Indexing Shared Content in Information Retrieval Systems

Andrei Z. Broder[1,⋆], Nadav Eiron[2,⋆], Marcus Fontoura[1,⋆], Michael Herscovici[3], Ronny Lempel[3], John McPherson[4], Runping Qi[1,⋆], and Eugene Shekita[5]

[1] Yahoo! Inc.
[2] Google Inc.
[3] IBM Haifa Research Lab
[4] IBM Silicon Valley Lab
[5] IBM Almaden Research Center

Abstract. Modern document collections often contain groups of documents with overlapping or shared content. However, most information retrieval systems process each document separately, causing shared content to be indexed multiple times. In this paper, we describe a new document representation model where related documents are organized as a tree, allowing shared content to be indexed just once. We show how this representation model can be encoded in an inverted index and we describe algorithms for evaluating free-text queries based on this encoding. We also show how our representation model applies to web, email, and newsgroup search. Finally, we present experimental results showing that our methods can provide a significant reduction in the size of an inverted index as well as in the time to build and query it.

1 Introduction

Modern document collections such as e-mail, newsgroups and Web pages, can contain groups of documents with largely overlapping content. On the Web, for example, studies have shown that up to 45% of the pages are *duplicates* – pages with (nearly) identical content that are replicated in many different sites [6, 8, 22]. In e-mail collections, individual documents with significant amounts of overlapping content are naturally created as people reply to (or forward) messages while keeping the original content intact. E-mail exchanges often contain long chains or *threads* of replies to replies, causing early messages in the thread to be replicated over and over. Similar threading patterns are also common in newsgroup discussions.

Information Retrieval (IR) systems typically use an inverted text index to evaluate free-text queries. During indexing, most IR systems process each document separately, causing overlapping content to be indexed multiple times. This, in turn, leads to larger indexes that take longer to build and longer to query. In this paper, we describe a scheme where overlapping content is indexed just once

⋆ Work done while this author was employed by IBM corporation.

and is logically *shared* among all documents that contain it. Thus, index space and index build times are greatly reduced. This does not come at the expense of any retrieval capabilities – queries can continue to be evaluated as if the full text of each document was indexed separately. Query evaluation is also faster due to the reduced index size.

Content is logically shared using a new *document representation model* where related documents are organized as nodes in a tree. Each node in a *document tree* can include content that it shares with all of its descendents as well as content and meta-data that is not shared with its descendents. The former is referred to as "shared content", while the latter is referred to as "private content". For example, in an e-mail or newsgroup thread, its document tree will mirror the history of the thread, with the root of the tree representing the first message of the thread whose text is quoted (and shared) with subsequent messages. We show how to encode our document representation model in a standard inverted index, and describe algorithms for evaluating free-text queries based on this encoding.

The basic operation of any inverted text index is the merging and intersection of *posting lists* - the lists of documents associated with each of the terms. The goal is to find the documents that contain terms appearing in a query. For efficiency, there are data structures and algorithms that allow skipping over the portions of the posting lists where no intersections might occur [5]. This operation is sometimes called a *zig-zag join* [11] and it is most useful for conjunctive queries. At its most basic, a zig-zag join of two lists proceeds by keeping two *index cursors* (also known as *posting list iterators*), one for each list. At every step, the cursor that points to a smaller document number is advanced at least as far as the other cursor. When the cursors meet, an intersection is reported, and one cursor is advanced to the next document in its list.

A key feature of our document representation model is that it can be easily encoded into an inverted index in such a way that the standard algorithms for evaluating free-text queries are still applicable. This is accomplished by defining virtual index cursors that are aware of our representation model and its encoding. In particular, the zig-zag join procedure uses virtual cursors as if they were normal physical cursors, and furthermore virtual cursors are relatively simple to implement on top of normal index cursors.

In the context of Web search, one may wonder why we opt to develop machinery for logically indexing all duplicates instead of simply retaining a single representative of any group of duplicates and discarding the rest. Some search engines (e.g., AltaVista as of 2000 [4]) adopted such a solution, which naturally also implies that they avoid returning duplicate content in their result sets. However, keeping a single representative is problematic for queries that include, in addition to some query terms, restrictions on meta-data such as URL, domain, or last-modified-date. Such meta-data is typically different for each duplicate. By sharing duplicate content and keeping the meta-data private, our representation model can support these queries, while at the same time indexing the duplicate content just once and preventing the return of multiple copies of it in the returned set of search results.

The main contributions of this paper include:

– A new document representation model where related documents are organized as a tree, allowing overlapping or shared content to be indexed just once.
– An encoding of our representation model that can easily support a standard zig-zag join for evaluating free-text queries on an inverted index.
– Descriptions of our representation model as applied to web, email, and newsgroup corpora, showing its usefulness in practical IR applications.
– Experimental results on large datasets showing that our representation model can provide a significant reduction in the size of an inverted index and in the time to build and query it.

2 Background

In this section we briefly review some basic IR concepts and terminology.

Inverted Index. Most IR systems use inverted indexes as their main data structure for full-text indexing [21]. There is a considerable body of literature on efficient ways to build inverted indexes (See e.g. [1, 3, 10, 13, 16, 21]) and evaluate full-text queries using them (See e.g. [5, 15, 21]).

In this paper, we assume an inverted index structure. The occurrence of a term t within a document d is called a *posting*. The set of postings associated to a term t is stored in a *posting list*. A posting has the form <*docid, payload*>, where *docid* is the document ID of d and where the payload is used to store arbitrary information about each occurrence of t within d. Here, we use part of the payload to indicate whether the occurrences of t are shared with other documents and also to store the offsets of each occurrence.

Each posting list is sorted in increasing order of *docid*. Often, a B-tree [11] is used to index the posting lists [10, 16]. This facilitates searching for a particular *docid* within a posting list, or for the smallest *docid* in the list greater than a given *docid*. Similarly, within a posting, term occurrences are sorted by offset thus making intra-document searches efficient.

Free-text Queries. Most IR systems support free-text queries, allowing Boolean expressions on keywords and phrase searches. Support for mandatory and forbidden terms is also common, e.g. the query `+apple orange -pear` indicates that `apple` is mandatory, `orange` is optional, and `pear` is forbidden. Most systems also support *fielded* search terms, i.e. terms that should appear within the context of a specific field of a document, e.g. `+title:banana -author:plum`. Note that the queried fields are most often meta-data fields of the documents.

Document at a Time Evaluation In this paper, we assume the document-at-a-time query evaluation model (DAAT) [20], commonly used in web search engines [3]. In DAAT, the documents that satisfy the query are usually obtained via a zig-zag join [11] of the posting lists of the query terms. To evaluate a free-text query using a zig-zag join, a *cursor* C_t is created for each term t in the query, and is used to access t's posting list. C_t.*docid* and C_t.*payload* access the *docid* and *payload* of the posting on which C_t is currently positioned. During a zig-zag join, the cursors are moved in a coordinated way to find the documents that satisfy the query. Two basic methods on a cursor C_t are required to do this efficiently:

- C_t.*next()* advances C_t to the next posting in its posting list.
- C_t.*fwdBeyond(docid d)* advances C_t to the first posting in its posting list whose *docid* is greater than or equal to d. Since posting lists are ordered by *docid*, this operation can be done efficiently.

Scoring. Once a zig-zag join has positioned the cursors on a document that satisfies the query, the document is scored. The final score for a document usually contains a query-dependent textual component, which is based on the document similarity to the query, and a query-independent static component, which is based on the *static rank* of the document. In most IR systems, the textual component of the score follows an additive scoring model like *tf* $\times$ *idf* for each term, whereas the static component can be based on the connectivity of web pages, as in PageRank [3], or on other factors such as source, length, creation date, etc.

3 The Document Representation Model

In our document representation model each group of related documents is organized as a *document tree*; the corpus being indexed is therefore a forest of document trees.

Each node in a document tree corresponds to a document that can include shared and private content. The private content of a document d is unique to d, whereas the shared content of d is inherited by its descendants. Sharing occurs top-down. Therefore, the document at a particular node effectively contains the shared and private content of that node plus the union of all its ancestors' shared content.

Our representation model is illustrated in Figure 1. This might correspond to an email exchange starting with d_1. It was quoted by two independent replies d_2 and d_4. It turn, d_2 was quoted in full (including d_1) by d_3, while d_4 was quoted in full by d_5 and d_6.

In Figure 1, d_i corresponds to the document whose *docid* $= i$, while S_i and P_i correspond to the shared and private content of document d_i, respectively. The content of d_1 is S_1 and P_1, while the content of d_3 is S_3 and P_3 plus S_1 and S_2.

We define a *thread* as the documents on a root-to-leaf path in a document tree. In Figure 1, there are three threads, d_1-d_2-d_3, d_1-d_4-d_5, and d_1-d_4-d_6.

Two functions on *docid* are needed for query evaluation:

- *root*(*docid d*) returns the root of d's document tree.

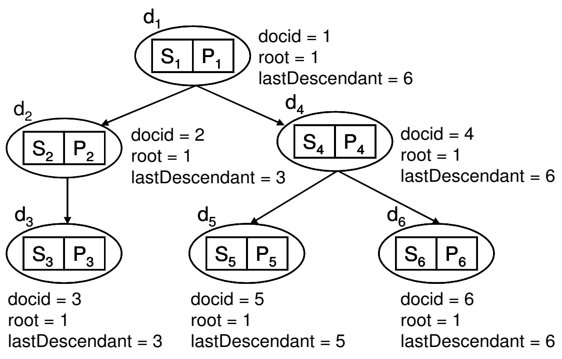


Fig. 1. Example of a document tree

– *lastDescendant*(*docid d*) returns the last descendant of the sub-tree rooted at d. If d is a leaf, then d itself is returned.

Figure 1 shows the *root*() and *lastDescendant*() values for the documents in our example. In principle, our representation model does not impose any restriction on the assignment of *docid*'s across document trees. However, the query evaluation algorithm described is Section 5 requires the *docid*'s within a document tree to be assigned sequentially using a depth first traversal. This guarantees that all documents in the range $\{d_{i+1}, \ldots, \mathit{lastDescendant}(d_i)\}$ are descendants of document d_i, and this is the assumption we are making from now on.

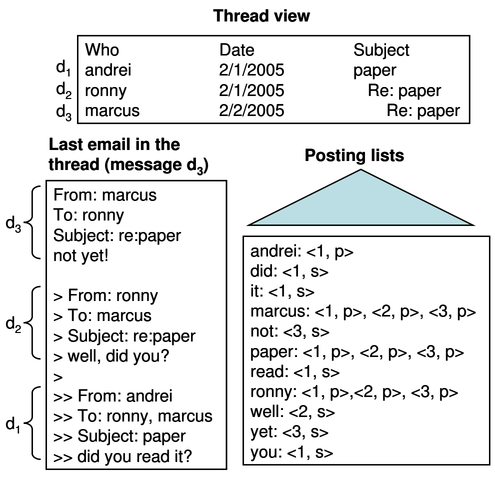
The main limitation of our document representation model is that related documents can only share content in a top-down, hierarchical manner. Nonetheless, we will show that there are many applications such as web, email, and newsgroup search that can still benefit from our representation.

4 Index Encoding

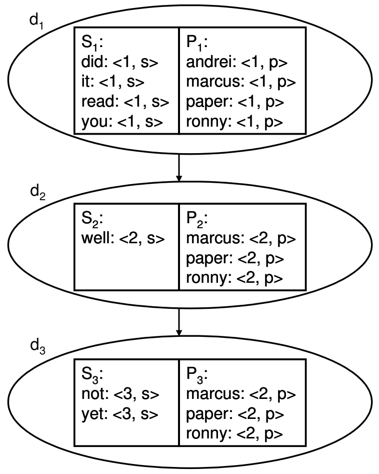
To support our document representation model, we use a standard inverted index with a few additions. First, each posting within the inverted index needs to indicate whether it is shared or private. This can be done by adding one bit to the payload. Second, the *root*() and *lastDescendant*() functions need to be implemented, which can be done using an in-memory table or an external data structure that allows efficient access.

Figure 2 illustrates how the posting lists might look for an email thread matching d_1-d_2-d_3 in the previous example. In the posting lists, the letter "s" indicates a shared posting, while the letter "p" indicates a private posting. Document d_1 corresponds to the original message, while d_2 is a reply to d_1, and d_3 is a reply to d_2.

In Figure 2, the content in the header fields is treated as private, while the content in the body is shared. For example, the posting for "andrei" is private to



(a) The inverted index


d1
S1: did: <1, s> it: <1, s> read: <1, s> you: <1, s>
P1: andrei: <1, p> marcus: <1, p> paper: <1, p> ronny: <1, p>
d2
S2: well: <2, s>
P2: marcus: <2, p> paper: <2, p> ronny: <2, p>
d3
S3: not: <3, s> yet: <3, s>
P3: marcus: <2, p> paper: <2, p> ronny: <2, p>

(b) The document tree

Fig. 2. An example of index encoding for a thread d_1-d_2-d_3

d_1, since it appears in the "From" field, whereas the posting for "did" is shared, since it appears in the body. In the latter case, d_1 shares the posting for "did" with its descendants, that is, d_2 and d_3. More generally, a document d_i *shares* a posting for a term t with document d_j if the posting is marked as shared and $d_j \in \{d_i, \ldots, \mathit{lastDescendant}(d_i)\}$ or if the posting is marked as private and $d_i = d_j$.

One subtle point to notice in Figure 2 is that, although the terms "did" and "you" appear in both d_1 and d_2, only one posting for them is indexed, namely, the one for d_1. This is because d_1 already shares those postings with d_2, which in turn means it would be redundant to index them twice. The virtual cursor algorithms that we will present later assume that these redundant postings do not appear in the index. However, for scoring, their occurrences and an indication of which document they belong to are reflected in the payloads. For example, both occurrences of "did" (for d_1 and d_2) would appear in d_1's payload, with an indication of the document they actually appear in. When scoring d_1 the system will ignore the occurence that is tagged with d_2, whereas when scoring d_2, both occurences will be taken into account.

5 Query Evaluation

We have described how our representation model can be encoded in an inverted index. Given this encoding, we now show how free-text queries can be evaluated using a standard zig-zag join. We allow queries to contain required, forbidden, and optional terms. The following high-level steps, which are described in the remainder of this section, are repeated during query evaluation:

– Enumerate candidates, that is, identify documents containing all required terms and none of the forbidden terms.
– Score the candidates, taking into account all occurrences of the query terms, including optional terms.
– Choose the next *docid* to resume searching for candidates.

5.1 Enumerating Candidates

A zig-zag join on required and forbidden terms is used to enumerate candidate documents. We define *virtual index cursors*, which look like normal cursors to a zig-zag join but are aware of our representation model and its encoding. A virtual cursor for a term t enumerates the same documents that a physical cursor for t would if shared content had been indexed multiple times. For example, in Figure 2(b), a virtual cursor for "well" would enumerate d_2 and d_3.

We create a "positive" virtual cursor for each required term, and a "negative" virtual cursor for each forbidden term. The latter allows a zig-zag join to treat forbidden terms the same as required terms.

Recall that two basic cursor methods are required for a zig-zag join, namely, *next()* and *fwdBeyond()*. The algorithms for the positive and negative versions of these methods are shown in Figure 3 and Figure 4, respectively. In the algorithms, $this.docid$ corresponds to the virtual cursor's current position, while C_p corresponds to the underlying physical cursor.

Positive next() and fwdBeyond()

Turning to Figure 3, the algorithm for positive *next()* is relatively straightforward, except when C_p is on a shared posting. In that case, all of C_p's descendants, which inherit the term from C_p, are enumerated (lines 2–4) before C_p is physically moved (line 7).

The algorithm for positive *fwdBeyond(d)* relies on the physical cursor method *fwdShare()*, which will be describe shortly, to do most of its work. The call to C_p*.fwdShare(d)* tries to position C_p on the next document that shares the term with d (line 6). If there is no such document, *fwdShare()* returns with C_p positioned on the first document beyond d.

Negative next() and fwdBeyond()

The algorithm for negative *next()* is shown in Figure 4. It works by trying to keep C_p positioned ahead of the virtual cursor. The documents $d \in \{this.docid, \ldots, C_p - 1\}$, which do *not* contain the term, are enumerated until the virtual cursor catches up to C_p (line 4). When that happens, the virtual cursor is forwarded past the documents that inherit the term from C_p (lines 5–9), after which C_p is moved forward (line 10). These steps are repeated until C_p moves ahead of the virtual cursor again.

The algorithm for negative *fwdBeyond(d)* calls *fwdShare(d)* to position C_p on the next document that shares the term with d (line 6). Then *next()* is called to position the virtual cursor on the next document that does not contain the term (line 14).

Physical fwdShare()

The algorithm for *fwdShare(d)* is shown in Figure 5. It keeps looping until the physical cursor moves beyond d or to a posting that shares the term with d (line 1). The movement of the physical cursor depends on whether the cursor lies outside d's document tree (lines 5–7), within the tree but outside d's thread (lines 9–11), or is on a private posting (lines 13–15).

```
PositiveVirtual::next()
  // Forward the virtual cursor to the next
  // document that contains the term.
  1. last = lastDescendant(Cp.docid);
  2. if (Cp.payload is shared and this.docid < last) {
  3.    // not done enumerating descendants of Cp
  4.    this.docid += 1;
  5. } else {
  6.    // advance Cp and reset docid
  7.    Cp.next();
  8.    this.docid = Cp.docid;
  9. }

PositiveVirtual::fwdBeyond(docid d)
  // Forward the virtual cursor to the next document
  // at or beyond document d that contains the term.
  1. if (this.docid >= d) {
  2.   // already beyond d, so nothing to do
  3.   return;
  4. }
  5. // try to forward Cp so it shares the term with d
  6. Cp.fwdShare(d);
  7. // set docid to Cp if it
  8. // is beyond d, else set it to d
  9. this.docid = max(Cp.docid, d);
```

Fig. 3. positive next() and fwdBeyond()

5.2 Correctness Proof for next() and fwdBeyond()

Because of space limitations, we can only provide a sketch of the correctness proof for virtual next() and fwdBeyond(). The proof follows from:

Theorem 1. *On a posting list generated from our representation model, the virtual next() and fwdBeyond() methods accurately simulate the behavior of next() and fwdBeyond() on a standard posting list.*

```
NegativeVirtual::next()
  // Forward the virtual cursor to the
  // next document not containing the term.
  this.docid += 1;
  // keep incrementing the cursor until it
  // is on a document not containing the term
  while (this.docid >= Cp.docid) {
     if (Cp.payload is shared) {
        this.docid = lastDescendant(Cp.docid) + 1;
     } else {
        this.docid = Cp.docid + 1;
     }
     Cp.next();
  }

NegativeVirtual::fwdBeyond(docid d)
  // Forward the virtual cursor to the next
  // document at or beyond the document d
  // that does not contain the term.
  if (this.docid >= d) {
     // already beyond d, so nothing to do
     return;
  }
  // try to forward Cp so it shares the term with d
  Cp.fwdShare(d);
  this.docid = d;
  if (Cp.docid > d) {
     // document d does not contain the term
    return;
  }
  // document d contains the term
  // call next() to move the cursor and Cp
  this.next();
```

Fig. 4. negative next() and fwdBeyond()

Proof sketch: The proof is based on proving the invariants we keep for the two types of virtual cursors, namely, for positive ones, that all *docid*'s between the current physical cursor position and its last descendant are valid return values, and that in the negated case, all *docid*'s between the current virtual position and the physical cursor position are valid return values. These invariants guarantee that our methods do not return spurious results. We further prove that we never skip valid results, completing the proof. □

5.3 Scoring Candidates

Our representation model does not impose any restriction on the scoring function. However, the scoring function could take thread information into account.

```
Physical::fwdShare(docid d)
  // Try to forward the physical cursor so it shares
  // the term with document d. If there is no such
  // document, return with the cursor positioned on
  // the first document beyond d.
  while (this.docid <= d and this.docid
           does not share the term with d) {
     root = root(d);
     last = lastDescendant(this.docid);
     if (this.docid < root) {
        // the cursor is not in the
        // same document tree as d
        this.fwdBeyond(root);
     } else if (last < d) {
        // in the same document tree
        // but not in the same thread
        this.fwdBeyond(last + 1);
     } else {
        // in the same thread, but private
        // posting on a different document
        this.next();
     }
  }
```

Fig. 5. fwdShare() on a physical cursor

For example, documents toward the bottom of a document tree could be assigned a lower score.

Candidate enumeration returns with the virtual cursors positioned on postings for a candidate document d. Scoring usually needs to take into account all the occurrences of each query term t in d, including optional terms. Given a virtual cursor C_t, this can be done by iterating all the occurrences of t in the posting's payload. Since C_t is virtual, the physical cursor C_p associated with C_t would be used to access the payload.

If the textual component of the scoring function is expensive to compute, we can remember the textual score of a document d to speedup the scoring of other candidate documents in d's thread. For example, suppose d_i has been scored and now another candidate d_j on the same thread needs to be scored. We can compute its textual score as:

$$Score_{text}(d_j) = PScore(d_j) + \sum_{d_k \in \text{path}(d_i, d_j)} SScore(d_k)$$

Here, *PScore*() and *SScore*() are the private and shared parts of the textual score, respectively, and $\text{path}(d_i, d_j)$ is the path from d_i to d_j in the document tree.

5.4 Choosing the Next Docid

After a candidate document d has been scored, the next document to resume searching for candidates has to be chosen. This step depends on the retrieval policy of the system, which may be tuned for performance or a particular application's requirements. Possible choices include:

– Resume from the document $d + 1$. This allows all the qualifying documents in a document tree to be returned.
– Resume from document $lastDescendant(root(d)) + 1$. This allows only the first qualifying document in a document tree to be returned.
– Resume from document $lastDescendant(d) + 1$. This is a hybrid of the first two approaches, allowing only the first qualifying document in a thread to be returned.

6 Applications

This section describes how our document representation model can be applied to web and email search, showing its usefulness in practical IR applications.

6.1 Web Search

Studies have shown that up to 45% of web pages are duplicates and near duplicates [6, 8]. On the web scale, repeated indexing of duplicate content is a huge waste of resources, and furthermore, users are seldom interested in seeing duplicates in the search results. Hence, web search engines need some way to identify and filter duplicates from results. To identify duplicates, a *signature* is typically computed for each document by hashing its content. These signatures are then used to identify groups of duplicate documents, that is, *duplicate groups.* To filter duplicates from search results, one of the following techniques is commonly used:

– Only the *master* of each duplicate group is indexed and returned in searches. The master of a duplicate group can be chosen arbitrarily or by using some heuristic, like picking the duplicate with the highest static rank. (This technique has been used in the *AltaVista* search engine.)
– All documents in a duplicate group are indexed. A post-processing filter is used to make sure that only one document per duplicate group appears in search results. Optionally the entire group is presented to the user.

By indexing only masters, the inverted index is kept small and query evaluation can ignore duplicates. However, queries that include restrictions on metadata become problematic. For example, suppose that for performance reasons, IBM has mirrored its main HR web page at `us.ibm.com/hr.html` and `canada.ibm.com/hr.html`, but the US version has been chosen by the US-centric search engine as the master. Then the query `hr domain:canada.ibm.com` that asks for web pages from IBM Canada that have the term "hr" in their content, will not return the main page. This type of domain restriction might be explicit in the query, or it might have been added to the query by the query interface without the user knowledge, based, say on user's location or IP address.

Conversely, if all the documents in a duplicate group are indexed, queries that include restrictions on meta-data do not pose a problem, but then the same content must be indexed multiple times and the query evaluation runtime has to filter duplicates from results. This filtering might be expensive, since it has to be done in a separate post-processing step.

Using our document representation model, it becomes possible to index duplicate content just once, while still allowing queries to be answered as if all the duplicates were indexed. This is done by creating a linear tree for each duplicate group, in which the master of the group serves as the root and is thus indexed with both its (shared) content and its private meta-data. The rest of the documents in the duplicate group follow in arbitrary order, where only the (private) meta-data of each duplicate (such as URL, creation date, geo-location, etc) is indexed. Note that this representation naturally applies to documents that have no duplicates, i.e. to duplicate groups of size 1.

Since duplicates are not returned in search results, an added benefit of using our representation is that the postings for all the remaining duplicates in a duplicate group can be skipped during query evaluation as soon as one of them has been returned. As our experimental results will show, this can dramatically improve query performance.

6.2 Email and Newsgroup Search

In most email and newsgroup clients replies include the full content of the original message. As illustrated earlier in Figure 2(a), our document representation model can support email or newsgroup search by simply creating a document tree for each message thread with a structure that mirrors the thread's history.

Unfortunately for our aims, most email and newsgroup clients allow users to edit the reply history, which potentially prevents any sharing between a message and its reply. Although such editing is common, our experimental results will show that 33% of the email messages in the Enron dataset [14] include an unedited reply history.

Another application for our document representation model is indexing on centralized email servers. For example, suppose an email message is sent to N users on the same server. Rather than index the message and its attachments N times, the message could be indexed as a two-level document tree, with the message body and attachments appearing as the shared root of the tree, and the meta-data of each recipient appearing as a separate, private leaf in the tree. Presumably, security information would be stored in the private meta-data and security meta-terms added to queries by the server to ensure that users could only search their own email.

Num Docs (K)	Pct. Dups	Meta Data (GB)	Content Data (GB)	Index w/o Dups (GB)	Index w/ Dups (GB)	Expected Comp.	Actual Comp.
500	36%	1.2	3.7	2.5	3.6	72%	69%
1000	37%	2.3	7.9	5.1	7.4	71%	69%
1500	41%	3.5	11.1	7.1	11.0	69%	64%
2000	43%	4.8	13.9	8.8	13.0	68%	68%
2500	44%	6.0	19.9	11.0	16.0	66%	69%

Table 1. Index sizes with and without duplicates

7 Experimental Results

In this section, we present experimental results for web and email search. We implemented our algorithms on the Trevi search engine [10], which is currently used to support web searches on IBM's global intranet. Experiments were run on a two-way SMP with dual 2.4 Ghz Intel Xeon processors running Linux. The disk storage was configured as two physical RAID arrays, each with 6 drives.

7.1 Results for Web Search

For this set of experiments, we built a series of inverted indexes from snapshots of IBM's intranet, varying the corpus size between 500K to 2.5M documents[6]. These sizes are indicative since the actual IBM intranet has about 15M documents (before duplicate elimination). We built each index with and without duplicates, using our document representation model in the latter case, as described in Section 6.

For each dataset, Table 1 shows the percentage of duplicates in the index, the overall amount of meta-data, the overall amount of content data, the size of the index without duplicates, the size of the index with duplicates, the expected compression ratio of the index without duplicates to the index with duplicates, and the actual measured compression ratio. The expected compression ratio can be computed as:

$$ExpectedComp = \frac{MetaData + ContentData \cdot (1 - PctDups)}{MetaData + ContentData}$$

Table 1 shows that our representation model reduced index sizes by roughly 30% on the IBM intranet data. Note that the actual compression ratio was better than expected in some cases. This was because content tokens, which include payload and offset information, tended to be bigger than than meta-data tokens, whereas our calculation of the expected compression assumes all postings are of equal size.

[6] When selecting a subset from the IBM intranet corpus we selected complete duplicate groups, so that the ratio of duplicate documents in all subsets has the same expectation, irrespective of the subset size.

Num Docs (K)	Index w/o Dups (sec)	Index w/ Dups (sec)	Pct. Decrease
500	540	780	31%
1000	1020	1440	29%
1500	1500	2340	36%
2000	1800	2940	39%
2500	2160	3540	39%

Table 2. Index build performance

The time to build an inverted index is an important metric in web search, since a decrease in the time to build an index allows it to be refreshed more frequently [10]. Table 2 shows the time to build each index on the IBM intranet data. Our indexing algorithm [10] is designed for batch builds of complete indexes, and hence a complete build was done for each experiment, rather than incrementaly growing an existing index. The build times reported do not include the time spent on separate analysis phase was run to identify duplicates. However, note that this phase would be necessary to filter duplicate results, regardless of whether our representation model was being used.

Table 2 reflects the time to scan the dataset (with duplicates) and then build the index (with or without duplicates). On average, it took about 30% to 40% less time to build an index without duplicates because of its smaller size. This is in keeping with the results in [10], which showed that the time to build an index is mostly I/O bound and a linear function of its size.

To study query performance, we ran one- and two-term queries on the dataset with 2.5M documents. We felt that this was a realistic set of experiments, since the average query on the IBM intranet contains only 1.2 terms. The single-term queries were on syntheticaly generated terms. 5 synthetic terms were added to the documents with selectivities that ranged from 20% to 100% (i.e., for each document, we added term $t_{20\%}$ with probability .2, a term $t_{40\%}$ with probability .4, etc.). We then used these five terms in five single-term queries. Note, however, that a query on such terms actually returned fewer documents than the expected fraction of the corpus size, because duplicates were filtered from the final result.

Query performance is strongly correlated to the number of physical cursor moves on the index – fewer moves translates into better query performance. Figure 6 shows the number of physical cursor moves and the execution time for single-term queries on a synthetic term. The results show that the number of cursor moves decreased by roughly 30% to 45% when our representation model was used. This is because the remaining postings in a duplicate group can be skipped as soon as one of them is returned. The improvement in execution time was even more pronounced, with a decrease of up to 80%. This was because of the combined effect of fewer cursor moves along with having a smaller index, which resulted in less CPU and I/O.

Figure 7 is provided to help understand the results in Figure 6. It illustrates how our representation model can decrease the number of physical cursor moves.

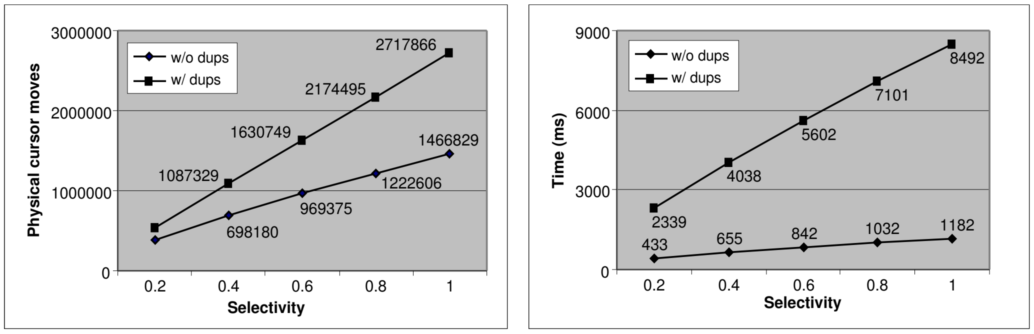


Fig. 6. Results for single-term queries

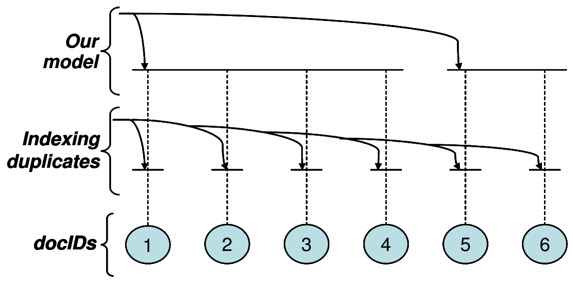


Fig. 7. Duplicate filtering at the index level

As shown, there are two document groups d_1-d_2-d_3-d_4 and d_5-d_6. Using our representation model, duplicates d_2, d_3, and d_4 can be skipped as soon as d_1 is enumerated. In addition to decreasing I/O, this means that those documents do not need to be filtered from the final results by the upper layers of the system, which in turn decreases CPU requirements. Our representation model effectively allows the filtering of duplicates to be pushed down to the physical index level, rather than doing it in the upper layers of the system.

Figure 8 shows the number of physical cursor moves and the execution time for two-term queries. For variety, real query terms on both content and meta-data were used in this case. The results for two-term queries were similar to those for one-term queries, with improvements of up to 80% in execution time when our representation model was used. Again, this was because of the combined effect of fewer cursor moves on a smaller index.

7.2 Results for Email Search

We used the Enron email dataset [14] to study how well our document representation model can be applied to email search. This dataset contains 517,431 emails that were made public in the Enron fraud investigation. Unfortunately,

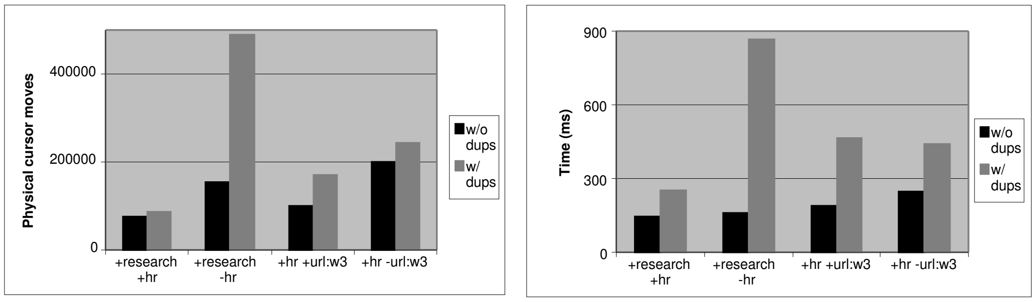


Fig. 8. Results for two-term queries

	Size (MB)
Meta Data	430
Content Data	910
Original Data	580

Table 3. Analysis of the Enron dataset

we did not have an email search engine available to us that could be used for experimentation. Consequently, we only used the Enron dataset to judge the potential effectiveness of our method on a real email collection. As discussed in Section 6, the effectiveness of our method in email search depends on the percentage of email threads that include an unedited reply history.

An analysis of the Enron dataset showed that 61% of its email messages belong to some thread, and that 33% of the messages in a thread included an unedited reply history. In Table 3 we show the overall amount of meta-data, the overall amount of content data, and the amount of content data that was left when the reply history was removed (the "Original Data"). We treated anything that appeared in email headers as meta-data.

Comparing the size of an inverted index for the Enron dataset with our representation model to one without it, the expected compression ratio can be computed as:

$$ExpectedComp = \frac{MetaData + OriginalData}{MetaData + ContentData}$$

This works out to 75%. In other words, using our representation method, an inverted index for the Enron data set would be roughly 25% smaller, which is comparable to what was observed on the IBM intranet data. We believe that the improvement in query performance would be comparable as well, due to the combined effect of fewer cursor moves on a smaller index.

It worth noting that, for email search, Gmail [12] returns only one email message per thread. Using our document representation model, this would allow the remaining messages in a thread to be skipped during query evaluation as soon as one of them was returned. As our results on the IBM intranet data showed, this kind of skipping can dramatically improve query performance.

8 Related Work

Much of the motivation for developing our document representation model is to save index space. In this respect, our representation model can be viewed as a special type of an index compression that relies on explicit knowledge of the data. Different compression methods as well as the trade offs involved in using them, have been widely published [7, 9, 17, 18, 23]. However, we regard our scheme as independent of those methods, and indeed it can be used in combination with them.

This paper focused on the problem of efficiently building and querying an inverted index once the shared content has already been identified. How to identify the shared content is beyond the scope of this paper but is related to the problem of identifying duplicates in a text corpus [2, 4, 6]. In the case of email or newsgroup threads, identifying shared content may be aided by cross-reference information in message headers.

Our representation model is particularly effective in email or newsgroup search. Recently, email search like that provided by Gmail [12] and Bloomba [19] has become quite popular. Unfortunately, the Gmail architecture is not public, and although Bloomba is tailored for indexing and searching email, it does not seem to do anything special for the shared content in email threads.

9 Conclusions

In this paper, we described a new document representation model where related documents are organized as a tree, allowing shared content to be indexed just once. A key feature of our representation is that it can easily support the standard zig-zag join algorithm for processing free-text queries on an inverted index. We described how this can be accomplished by defining virtual index cursors that are aware of our representation model and its encoding.

Our model can be applied to practical IR applications such as web, email, and newsgroup search. Using data from the IBM intranet, we provided experimental results showing that our method was able to reduce the size of the inverted index by roughly 30% and improve the performance of one- and two-term queries by up to 80%. The improvement in query performance was due to the synergistic effect of fewer cursor moves on a smaller index.

For future research, we hope to extend this work to improving XML retrieval. In particular, we would like to index XML documents efficiently while enabling search at the individual *tag* level, i.e. to retrieve the tag within each document that is most relevant to the query. To that effect,we can represent each document using its DOM tree, and share the content of each node (tag) with the content of its ancestors (enveloping tags). Note that this down-up model of content sharing is exactly the opposite of the top-down sharing model discussed in this paper.

References

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[2] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *KDD '03*, pages 39–48, 2003.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *WWW '98*, pages 107–117, 1998.

[4] A. Z. Broder. Identifying and filtering near-duplicate documents. In *CPM '00*, pages 1–10, 2000.

[5] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM '03*, pages 426–434, 2003.

[6] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. In *WWW '97*, pages 1157–1166, 1997.

[7] D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovici, Y. S. Maarek, and A. Soffer. Static index pruning for information retrieval systems. In *SIGIR '01*, pages 43–50, 2001.

[8] J. Cho, N. Shivakumar, and H. Garcia-Molina. Finding replicated web collections. In *SIGMOD '00*, pages 355–366, 2000.

[9] E. S. de Moura, C. F. dos Santos, D. R. Fernandes, A. S. Silva, P. Calado, and M. A. Nascimento. Improving web search efficiency via a locality based static pruning method. In *WWW '05*, pages 235–244, 2005.

[10] M. Fontoura, E. J. Shekita, J. Y. Zien, S. Rajagopalan, and A. Neumann. High performance index build algorithms for intranet search engines. In *VLDB '04*, pages 1158–1169, 2004.

[11] H. Garcia-Molina, J. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 2000.

[12] Gmail. `http://gmail.google.com/gmail/help/about.html`.

[13] S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *JASIST*, 54(8), 2003.

[14] B. Klimt and Y. Yang. The Enron corpus: A new dataset for email classification research. In *European Conference on Machine Learning*, 2004.

[15] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *VLDB '03*, pages 129–140, 2003.

[16] S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the web. In *WWW '01*, pages 396–406, 2001.

[17] A. Moffat and J. Zobel. Compression and fast indexing for multi-gigabyte text databases. *Australian Computer Journal*, 26(1), 1994.

[18] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *SIGIR '02*, pages 222–229, 2002.

[19] R. Stata, P. Hunt, and M. G. Thiruvalluvan. The Bloomba personal content database. In *VLDB '04*, pages 1214–1223, 2004.

[20] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Inf. Proc. Management*, 31(6):831–850, 1995.

[21] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann, 1999.

[22] Z. Zhang. The behavior of duplicate pages on the world wide web. Submitted to CIKM, 2005.

[23] J. Zobel and A. Moffat. Adding compression to a full-text retrieval system. *Software - Practice & Experience*, 25(8), 1995.