Query-based Partitioning of Documents and Indexes for Information Lifecycle Management

Soumyadeb Mitra[*] *and* Marianne Winslett[†]
Department of Computer Science
University of Illinois at Urbana Champaign
{mitra1,winslett}@cs.uiuc.edu

Windsor W. Hsu[‡]
Data Domain Inc.
Santa Clara, CA, USA
windsor.hsu@datadomain.com

ABSTRACT

Regulations require businesses to archive many electronic documents for extended periods of time. Given the sheer volume of documents and the response time requirements, documents that are unlikely to ever be accessed should be stored on an inexpensive device (such as tape), while documents that are likely to be accessed should be placed on a more expensive, higher-performance device. Unfortunately, traditional data partitioning techniques either require substantial manual involvement, or are not suitable for read-rarely workloads. In this paper, we present a novel technique to address this problem. We estimate the future access likelihood for a document based on past workloads of keyword queries and the click-through behavior for top-K query answers, then use this information to drive partitioning decisions. Our overall best scheme, the *document-split inverted index*, does not require any parameter tuning and yet performs close to the optimal partitioning strategy. Experiments show that document-split partitioning improves performance on a large intranet query workload by a factor of 4 when we add a fast storage server that holds 20% of the data.

Categories and Subject Descriptors

H.3.2 [**Information Systems**]: Information Storage and Retrieval-Information Storage

General Terms

Algorithms, Design, Management.

Keywords

Hierarchical Storage Management, Lifecycle Management, Tiered Storage, Inverted Index Partitioning, Keyword Query.

[*]This author was a co-op student at IBM Almaden Research Center when this research was started. Subsequently, he was supported by an IBM PhD Fellowship.

[†]This author was supported in part by NSF grants CNS 0716532 and CNS 0325951.

[‡]This author was with IBM Almaden Research Center when the research was started.

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

1. INTRODUCTION

The Sarbanes-Oxley Act and many other recent regulations require large businesses to archive huge amounts of digital information for years, including all business email and financial documents. These regulatory requirements have spawned an entire subindustry devoted to *archival data products*, which can compress and store documents for decades, protect them from tampering, and retrieve them as needed for internal decision-making, audits, or litigation. For quick lookup, each archival email, spreadsheet, and report is stored as a separate document that can be searched and retrieved independently using a full-text inverted index [3]. The inverted index supports *keyword queries*, where the user provides a list of words and receives a ranked list of the K documents judged to be most relevant by the search engine. For example, the archival product from HP (h18006.www1.hp.com/products/storageworks/riss/) provides this gmail-style lookup functionality.

Given the volume of documents and their long retention times, a cost-effective storage strategy is vital. In 2007, high-performance enterprise class storage devices cost as much as $10/GB, but lower-performing low-end disk systems or tape systems cost only $1-2/GB. Archiving all documents on high-end storage is too expensive, but keeping them all on low-end storage makes queries too slow; archival storage must balance these two needs. Thus the question is how best to partition the data and index across a multi-tier storage hierarchy.
(See www.powerfile.com/downloads/DemoFlash.htm for a description of this problem from an industrial perspective.)

The Storage Networking Industry Association uses the term *Information Lifecycle Management* (ILM) to refer to this and related challenges. Current ILM tools provide system administrators with interfaces to classify data, typically in the form of rules that drive data partitioning decisions. Unfortunately, current ILM tools require substantial manual intervention and hence are very costly to deploy and use. Our goal in this paper is to alleviate the system administrator's burden by providing high-quality automated assistance in the form of a decision procedure for *default partitioning* of data and index entries.

Not all data are equally likely to be accessed after being archived: a customer service email is less likely to be read than a critical business policy email. A simple and appealing approach is to keep the index on high-end storage, together with the most popular documents. However, a full-text inverted index can be 25-30% of the size of the data it indexes [14], and 95% of the words it contains will rarely or never be queried [8]. Thus keeping all the index on high-end storage is not cost-effective: we can obtain just as good average query performance at much lower cost, by moving most of the index to low-end storage.

$$\mathcal{W}(d,q) = \sum_{t \in q} TF(d,t) * IDF(t)$$

$$TF(d,t) = \frac{\log(1+\log(n(d,t))}{(1-s)+s*\frac{|d|}{avgDocSize}}$$

$$IDF(t) = \log(\frac{N}{N(t)+1})$$

Figure 1: Basic information retrieval functions. $n(d,t)$: Number of occurrences of t in d. $|d|$: d's document length. $avgDocSize$: Average document length. N: Total number of documents. $N(t)$: Number of documents containing term t. s: Tunable parameter (set to 0.4 for our experiments).

In this paper, we propose and evaluate **query-based partitioning**, a novel approach to partitioning documents and indexes across the storage hierarchy, based on the insight that documents not present in the top-K results of a query are unlikely to be accessed through that query. Query-based partitioning leverages this user behavior by placing a document according to the probability of it appearing in the top-K results of any likely keyword query. We frame the partitioning problem as an optimization problem that we solve analytically. We use a real-world workload to show that without any prior knowledge of which queries are most likely, query-based partitioning provides a factor of 4 (resp. 10) query speedup when used to place 20% (resp. 40%) of the data on top-tier storage, compared to an approach that employs only bottom-level storage.

This paper is organized as follows. Section 2 describes archival storage and keyword search, and Section 3 discusses related work. In Sections 4 and 5, we give an overview of query-based partitioning and analyze it under different query models. We discuss experimental results in Section 6. Sections 7 and 8 consider prefetching and throughput, and Section 9 concludes the paper.

2. BACKGROUND

Archival Data Properties. Most archived documents are rarely read. For example, archived email messages are only queried by their owners, auditors, and in response to litigation. However, to be competitive in the marketplace, archive search engines need good query response times. Search engines receive queries from many different users, so good query throughput is also important. Thus an ILM document and index partitioning strategy should maximize the number of requests that are served from the higher-performance top-level storage.

As illustrated in Figure 1(a), current ILM approaches make use of *epochs*, i.e., fixed-length intervals of time. Newly created documents belong to the current epoch, and are indexed using an index specific to that epoch. After the epoch ends, all the documents in that epoch are handed to the archival system for classification and partitioning, re-indexing, and migration to long-term storage. The original epoch index and data are then deleted or shredded, which is easy because they are isolated in epoch-specific files. This approach is popular because queries that specify the document creation time can often be executed by scanning the small epoch-specific indexes, rather than a single big index over all the documents. it true? In this paper, we are concerned with documents that are either already on long-term storage, or are in the process of being moved there.

Inverted Indexes and Keyword Search. As shown in Figure 4, a full-text inverted index contains one *posting list* for each word (also called a *term* or *keyword*) that occurs in the documents. Each posting list contains the IDs of all the documents that contain that word, plus some additional metadata. Keyword queries are answered by scanning the posting lists of the query words, producing a list of the documents containing some or all of the words. The resulting documents are assigned weights describing the extent of the match between the document and the query. For example, in the pivoted normalization method of the vector space model [12], the weight $\mathcal{W}(d,q)$ of a document d for a query q is based on the number of times each queried word occurs in the document d (its *term frequency* (TF)) and the inverse of the number of documents that contain each queried word (its *inverse document frequency* (IDF)), as shown in Figure 1

The documents in the search result are displayed in decreasing order of weight. The *rank* of a search result document is its position in this list, i.e., first, second, third, etc. Along with the document IDs or URLs, search engines typically display a document abstract, which may list the owner, creation time, document header, and so on. This abstract is stored in the index itself.

Storage Model. We consider an archival storage system consisting of top-level storage, such as a state-of-the-art SCSI-based device, and bottom-level storage such as IDE drives or tape-based storage as shown in Figure 1(b). We take the storage capacity C of the top-level device as an input to our algorithms, and assume that the bottom-level storage can store all the documents and index fragments that do not fit on top-level storage.

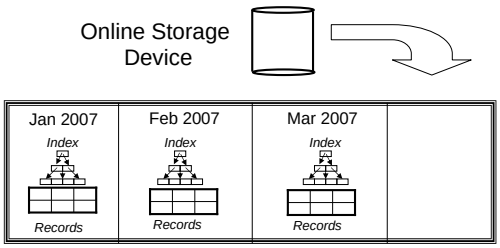
We characterize storage server performance by its seek time and rotational latency, i.e., the average time required to position the read head at a random location (including both seek and latency). We also consider the transfer time to actually read and transfer the document and/or posting list. Except in Section 7, which discusses prefetching, we fetch a document *after* the user clicks on it.

In the rest of the paper, we use the terminology *top-level and bottom-level documents* to refer to documents stored on top-level and bottom-level storage, respectively. Similarly, the index fragments stored on top-level and bottom-level storage are called the *top-level index and bottom-level index*, respectively. We also use the terminology *unsplit* and *split* storage to refer to unpartitioned and partitioned storage, respectively.

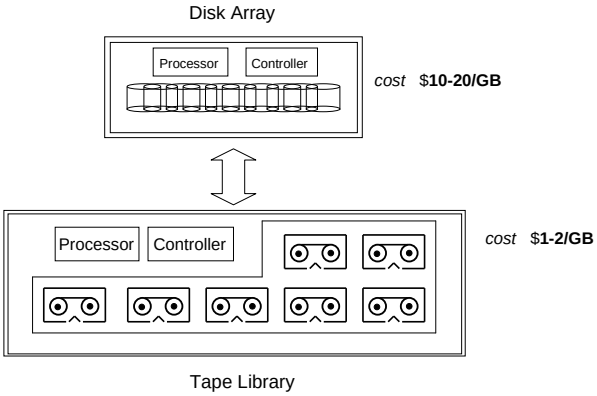
3. RELATED WORK

Most archival data partitioning systems are rule-based [10]. System administrators write a sequence of data placement rules based on document metadata (e.g., the CEO's documents go to the top level), document type (e.g., Excel documents go to the bottom level), source application, and/or quantitative or qualitative measures of the document's business criticality [4]. Developing these rules is a highly subjective, time-consuming, and human-intensive process. Query-based partitioning can alleviate this burden by serving as the default placement rule for documents not covered by manually generated rules. At an extreme, query-based partitioning can serve as the only placement rule, as in our experiments.

Researchers have also proposed making storage placement decisions based on the business importance of data: critical data must be stored on top-level storage. The business value can be estimated using historical information such as the cost of capturing, producing, or acquiring the information [10]; or through utility models that analyze how the information usage leads to actual business transactions and hence revenue [4]. Unfortunately, developing these exact cost models is also a highly subjective, time-consuming,



(a) Documents organized in epochs



(b) Proposed archival storage hierarchy

Figure 2: (a) Documents are stored on the online storage device and migrated to the archival storage device at the end of the epoch. (b) The archival storage server consists of fast top-level storage and slow bottom-level storage.

and human-intensive process. Furthermore, these techniques are only applicable to documents generated through specialized business processes and workflows and are difficult to apply to arbitrary documents like email and instant messages. We take a different approach to this problem. Instead of trying to estimate the business value of a document, we estimate the likelihood of it being accessed through the query interface. This likelihood only depends on the keywords contained in the document and the other documents in the corpus. The same model and partitioning technique can be used, independent of the business domain or application.

Another partitioning approach in current archival products is to classify data based on its recency of access [6]. Data caching exploits this by migrating recently accessed data to top-level storage, with the expectation that such data will be accessed again. Data caching is not very effective for archival workloads since queries are often run over data sets from different time epochs: consider year-end audit queries that look up financial documents from that year. Cached results from queries run over old records do not benefit current audit queries.

The cost of storing the index on top-level storage can be reduced by compressing it. Soffer et al. proposed a compression technique based on discarding the tails of posting lists [13]. Buttcher proposed a document-centric approach to index partitioning, in which the index entries for certain key terms of a document are stored in a compressed version of the index on top-level storage, while the rest of the document's terms are indexed on bottom-level storage. Ntoulas and Cho [9] also addressed the problem of partitioning the index across a storage hierarchy, to maximize the number of queries answered from the top-level (pruned) index.

We address a more general problem: partitioning both documents and index across hierarchical storage to minimize total execution time, including query execution and document access. To this end, we must determine the fraction of top-level storage allocated to the index and to the documents. The previous index partitioning approaches do not provide any analytic relationship between the space occupied by the index and its query performance. Without such a measure, it is hard to analytically determine the fraction of top-level storage that should be allocated to the index. Our results show that heuristically setting this fraction and then partitioning the documents and the index independently (in a direct extension of the previously proposed approaches) is suboptimal. One of our main contributions is to show how to analytically determine the optimal fraction of top-level storage to allocate to the index.

This aspect of our work also sets us apart from Baeza-Yates et al. [1], who devised algorithms to choose which index entries and query answers to store on top-level storage. Baeza-Yates et al. relied on simulations with a specific workload to find the best value for the fraction parameter in a particular setting. When conditions change, their approach will require simulation runs tuned to the new environment. One goal of Baeza-Yates et al. is to store the answers to certain queries on top-level storage. We do not store any query answers on top-level storage, because we expect low locality in an archival query workload; instead we put selected documents on the top-level storage. This difference in goals results in different problem formulations and hence different partitioning algorithms.

Another of our contributions is the *simplified query model*, which can be used to partition the index and data when no information about the expected workload is available, with excellent performance. The approach of Ntoulas and Cho and that of Baeza-Yates et al. use a query log for training, so their approaches cannot be used in this setting.

4. PARTITIONING APPROACHES

4.1 Additional Straw-man Approaches

In Section 1, we argued that it is not optimal to keep all of the index plus the most popular documents on the top level. Even worse is to store the entire index on the top-level and all documents on the bottom level; then all document accesses are expensive, and many posting lists in the index will never be read. This scheme also breaks down if the top-level storage is smaller than the index.

If we instead put the index on bottom-level storage, then every query involves accesses to the bottom level, which are slow. A better strategy is to create the top-level index over the documents on top-level storage. The disadvantage is that accessing a document on the bottom-level requires rerunning the query on the bottom-level index. Rerunning the query incurs as many disk seeks as the number of query keywords, and hence is likely to be more expensive than accessing a document, which typically incurs one seek. We should hence give priority to storing a document's index entries on the top level, rather than the document itself.

From: ceo@abc.com *To:* secretary@abc.com Let's meet.	*From:* ceo@abc.com *To:* broker@stock.com Sell all ABC shares.

Figure 3: Example email messages.

4.2 Query-based Partitioning

In a typical document lookup operation, the user formulates a query that she thinks will satisfy her information need. For example, to find all email between Martha and Ralph that discusses ImClone stock, she might use the query *Martha ImClone Ralph*. The search engine returns a ranked list of documents and document summaries based on information stored in the index. The user clicks on the returned documents that she thinks are relevant to her information need. In practice, users rarely click beyond the top K results, for some K (analyzed later). Thus a document not in the top K answers to any query is unlikely to be accessed by any user. This is the key idea behind our partitioning approach.

As a concrete example, consider the two emails of Figure 3. The email on the left has few identifying terms (terms with high IDF). Consequently, the right-hand email will appear in the top K results for more queries than the left-hand email. If every query is equally likely, the email on the left is less likely to be accessed. Query-based partitioning estimates the probability of a document occurring in the top K of any user query, and uses this estimate to place the document in a way that minimizes query workload run time.

In this paper, we place the entire document as a unit: it is stored entirely on either top-level or bottom-level storage. The index is partitioned (split) in one of several ways described below.

Document-split index. Here, either all the index entries of a document are in the top-level index, or all of them are in the bottom-level index. This is shown in Figure 4(a), where all the posting list entries of documents 3 and 19 are on bottom-level storage (circled). In this approach, a document omitted from the top-level index can only be accessed by rerunning the query over the bottom-level index.

Term-split index. Here, the posting list for a term is stored either entirely on top-level or entirely on bottom-level storage. A query containing a term whose posting list is on bottom-level storage can only be answered by fetching the term's posting list from bottom-level storage.

Combination index. One can also adopt a combination of document and term splitting. For example, similar to Buttcher et al.'s index compression scheme [2], one can store the top M keywords of every document in the top-level index, and the rest of the keywords in the bottom-level index, where M is either a fixed constant or a constant fraction of the number of keywords in the document. The top terms are identified on the basis of their contribution to the KL divergence [2] between the document and the underlying corpus's word distributions. Another approach is to store only the top N document IDs of each posting list in the top-level index and the rest of the IDs in the bottom-level index, where the document ID order for a term t's posting list is defined by the document term weights $\mathcal{W}(d, t)$. Here N can either be a constant number of elements per posting list, or a constant fraction of the weight of the top document ID. In this paper, we analyze the document- and term-split approaches and compare them empirically to the combination approaches.

Along with the posting lists, term statistics such as IDF values are required to answer queries. We store the term statistics for the entire set of documents with the top-level index. Document abstracts are also all kept on top-level storage. In a term-split index, and in any other splitting strategy that retains a proper subset of the terms of a document in the top-level index, a document d's weight $\mathcal{W}(d, q)$ for a query q can be lower than its weight with an unsplit index. Hence, the document order in a term-split top-level index query answer can be different from that of an unsplit index answer. In a document-split index, the weight of a document in the top-level index will be the same as its weight for the full index, so the document order produced from the top-level index will be consistent with that of an unsplit index.

4.3 Usage Model for Partitioned Indexes

Figure 5 illustrates typical user search behavior with a partitioned index. The user's query is first executed using the top-level index. Since the top-level index is incomplete, the search result may differ from that obtained with an unsplit index. She inspects the returned ranked list and accesses those she considers relevant (ranked 1 and 3). Some of these accesses might require fetching documents from bottom-level storage. If she is not yet satisfied, she then reruns the query over the bottom-level index. In this step, the results are obtained by consolidating the top-level and bottom-level indexes and so are the same as for a query over the unsplit index.

The key parameters for performance are the set of documents that the user clicks on in the top-level search result, and whether she reruns the query on the bottom-level index. This can be captured by the probability distribution $p(d|q)$, which is the probability that the user considers document d relevant for a query q and clicks on it. The probability can also be thought of as the fraction of times the document d is accessed in multiple executions of q, possibly by different users. Because the query is only an approximation of the user's information need, different users may judge different documents to be relevant for the same query.

When she queried unsplit storage, the user's information need was not met until she clicked on a certain set of documents. For a fair comparison of the different splitting schemes, our experiments assume that the user is likely to persist until she has clicked on all those documents, whether the storage is split or unsplit. In other words, $p(d|q)$ is the same for split and unsplit storage. When some of her click-through choices in the unsplit search result do not appear in the top-level result, the user is likely to rerun her query on the bottom-level index and click on the remaining relevant documents.

This assumption is too pessimistic if the user's information need could be met equally well by other documents that appear in the top-level search result and would save her from rerunning her query on the bottom-level index; in this case, while browsing, she is likely to prefer documents that do not require accessing the bottom-level index. In this case, $p(d|q)$ values for documents in top-level and bottom-level results are likely to be higher and lower, respectively, than in unsplit storage. Our cost formulas in the next section do not reflect this differentiation. In other words, the speedups obtained by partitioning in real-world split browsing are likely to be *higher* than what we report in our evaluation.

4.4 Cost Model for a Document-split Index

Under our usage model, the time that the storage server takes to process a user query (*total query response time*) can be written as the sum of the query run time for top-level storage, the query rerun time for bottom-level storage (when required), and the document

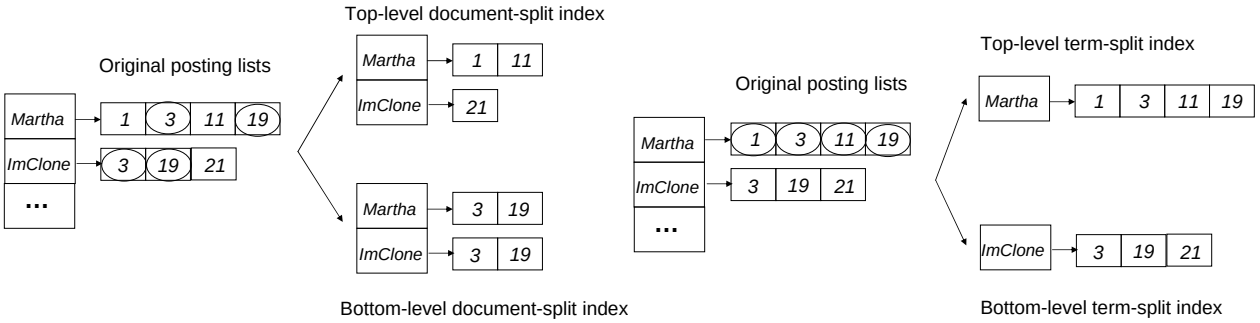


Figure 4: Index partitioning approaches: document-split (left) and term-split (right). Circled entries are destined for the bottom-level index.

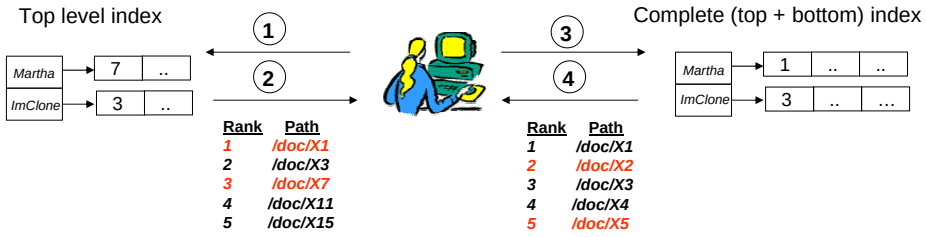


Figure 5: Steps in document access. (1) User queries the top-level index. (2) User clicks on the red documents from the top-level search result (some may be on bottom-level storage). (3) User reruns query on bottom-level index and sees unsplit results. (4) User accesses additional red documents.

access time for each click-through. Let D^t, D^b, and S be the set of top-level, bottom-level, and all documents, respectively. With probability $p(d|q)$, the user clicks through to document d on query q. Let $C^t(d)$ and $C^b(d)$ be the time required to fetch document d from top-level or bottom-level storage, respectively. The total expected document access cost incurred for query q is:

$$C_{doc}(q) = \sum_{d \in D^t} C^t(d)\, p(d|q) + \sum_{d \in D^b} C^b(d)\, p(d|q).$$

The user also incurs costs for running and possibly rerunning q. Let $C^t(q)$ and $C^b(q)$ be the execution times for running q on the top and bottom levels, respectively. Let I^t and I^b be the sets of documents indexed in the top-level and bottom-level index, respectively. Documents in I^b do not appear in the top-level index results, so the user must rerun the query to access them. The probability that any such document will be accessed by the user is the following:

$$1 - \prod_{d \in I^b} (1 - p(d|q)).$$

Thus the expected query cost is:

$$C_{query}(q) = C^t(q) + (1 - \prod_{d \in I^b} (1 - p(d|q))\, C^b(q).$$

The total cost of a set Q of queries is obtained by weighting the above per query cost with the probability $p(q)$ of the query being invoked:

$$Access\,Cost \quad = \quad \sum_{q \in Q} [C_{doc}(q) + C_{query}(q)]\, p(q) \qquad (1)$$

The optimization problem is to choose sets D^t and I^t that minimize this cost, given the following constraint on the top-level storage capacity C:

$$\sum_{d \in D^t} S_{doc}(d) + \sum_{d \in I^t} S_{index}(d) \quad \leq \quad C, \qquad (2)$$

where $S_{doc}(d)$ is the size of document d, and $S_{index}(d)$ is the total size of all of d's index information, including the space for its posting list document IDs plus its document abstract.

This optimization problem can be reduced to a knapsack problem (http://mathworld.wolfram.com/KnapsackProblem.html) , as explained in Figure 6.

4.5 Cost Model for a Term-split Index

The expected document access cost $C_{doc}(q)$ for query q is the same for term-split index and document-split indexes. During query execution, if a query term's posting list is not in top-level storage, its posting list is fetched from bottom-level storage before running the query. Let W^t and W^b be the sets of terms whose posting lists are in top-level and bottom-level storage, respectively. Posting lists of terms in $q \cap W^t$ must be accessed from top-level storage at cost $C^t(w)$, while the remaining are accessed from bottom-level storage with access cost $C^b(w)$. Hence the query cost $C_{query}(q)$ is:

$$C_{query}(q) \quad = \quad \sum_{w \in (q \cap W^t)} C^t(w) + \sum_{w \in (q \cap W^b)} C^b(w). \qquad (3)$$

The objective function to minimize is the total cost $Access\,Cost$ (formula 1), subject to the space constraint

$$\sum_{d \in D^t} S_{doc}(d) + \sum_{w \in W^t} S_{plist}(w) \quad \leq \quad C, \qquad (4)$$

where $S_{plist}(w)$ is the size of the posting list for w. This can also be reduced to a knapsack problem, as explained in Figure 6.

Document-split knapsack derivation

The objective function (formula (1)) is nonlinear in $p(d|q)$, and cannot be solved easily. We simplify formula (1) by ignoring its higher order terms:

$$1 - \prod_{d \in I^b} (1 - p(d|q)) \quad \approx \quad \sum_{d \in I^b} p(d|q). \tag{5}$$

This is a reasonable approximation since the optimal index split would place documents $d \in I^b$ with low $p(d|q)$. With this, $C_{doc}(q)+C_{query}(q)$ becomes

$$= \sum_{d \in D^t} C^t(d)\, p(d|q) + \sum_{d \in D^b} C^b(d)\, p(d|q) + C^t(q) + \sum_{d \in I^b} C^b(q)\, p(d|q)$$

In the above step, we have ignored $C^t(q)$ — it is independent of I^t and D^t and so can be removed from the objective function. Since $S = D^t \bigcup D^b = I^t \bigcup I^b$, the above can be written as

$$\begin{aligned} \sum_{d \in S} C^b(d)\, p(d|q) \quad &- \quad \sum_{d \in D^t} (C^b(d) - C^t(d))\, p(d|q) \\ + \sum_{d \in S} C^b(q) p(d|q) \quad &- \quad \sum_{d \in I^t} C^b(q) * p(d|q) * p(q). \end{aligned}$$

The first and third components are independent of I^t and D^t and can be removed from the objective function. The minimization problem can be changed to maximization by negating the second and fourth terms. Finally, we need to sum over all the queries, weighted by the query probabilities, as given in formula (1). Hence, the final objective function is to maximize

$$\sum_{d \in D^t} (C^b(d) - C^t(d)) \sum_{q \in Q} p(d|q) * p(q) + \sum_{d \in I^t} \sum_{q \in Q} C^b(q) p(d|q) p(q).$$

This is equivalent to a knapsack problem with $2|S|$ objects. $|S|$ of the objects ($d \in S = D^t \cup D^b$) have values $\mathsf{V}_1(d) = (C^b(d) - C^t(d)) \sum_{q \in Q} p(d|q)\, p(q)$ and sizes $S_{doc}(d)$. The remaining $|S|$ objects ($d \in S = I^t \cup I^b$) have values $\mathsf{V}_2(d) = \sum_{q \in Q} C^b(q)\, p(d|q) * p(q)$ and sizes $S_{index}(d)$. The objective is to choose subsets D^t and I^t of the objects that maximize the total value, given space constraint formula (2).

Term-split knapsack derivation

In this approach, the total query cost for a set of queries Q can be obtained by weighting the per query q cost (formula (3)) by the probability of q occurring.

$$\begin{aligned} & \sum_{q \in Q} \{ \sum_{w \in (q \cap W^t)} C^t(w) + \sum_{w \in (q \cap W^b)} C^b(w) \}\ p(q) \\ = & \sum_{w \in W^t} \sum_{\substack{q \in Q \\ (q \cap w) \neq \phi}} C^t(w)\, p(q) + \sum_{w \in W^b} \sum_{\substack{q \in Q \\ (q \cap w) \neq \phi}} C^b(w)\, p(q) \\ = & \sum_{w \in W^t} C^t(w)\, p(w) + \sum_{w \in W^b} C^b(w)\, p(w) \end{aligned}$$

where $p(w) = \sum_{\substack{q \in Q \\ q \cap w \neq \phi}} p(q)$ is the probability of the word w occurring in any query q (as per the distribution $p(q)$). If $W^t \bigcup W^b = W$, where W is the set of words, then the above can be rewritten as follows.

$$\sum_{w \in W^t} (C^t(w) - C^b(w))\, p(w) + \sum_{w \in W} C^b(w)\, p(w)$$

The 2nd summation is a constant independent of W^t. The document access cost $C_{doc}(q)$ is the same as for a document-split index, and can be written as

$$C_{doc}(q) = \sum_{d \in S} C^b(d)\, p(d|q) - \sum_{d \in D^t} (C^b(d) - C^t(d))\, p(d|q).$$

The first summation again is a constant. Ignoring the constant terms, converting the minimization to maximization (by negating) and summing over all the queries, the final optimization problem is to maximize

$$\sum_{d \in D^t} (C^b(d) - C^t(d)) \sum_{q \in Q} p(d|q)\, p(q) + \sum_{w \in W^t} (C^b(w) - C^t(w)) p(w).$$

This is equivalent to a knapsack problem with $|S| + |C|$ objects. $|S|$ of the objects ($d \in S$, the documents) have values $\mathsf{V}_1(d) = (C^b(d) - C^t(d)) \sum_{q \in Q} p(d|q) * p(q)$ and size $S_{doc}(d)$. $|C|$ of the objects ($w \in C$, the posting lists) have values $\mathsf{V}_2(w) = (C^b(w) - C^t(w)) * p(w)$ and size $S_{plist}(w)$. We want to choose sets D^t and W^t that maximize the total value, subject to space constraint formula (4).

Model for $p(d|q)$

Under the model $p(d|q) = c_N * \frac{\mathcal{W}(d,q)}{|q|}$,

$$\sum_{q \in Q} p(d|q) * p(q) \quad = \quad c_N * \sum_{q \in Q} \frac{\mathcal{W}(d,q)}{|q|} * p(q). \tag{6}$$

Now, consider the space of two-word queries, $q = (w_1, w_2)$, where w_1 and w_2 are in the set of keywords. Suppose the query (w_1, w_2) occurs with probability $p(w_1, w_2)$, and we have

$$p(w_1, w_2) = p(w_1) * p(w_2).$$

The following holds:

$$\begin{aligned} & \sum_{q=(w_1,w_2)} \frac{\mathcal{W}(d,q)}{|q|} * p(q) \\ = & \sum_{w_1 \in C} \sum_{w_2 \in C} \frac{(\mathcal{W}(d,w_1) + \mathcal{W}(d,w_2))}{2} * p(w_1) * p(w_2) \qquad \text{Here,} \\ = & \sum_{w \in C} \mathcal{W}(d,w) * p(w) \end{aligned}$$

$p(w)$ is the probability of a term w occurring in a two-word query. Let us use the notation $p(w|2)$ for this. *continued*

continued

For a query with k words, we have

$$\sum_{q=(w_1 \ldots w_k)} \frac{\mathcal{W}(d,q)}{|q|} * p(q) = \sum_{w \in C} \mathcal{W}(d,w) * p(w|k).$$

To estimate formula (6), we have to consider queries of size $k \geq 1$, multiplied by the probability of a query of size k being invoked ($p(|q| = k)$). Hence we can write $\sum_{q \in Q} p(d|q) * p(q)$ as

$$\begin{aligned} & = c_N * \sum_{k \geq 1} \sum_{q=(w_1,\ldots,w_k)} \frac{\mathcal{W}(d,q)}{|q|} * p(q) * p(|q| = k) \\ & = c_N * \sum_{k \geq 1} \sum_{w \in C} \mathcal{W}(d,w) * p(w|k) * p(|q| = k) \\ & = c_N * \sum_{w \in C} \mathcal{W}(d,w) * p_Q(w), \end{aligned}$$

where $p_Q(w) = \sum_{k \geq 1} p(w|k) * p(|q| = k)$ is the probability of the word w occurring in any query.

Figure 6: Derivations

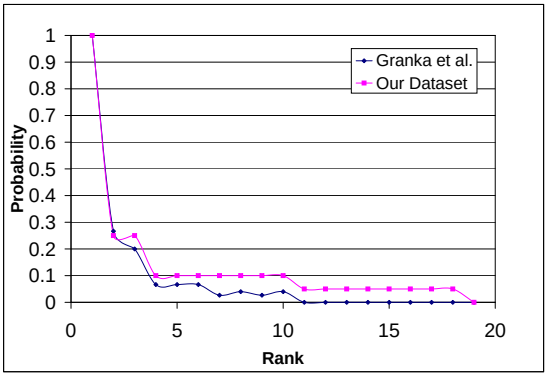


Figure 7: Probability of a document being accessed, as a function of its rank in a query result.

4.6 Estimating p(q) and p(d|q) from a Log

If the set Q of future queries and their frequencies are known in advance, e.g., issued by a high level application, then we can compute query probability $p(q)$ exactly. Otherwise, queries may still follow a definite pattern. For example, there may be a similar set of financial document queries at the end of every quarter and fiscal year. In such cases, previous queries can be a good predictor of future queries. The $p(q)$ values can be learned using a training set of past queries and used for future placement decisions; we call this *non-ad hoc estimation* of $p(q)$ and $p(d|q)$.

The click-through probability $p(d|q)$ can be estimated if the set of documents previously clicked on for q is known. We examined the click-through information for the 300 most popular queries in an IBM intranet search engine workload (described later). We observed that $p(d|q)$ for these queries can be very well approximated as a function of the rank of d in the query result for q. For example, the top 1 or 2 documents are almost always accessed, and documents beyond the top 20 are rarely accessed. Similar results were obtained in a click-through study by Granka et al. [5]. The click-through probabilities for these two cases are shown in Figure 7. To obtain $p(d|q)$, we first run the query q on the target data, obtain the rank of d in the search result, and estimate $p(d|q)$ using the above function.

In a truly ad hoc querying environment, or when an archival system is first installed and no query log is available, query history may not be a good predictor of future queries. In such a case, in theory we can assume that every possible query is equally likely—the uniform query model—and calculate the document values accordingly. However, the space of possible queries becomes prohibitively large: if there are n words in the dictionary, the total number of queries is 2^n. It is infeasible to run all possible queries to obtain the $p(d|q)$ values, as described for the non-ad hoc case. Furthermore, we have experimentally verified on a smaller data set that this approach does not give good partitioning results. Fortunately, there are other alternatives with better performance, as discussed in Section 5.

4.7 Knapsack Partitioning Solution

Since the knapsack packing problem is NP complete, we choose the index fragments and documents that will reside on top-level storage using a greedy knapsack packing algorithm, where the item with the highest value to size ratio is selected first. We present the pseudocode for the resulting Partition() algorithm below. The pseudocode shown here for is for the document-split approach.

Partition() takes the query distribution $p(q)$ as input and calculates the knapsack values for the documents and index entries, as explained in previous sections. It then solves the knapsack problem by selecting the documents or index entries with the largest value density (value by size ratio). The document and index values are computed in the AssignValues() function, which is different for the document-split and term-split approaches. Partition() performs batch processing because documents are normally moved to archival storage periodically in large batches (epochs); we leave for future work the question of how one could partition incrementally as new documents arrive.

Partition(p) p = Query Probability Distribution

```
1: AssignValues(p)
2: for all (d ∈ S) do
3:    val = DocValue[d] {For the document d}
4:    size = S_doc(d)
5:    heap.insert(val/size, ⟨d, doc⟩) {For the index for d}
6:    val = IndexValue[d]
7:    size = S_index(d)
8:    heap.insert(val/size, ⟨d, index⟩)
9: end for
10: avail_space = C;
11: while (avail_space ≥ 0) do
12:    data=heap.PopMin();
13:    if (data == ⟨d, doc⟩) then
14:       Store document d in top level store
15:    else
16:       Store d in top level index
17:    end if
18: end while
```

AssignValues(p) This version is for the document-split approach only. Input parameter p = Query Probability Distribution

```
1: Q = p.domain();
2: for all (q ∈ Q) do
3:    rank_list=RunQuery(q) {Run the query}
4:    for (i = 0; i < rank_list.size(), i++) do
5:       docId = rank_list[i]
6:       prob = ClickProb(i) {Estimate p(d|q) using the rank}
7:       DocValue[docId] += (C^b(d) − C^t(d)) * prob * p(q)
8:       IndexValue[docId] += C^b(q) * prob * p(q)
9:    end for
10: end for
```

5. A SIMPLIFIED QUERY MODEL

In this section, we present a simplified, less accurate model for $p(d|q)$ and $p(q)$ that lets us calculate the knapsack values very efficiently and does not require a query log. In spite of its inaccuracies, experiments presented later show that partitions obtained with the simplified model perform very well in practice. The key properties of the simplified query model are:

- We model $p(d|q)$ as

$$p(d|q) = \frac{c_N * \mathcal{W}(d,q)}{|q|}.$$

  Intuitively, the weight $\mathcal{W}(d,q)$ captures the extent of the match between the document d and the query q, and hence indicates the likelihood of the user clicking through to the document. For multi-term queries, we normalize the weight by the query length. c_N is a normalization constant discussed later.

- The query words are chosen independently. That is, the probability of a query (w_1, w_2) being asked satisfies

$$p(w_1, w_2) = p(w_1)\, p(w_2).$$

As derived in Figure 6, the following holds under the simplified query model:

$$\sum_{q \in Q} p(d|q) * p(q) = \sum_{w \in C} \mathcal{W}(d, w) * p_Q(w).$$

where $p_Q(w)$ is the probability of the word w being involved in a query.
The knapsack values $\mathsf{V}_1(d)$ and $\mathsf{V}_2(d)$ for a document-split index hence become

$$\mathsf{V}_1(d) = (C^b(d) - C^t(d)) * c_N * \sum_{w \in C} \mathcal{W}(d, w) * p_Q(w)$$

$$\mathsf{V}_2(d) = \sum_{q \in Q} C^b(q) * p(d|q) * p(q)$$

Previous studies have shown that for any particular document, there are only a few terms that users are likely to use to search for it [2]. Under the simplified query model, these important terms are equally likely to be queried, while the other terms are never queried. That is, $p_Q(w) = c_Q$ (a constant), if there exists a document d such that $w \in Key(d)$; and $p_Q(w) = 0$ otherwise, where $Key(d)$ is the set of important terms for the document. We observed empirically that identifying the important terms on the basis of their $TF * IDF$ scores is more effective than using KL-divergence. In our experiments, we set $Key(d)$ to be the 10 terms w of document d with the highest $\mathcal{W}(d, w)$ weights. We estimate the time to execute a $|q|$ keyword query on bottom-level storage as $|q| * C^b_{seek}$, where C^b_{seek} is a constant dependent on the seek time of the device (we validate this in Section 6.2). Hence for a document-split index, the knapsack values can be further simplified to

$$\mathsf{V}_1(d) = c_N * (C^b(d) - C^t(d)) * c_Q * \sum_{w \in Top10} \mathcal{W}(d, w).$$

$$\mathsf{V}_2(d) = avg\_q\_sz * c_N * C^b_{seek} * c_Q * \sum_{w \in Top10} \mathcal{W}(d, w).$$

where avg_q_sz is the average number of keywords in a query. The same document values $\mathsf{V}_1(d)$ apply for a term-split index, while the term values are:

$$\mathsf{V}_2(w) = \quad c_Q * (C^b(w) - C^t(w)), \quad if \;\; w \in Top10$$
$$\mathsf{V}_2(w) = \quad 0, \quad otherwise.$$

Estimating c_Q, c_N, and avg_q_sz. As mentioned earlier, we pack items onto top-level storage using a knapsack heuristic, where items are packed in order of their value to size ratio. The knapsack solution for a document-split index, hence, is independent of the $c_N * c_Q$ constant scaling of the item values. Similarly, the knapsack solution for a term-split index is independent of the scaling factor c_Q. For the latter, we empirically set $c_N = 10$, which we found to provide the best performance for a term-split index.

We found that $avg\_q\_sz = 2.45$ in our real-world query log (described later). We believe that this value is independent of any specific query log, and is related to human behavior with search engines. For example, Craig et al. reported an average query size of 2.35 in their study of 1 billion queries from the Altavista search engine [11], which is close to our estimate.

Advantages of the Simplified Query Model. The knapsack values V_1 and V_2 involve computing $\sum_{w \in Top10} \mathcal{W}(d, w)$. This can be computed for each document independently, without considering other documents or the query history. Thus it is much more efficient than our other approach to computing knapsack values, which requires computing the rank of each document for every query in the query training data. As the simplified query model does not require any prior query log information, it is easy to deploy.

The experimental results in Section 6 show that the simplified query model gives excellent partitioning results. This is because the document word weight $\mathcal{W}(d, w)$ does model the importance of the word w. A document with only common words (and hence low $\mathcal{W}(d, w)$) gets a lower knapsack value $V_1(d)$ than a document with rare words.

6. EXPERIMENTAL VALIDATION

6.1 Data and Methodology

Confidentiality concerns make it difficult to obtain query workloads for corporate email and documents, which are the primary targets of this work. We know of only one publicly available business email archive, and it has no query log [7]. Since business intranet queries typically search for specific information, such as business or policy documents or web pages, we believe that they are a reasonable approximation to a document archive workload. Therefore, our experiments use a collection of one million documents crawled by an intranet search engine at a large company. For the query workload, we use 300,000 actual user queries logged by the same search engine.

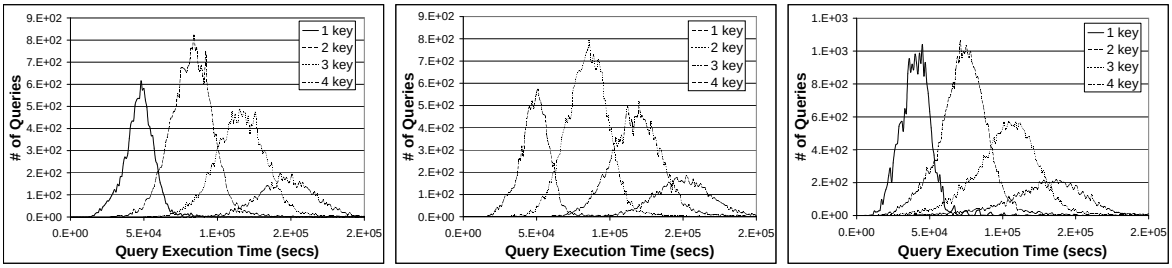
Our experiments use three models for estimating $p(q)$, the probability of query q:

- *Exact query model.* Learn $p(q)$ exactly, from the 300,000 queries. This gives the best possible estimate of document and query hits on top level storage, and is used as a baseline to compare other techniques.

- *Learn $p(q)$ using the first 10% of the query log.* This reflects the scenario where query probabilities are estimated using past queries. The remainder of the log is used as the query workload.

- Simplified query model. We have no prior knowledge of $p(q)$, as discussed in Section 5.

We evaluated query-based partitioning against the 300,000-query log, for each of the three $p(q)$ estimates. After splitting the index and documents across the top and bottom level archival storage, we ran the queries in the log and computed the document accesses and query runs on the top-level and bottom-level storage under the usage model of Section 4.3. That is, a document d is accessed with probability $p(d|q)$; documents not on the top-level storage are fetched from the bottom-level storage; and the query q is rerun with probability $(1 - \prod_{d \in I}(1 - p(d|q)))$, where I is the set of documents that do not appear in the top query results. Based on these probabilities, we measure the expected document access and query cost on top-level and bottom-level storage.



(a) Random Queries (b) Log Queries (c) Document Accesses

Figure 8: Validation of the cost model

6.2 Cost Model Validation

Our experimental evaluation is based on the following cost estimates:

- The time to fetch a document is approximated by C_{seek}, where C_{seek} is a constant dependent on the average seek (including rotational latency) time of the device. Specifically, it is a constant multiple of the device seek time.
- The time to execute a $|q|$ keyword query is $|q| * C_{seek}$.

Both estimates assume that transfer time is negligible compared to seek time. This is certainly true for tape drives, where the seek time exceeds the transfer time by almost an order of magnitude, even for files that are megabytes in size. To validate the cost model for an ATA disk drive, we built an index over our intranet documents and measured the time to execute different query workloads. Figure 8(a) plots the query runtime distribution for the entire query log. The different curves correspond to different numbers of keywords in the query. The query distribution curves in the figure closely resemble a Gaussian distribution with means of 46, 87, 121, and 151 msec for 1, 2, 3, and 4-keyword queries, respectively. The mean execution time for 2, 3, and 4-word queries is close to 2, 3, and 4 times the mean for single-keyword queries. Figure 8(b) shows the query execution times for only those queries in the log which are rerun on bottom-level storage, for the storage configuration where the top-level storage is 40% of the total storage size. The distributions here are very close to the previous distributions, and the curves for other storage configurations were similar. Through a separate experiment, we computed that the average time to read a single 1 KB file (for which the transfer time is negligible) is 44 msec. We use this value as C_{seek} in the cost model. In practice, accessing a file generally requires one seek to fetch the inode and one seek to fetch the data block. So, 44 msec is 2 times the seek time of the underlying device. We use the same multiplicative factor of 2 times the seek time in the rest of our experiments. Furthermore, the knapsack solutions are independent of this constant scaling factor.

To model the document access cost, we materialized the intranet documents on disk. For each query in the intranet query log, we accessed all the relevant documents that could have been accessed by the user, according to the probability distribution of $p(d|q)$. In other words, the number of documents accessed for a particular query is a random variable. Figure 8(c) plots the document access time as a function of the number of documents accessed. The curves are very similar to the curves in Figures 8(a-b). Again, this validates our model that the time to access a document can be approximated by C_{seek}.

6.3 Disk-Tape Archival Storage Architecture

Our experiments consider three performance measures: the *document access cost ratio*, which is the ratio of the document access cost when the documents and index are partitioned to the document access cost with unsplit storage; the *query access cost ratio*, the corresponding ratio for query execution times without any clickthroughs; and *total access cost ratio*, the corresponding ratio for document access plus query costs. The access costs are computed using the formulas in Sections 4.4 and 4.5 for the document-split and term-split indexes, respectively.

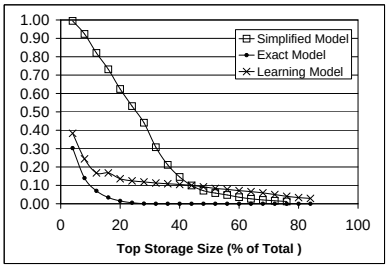
Our first experiment uses an archival storage architecture consisting of fast disk-based top-level storage and slow tape based bottom-level storage. The goal of this experiment is to evaluate the speedup that can be obtained by the use of top-level storage, as a function of its size. As explained in Section 6.2, a tape seek is more than 10 times slower than a disk seek; thus we let 0 be the cost $C^t(d)$ to fetch a document from top-level storage and the cost $C^t(q)$ to execute a query on top-level storage. As justified in Section 6.2, we compute the disk seek and latency time, and omit the transfer time. Hence $C^b(d)$ and $C^b(q)$ are equal to C_{seek} and $C_{seek}|q|$, respectively.

Figures 9(a) through 9(c) show the document, query, and total access cost ratios, respectively, for document-split partitioning. The three graphs correspond to the exact, simplified, and learning-based query models for estimating $p(q)$. In these figures, the simplified query model performs poorly compared to the other two models when top-level storage is small (under 20% of the total data size). Many documents contain rare terms, but these terms may never be queried in the log. Such documents may not be in the top K for any query in our log, although they have high access probability under the assumption that every term is equally likely to be queried. Such documents account for 30-40% of the total storage. Under the simplified query model, the document-split index fails to filter out those documents, and hence performs poorly compared to the schemes that take the query log into account.

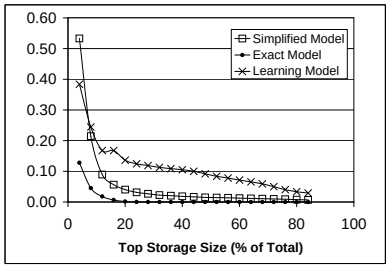
For larger top-level storage sizes, the simplified query model performs close to the exact query model. For example, it provides a factor of 10 speedup when 40% of the data is placed on the top-level storage. It even performs better than the learning-based query model. The latter approach assigns zero weight to documents that do not have keywords in the first 10% of the log. Some of those documents are queried and accessed in the rest of the query log. The simplified query model assigns more appropriate weights to these documents.

Figures 9(d) through 9(f) show the corresponding performance numbers for term-split partitioning. Under the simplified query

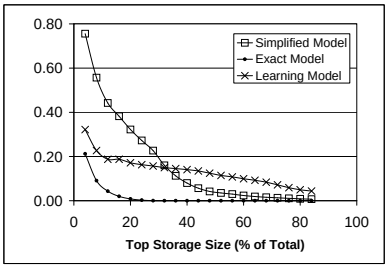
Disk-tape archival storage architecture



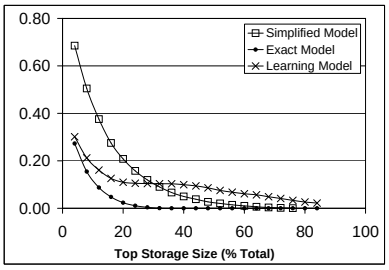
(a) Document access cost ratio (document-split partitioning)



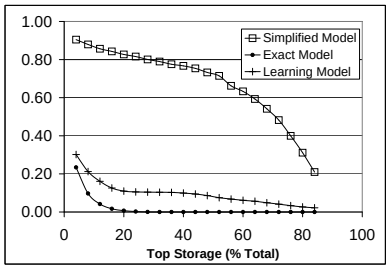
(b) Query access cost ratio (document-split partitioning)



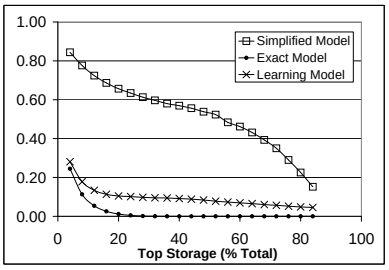
(c) Total access cost ratio (document-split partitioning)



(d) Document access cost ratio (term-split partitioning)

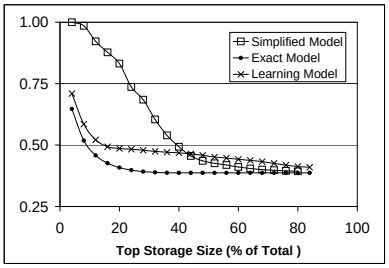


(e) Query access cost ratio (term-split partitioning)

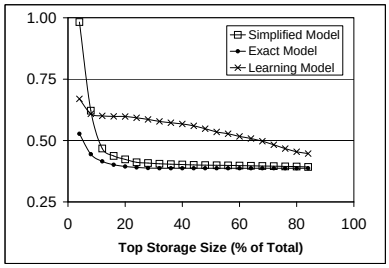


(f) Total access cost ratio (term-split partitioning)

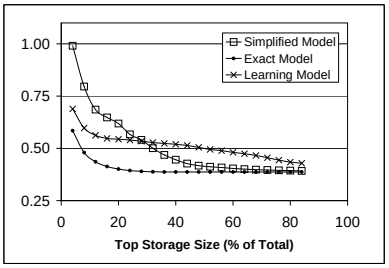
Disk-disk archival storage architecture



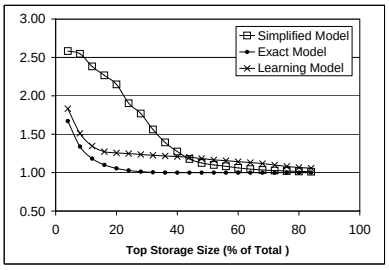
(g) Document access cost ratio (document-split partitioning)



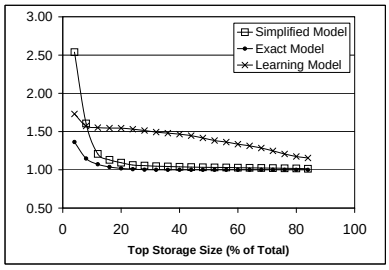
(h) Query access cost ratio (document-split partitioning)



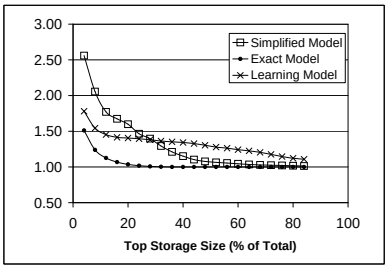
(i) Total access cost ratio (document-split partitioning)



(j) Document access cost slowdown (document-split partitioning)



(k) Query access cost slowdown (document-split partitioning)



(l) Total access cost slowdown (document-split partitioning)

Figure 9: Ratio of access cost with partitioning to access cost without partitioning

model, term splitting is inferior to document splitting. Term-split partitioning gives lower values to keywords with long posting lists (low ratio of query probability to list length), and so is less likely to place them in the top-level index. In our workload, some keywords with long posting lists are queried very often. Queries involving those keywords require frequent access to bottom-level storage and hence contribute to the poor performance of term-split indexes. Document-split partitioning, on the other hand, gives lower values to documents that only contain very common terms. Such documents are rarely accessed. However, term-split and document-split partitioning perform equally well when the exact query probabilities are known.

6.4 Disk-Disk Archival Storage Architecture

The next set of experiments employs a storage architecture with fast SCSI disk based top-level storage and slow IDE disk based bottom-level storage. Ignoring the transfer time, the cost of accessing a document or posting list from top-level and bottom-level storage is 4.9 and 12.7 msec, respectively. These performance parameters are summarized below.

Model	Interface	Seek	Latency	Transfer
Seagate Barracuda	ATA	8.5 msec	4.2 msec	40 MB/s
Seagate Savvio	SAS	2.9 msec	2.0 msec	80 MB/s

Figures 9(g) through 9(i) show the document, query, and total access cost ratios, respectively. The figures show that addition of top-level storage that can hold 30% of the data speeds up the workload by a factor of 2.5. This is close to the maximum achievable speedup of 2.58 ($\frac{12.66}{4.9}$) for this storage architecture.

Figures 9(j-l) illustrate the same result in a different manner. They plot the slowdown as the ratio of the total access cost when the index and documents are all on top-level storage to the total access cost when they are partitioned, as a function of the top-level storage size.

6.5 Other Partitioning Heuristics

We carried out experiments to compare document-split partitioning with the other partitioning heuristics described in Section 4.2. These heuristics do not utilize any prior knowledge of the likely queries, so we compare them against document-split partitioning with the simplified query model. For these simulations, we created the top-level index by storing the top K keywords per document or the top K document IDs per posting list, for different values of K. We select the documents occupying the remaining top-level storage space by solving the knapsack problem with item values $\mathsf{V}_1(d)$ and sizes $S_{doc}(d)$, as described in Section 4.4.

Figure 10(a) shows the total access cost ratio for document-split partitioning with a disk-tape storage architecture, where the top K terms of each document are retained in the top-level index. The "20/doc" and "0.7*#keywords" lines correspond to the case where the top 20 terms and top 70% of the terms in the document are retained in the top-level index, respectively. (For clarity, we have omitted the lines corresponding to other values of K.) The figure shows that the optimal value of K (lowest total access cost ratio) depends on the top-level storage size. For example, $K = 20$ is optimal when the top-level storage size is under 40% of the total storage, while for larger top-level storage sizes, $K = 0.7$ is optimal and $K = 20$ is substantially suboptimal. Document-split partitioning is close to optimal under all configurations. Figure 10(c) for the disk-disk storage hierarchy and Figures 10(b) and 10(d) for the strategy that retains the top K terms of each posting list exhibit similar behavior. The strategies labeled "k*Max" retain, for every term t, every document d whose term document weight $W(d,t)$ is more than $k < 1$ times the maximum term document weight for t.

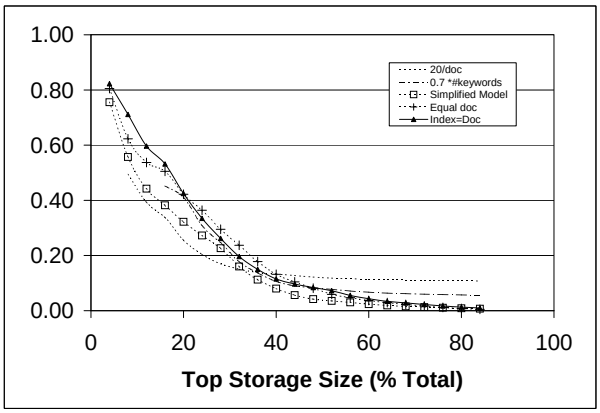
Figure 10 also shows the results for two other splitting heuristics. The "Index=doc" line shows the case where the top-level index is created only over the documents stored in the top-level storage. This is inferior to document-split partitioning, as explained in Section 4.1. The "Equal doc" line in Figure 10 is for the case where we assume each document is equally likely to be accessed. The document and index partitioning is decided by solving the knapsack problem, as before. Under this model, the smallest documents have the highest ratio of access probability to size, and hence are more likely to occupy top-level storage. This partitioning also works quite well (much better than random). Intuitively, this is because users only query a few important keywords per document, regardless of the document size. A document *A* twice the size of document *B* is not twice as likely to be queried by the user. On the other hand, *A* takes up twice as much space. Hence it makes sense to give priority to *B* for top-level storage. Of course, we are ignoring the document contents when we assume each document is equally likely to be queried; thus this scheme is slower than document-split partitioning.

7. PREFETCHING

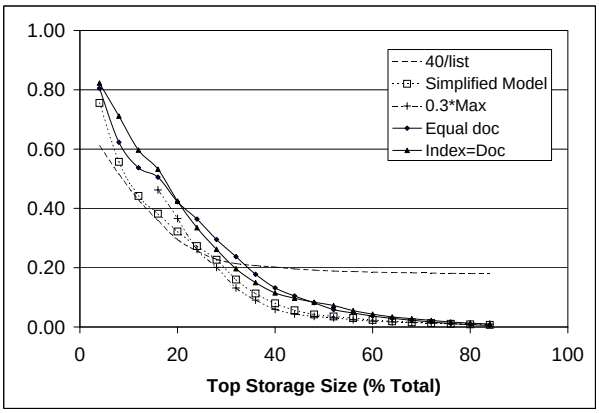
The cost models in Sections 4.4 and 4.5 assume that every document access and posting list fetch is performed in response to a user request. This allowed us to compute the response time as the sum of document access and query run times. In practice, storage devices often perform prefetching. For example, when a block is fetched, several subsequent blocks are also fetched at virtually no additional I/O penalty (i.e., without incurring any extra seek or rotation latency). Prefetching helps reduce the application-visible I/O latency for workloads whose file accesses exhibit strong locality. However, the documents matching a given keyword query are unlikely to exhibit significant spatial locality, because documents are ordered on archival disk/tape by arrival time.

Query-aware prefetching is more promising. For example, the indexing application can prefetch result documents from bottom-level storage while the user inspects top-level documents. This can reduce user-visible I/O cost by hiding bottom-level I/O behind user activities. In the best possible case, all the I/O costs are hidden behind user activity time, and the total time the user spends interacting with the search engine goes down by up to a factor of 2.

An important consideration for prefetching is when and how much to prefetch. In the worst case, the user does not click on any prefetched document, and the storage server's workload is needlessly increased. One option is to prefetch all the bottom-level documents listed on the first page of the ranked result (top-20) from the top-level index, while the user examines the top-level documents. Another option is to run every query in parallel on the top-level and bottom-level indexes. This can greatly reduce user-visible latency for rerun queries. Figure 11(a) shows the number of document and posting list accesses to bottom-level storage for the tape-disk storage architecture under these two schemes, and with no prefetching. The figure shows that for small top-level storage sizes, prefetching increases the number of document accesses to bottom-level storage by a factor of 4 or more. If query throughput is a bottleneck for bottom-level storage, a factor of 3-5 drop in query throughput for at most a factor of 2 decrease in user interaction and query time is unlikely to be acceptable. With larger top-level storage sizes, most queries are answered from top-level storage and hence the benefit of prefetching from bottom-level storage is quite limited.

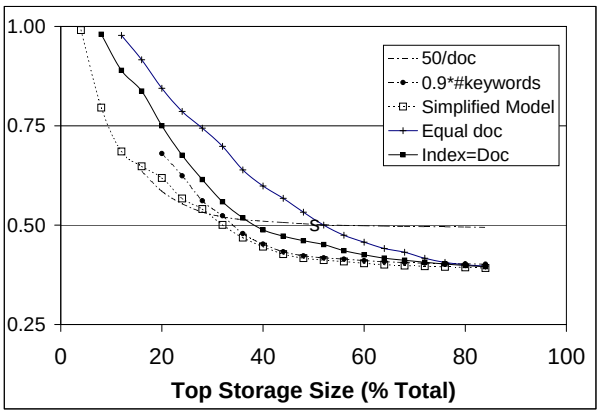


(a) Top keywords per document

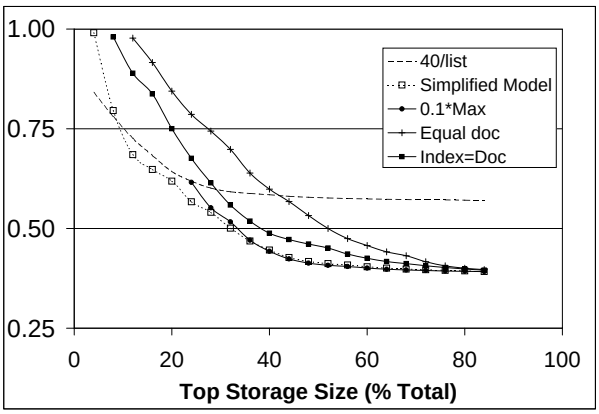


(b) Top elements per posting list

Tape-disk archival storage architecture



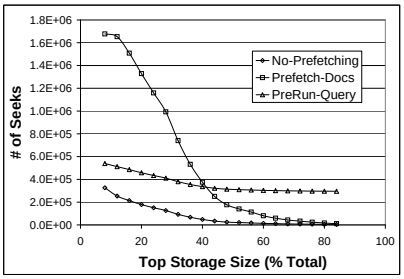
(c) Top keywords per document



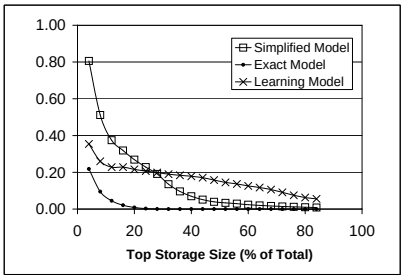
(d) Top elements per posting list

Disk-disk archival storage architecture

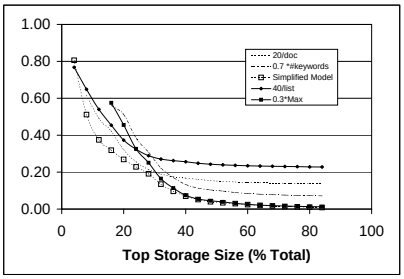
Figure 10: Ratio of total access cost with partitioning to cost without partitioning, for several partitioning heuristics



(a) Prefetching effect on bottom-level storage



(b) Total access cost ratio (document-split partitioning)



(c) Ratio of total access cost with partitioning to cost without partitioning, for several partitioning schemes

Figure 11: Effect of prefetching on storage I/O bandwidth. (a) Disk-tape storage with no prefetching, prefetching bottom-level documents in top-20 of top-level index result, and running query in parallel on top- and bottom-level storage. (b-c) Disk-tape storage with prefetching of the top-ranked document.

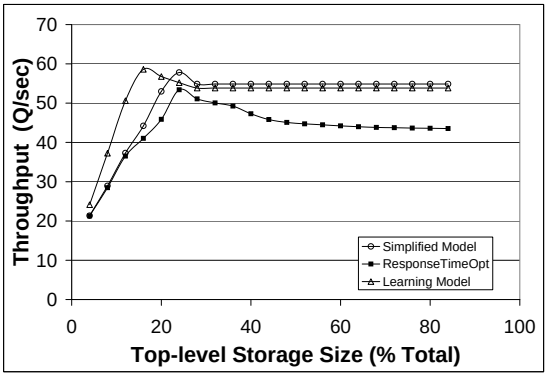


Figure 12: Maximum query throughput at the storage server

A radically different option is to prefetch only the top ranked document from wherever it resides. This can hide the I/O time to access the first document behind the time the user spends inspecting the document abstracts of the search result. The user almost always clicks on the top ranked document, so this prefetching scheme introduces very few additional I/Os. We modified the cost models of Section 4.4 for this prefetching scheme. Figure 11(b) shows the resulting performance of the partitions produced under all three query models, and Figure 11(c) compares the document-split results with the other heuristic partitioning schemes. These figures are quantitatively very similar to Figures 9(c) and 10(a), respectively. Overall, this prefetching scheme does not have a significant effect on the relative performance of the different partitioning techniques. We leave the investigation of the effect of other more sophisticated prefetching schemes as future work.

8. THROUGHPUT COST MODEL

The cost functions in Sections 4.4 and 4.5 model the storage server's response time for a user query. If documents and posting lists are accessed serially on a disk-disk archival storage server, then the same formulas help us to model query throughput: the cost function gives us the total (wall clock) processing time spent by the storage server for executing one query. For the disk-tape architecture, the I/O time for fetching a document from disk is negligible compared to fetching it from tape, so the cost function gives the total I/O time for the tape drive. In these cases, the reciprocal of the cost function gives the query throughput that the storage server can support. Minimizing this cost function maximizes the throughput.

However, in practice an archival storage server can process I/O requests on top-level and bottom-level storage in parallel. The query throughput in this case can be estimated as follows. The total I/O time required to process a query on top-level storage is:

$$C_{top}(q) = C^t(q) + \sum_{d \in D^t} C^t(d)\, p(d|q)$$

Similarly, the total I/O time for bottom-level storage is:

$$C_{bot}(q) = (1 - \prod_{d \in I^b} (1 - p(d|q))\, C^b(q) + \sum_{d \in D^b} C^b(d)\, p(d|q).$$

The total time required to process a workload Q of queries on top-level and bottom-level storage is given by:

$$C_{top} = \sum_{q \in Q} C^t(q) * p(q) \quad C_{bot} = \sum_{q \in Q} C^b(q) * p(q)$$

If I/O requests can be executed in parallel, the wall clock time to process workload Q is the maximum of the above two:

$$C_{wall} = Max(C_{top}, C_{bot})$$

We must minimize C_{wall} to maximize query throughput, subject to the space constraint on top-level storage (formula (2)). As this problem is NP complete, we used a greedy heuristic to solve this partitioning problem and then carried out simulation experiments to evaluate the query throughput.

We partitioned the documents and index using the simplified query model and the learning query model, where the first 10% of the query log is used to learn $p(q)$. Figure 12 shows the query throughput as a function of the size of the top-level storage. The "ResponseTimeOpt" curve shows the throughput when the partitioning is created with the goal of minimizing the response time under the simplified query model using document-split partitioning. The other two curves show the throughput using the greedy partitioning heuristic for throughput optimization, under the simplified and learning query models. For the two throughput optimization curves, the throughput increases as the top-level storage size increases and then flattens off. The flattening off point is where the total I/O time on the top-level storage is equal to the I/O time on the bottom-level storage. On the other hand, query throughput for the response time curve gradually falls off as the top-level storage size increases. This is because to minimize the total response time (the sum of top-level and bottom-level access time), the latter approach assigns documents to top-level storage even when the total top-level I/O time exceeds the bottom-level I/O time.

9. CONCLUSION

In this paper, we proposed a novel technique for partitioning documents and inverted indexes across an archival storage hierarchy, by exploiting the top-K query result occurrence probabilities of documents. We developed two different partitioning techniques, along with several variants. In *document-split partitioning*, the index entries for a document are stored either all in the top-level index or all in the bottom-level index. With *term-split partitioning*, a term posting list is stored either entirely in the top-level index or entirely in the bottom-level index. For these two partitioning approaches, we determined the analytic relationship between the space occupied by the index on top-level storage and its query performance. We solved the optimization problem associated with this relationship to determine what fraction of top-level storage should be allocated to the index and what fraction to the documents, and then used a greedy approach to choose which documents and index entries to place on top-level storage. Experimental evaluation of these partitioning techniques on a real-world intranet workload of 1 million documents and 300,000 queries showed that when a training data set of past queries is available, both partitioning techniques perform almost as well on the query workload as an oracle-based optimal partitioning strategy that has complete knowledge of all possible future queries. This holds whether the archival storage architecture consists of fast and slow disks, or disks and tape.

To handle the case where no information about likely future queries is available, we proposed a simplified query model that estimates the chance that a document will appear in the top-K results for some query, based on the intrinsic characteristics of the document. Even when its partitioning decisions are based on the simplified query model, the document-split partitioning approach performs within a factor of 2 of the optimal partitioning strategy, for the intranet query workload. On the other hand, term-split partitioning performs relatively poorly in this situation; it is a factor of 2-4 slower than document-split partitioning on the query workload. We also compared document-split partitioning to other heuristic strate-

gies for splitting the index, such as using top-level storage to hold the top terms of every document or the top document IDs from every posting list. Even after substantial hand-tuning of parameters, the speedups offered by these heuristic strategies are within 10% of that obtained by document-split partitioning. Since document-split partitioning does not require any parameter tuning, we conclude that it is the partitioning method of choice for most archival query workloads and storage architectures.

10. REFERENCES

[1] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *SIGIR*, 2007.

[2] S. Buttcher and C. L. A. Clarke. A document-centric approach to static index pruning in text retrieval systems. In *CIKM*, 2006.

[3] C. Faloutsos. Access methods for text. *ACM Computing Surveys*, 17:49–74, 1985.

[4] R. Glazer. Measuring the value of information: The information-intensive organization. *IBM Systems Journal*, 32(1):99–110, 1993.

[5] L. Granka, T. Joachims, and G. Gay. Eye-Tracking Analysis of User Behavior in WWW Search. In *SIGIR*, 2004.

[6] C. Johnson. ILM Case Study: Complete Data Lifecycle Management Solution. http://www.snia.org/, 2004.

[7] B. Klimt and Y. Yang. Introducing the Enron Corpus. In *Conference on Email and Anti-Spam (CEAS)*, 2004.

[8] S. Mitra, W. Hsu, and M. Winslett. Trustworthy Keyword Search for Regulatory Compliance. In *VLDB*, 2006.

[9] A. Ntoulas and J. Cho. Pruning policies for two-tiered inverted index with correctness guarantee. In *SIGIR*, 2007.

[10] E. Pierre. Introduction to ILM: A tutorial. http://www.snia.org/, 2004.

[11] C. Silverstein, H. Marais, M. Henzinger, and M. Moricz. Analysis of a very large web search engine query log. *SIGIR Forum*, 33(1):6–12, 1999.

[12] A. Singhal. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

[13] A. Soffer, D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovici, and Y. S. Maarek. Static index pruning for information retrieval systems. In *SIGIR*, 2001.

[14] I. H. Wittenm, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufman, San Francisco, CA, 1999.