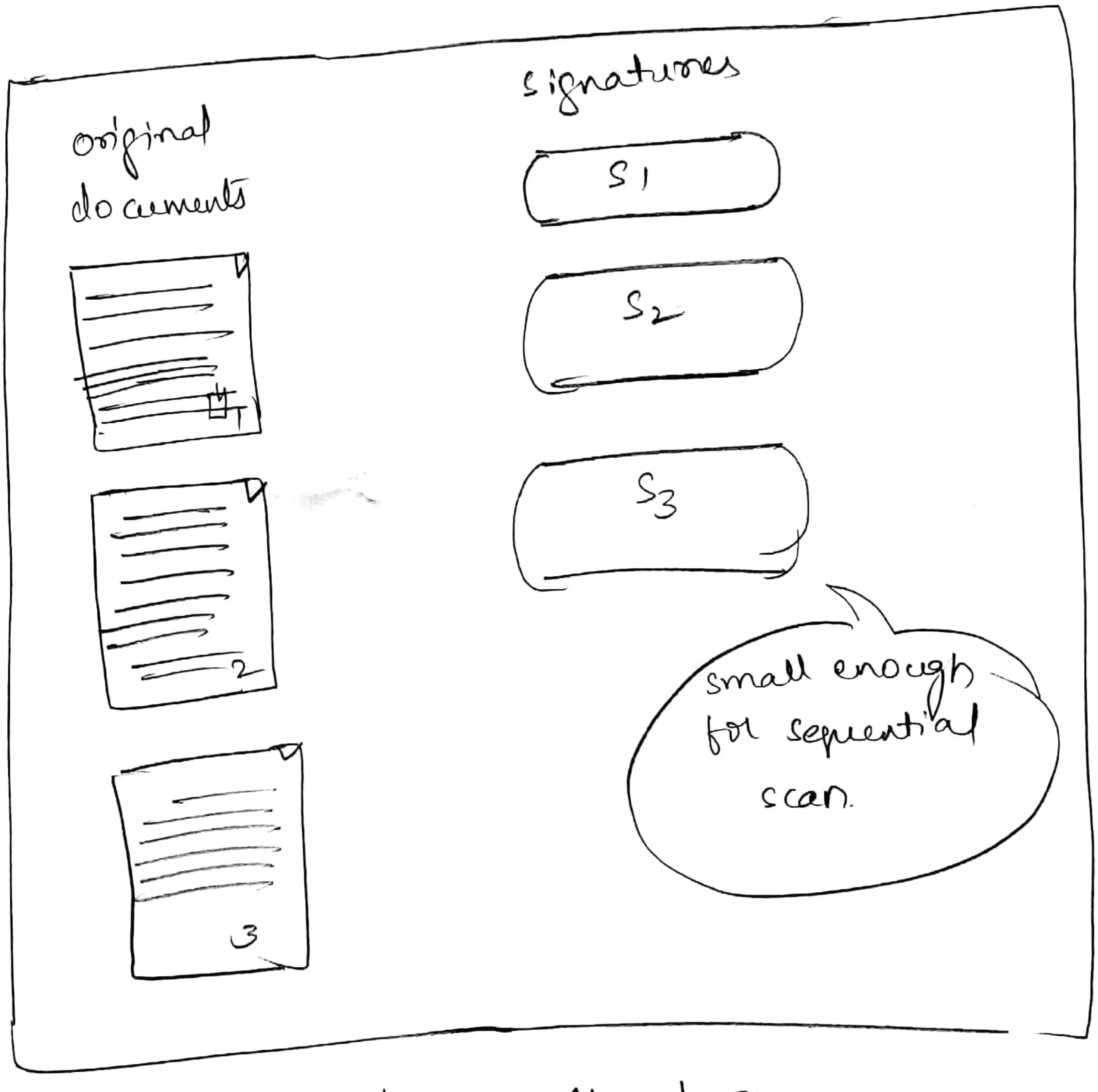
→ (often the goal is to represent a signature in only a few bits).

→ once this is done, the signatures can be scanned instead of the entire documents.

→ Typically, signatures donot uniquely represent a document (i.e, a signature represents multiple documents).

→ construction of a signature is often done with different hashing functions.

→ one (or) more hashing functions are applied to each word in the document.



Building a signature

term	b(term)
t_1	0101
t_2	1010
t_3	0011

ten, the hashing function is used to set a bit in the signature.

Ex: if terms information and retrieval were in document and h(Information) and h(retrieval) corresponded to bits one and four respectively, a four bit binary signature for this document would appear as 1001.

Document signature

Document	Signature
d_1	0101
d_2	0111
d_3	1111

To implement document retrieval, a signature is constructed for the query. A Boolean AND is executed between the query signature and each document signature.

→ Signatures are useful if they can fit into memory.

Scanning to Remove False positives

Once a signature has found a match, scanning algorithm are employed verify whether ~~or~~ not the match is a false positive due to collisions.

→ The FSA scans text from right to left, as done in Boyer-Moore.

→ Note this is done for a query that contains multiple terms [Kurataki and Takeda, 1993].

Scanned by CamScanner

2) Duplicate Document Detection

A method to improve both efficiency and effectiveness of an information retrieval system is to remove duplicates or near duplicates.

→ For web search, the duplicate document problem is particularly acute. A search for the terms "apache" might yield numerous copies of web pages about the webserver product and numerous duplicates about the Indian tribe.

Finding exact duplicates

Duplicate detection is often implemented by calculating a unique hash value for each document.

→ Each document is then examined for duplication by looking up the value (hash) in either in-memory hash (or) persistent lookup systems.

Finding similar Duplicates

while it is not possible to define precisely at which point a document is no longer a duplicate of another, researchers have examined several metrics for documenting the similarity of a document to another.

$$r(D_1, D_2) = \left[\frac{S(D_i) \cap S(D_j)}{S(D_i) \cup S(D_j)} \right]$$

Shingles

The first near-duplicate algorithm we discuss in use of shingles. A shingle is simply a set of contiguous terms in a document. Shingling techniques, such as COPS [Brin et al, 1995], KOALA [Heintze, 1996], and DSC [Broder, 1998], essentially all compare the number of matching shingles.

Duplicate document v/s Similarity

Another approach is to simply compute the similarity coefficient b/w 2 documents. If the document similarity exceeds a threshold, the documents can be deemed duplicates of each other.

→ A document to document similarity comparision approach is thus computationally prohibitive given the theoretical $O(d^2)$ runtime, where d is number of documents.

Treating the document as a query

Another approach treats each result document as a New query and looks for other documents that match this document. This approach is not computationally feasible for large collections or dynamic collections ∵ each document must be queried against the entire collection.

I- Match

Uses a hashing scheme that uses only some terms in a document. The decision of which terms to use is key to the success of algorithm. I- match is a hash of document that uses collection statistics Ex. idf, to identify which terms should be used as basis for comparision