Data Structures for Efficient Broker Implementation*

To Appear, ACM Transactions on Information Systems, 1996

Anthony Tomasic† INRIA

Luis Gravano‡ Stanford University

Calvin Lue§ IBM Almaden

Peter Schwarz¶ IBM Almaden

Laura Haas IBM Almaden

Abstract

With the profusion of text databases on the Internet, it is becoming increasingly hard to find the most useful databases for a given query. To attack this problem, several existing and proposed systems employ brokers to direct user queries, using a local database of summary information about the available databases. This summary information must effectively distinguish relevant databases, and must be compact while allowing efficient access. We offer evidence that one broker, *GlOSS*, can be effective at locating databases of interest even in a system of hundreds of databases, and examine the performance of accessing the *GlOSS* summaries for two promising storage methods: the grid file and partitioned hashing. We show that both methods can be tuned to provide good performance for a particular workload (within a broad range of workloads), and discuss the tradeoffs between the two data structures. As a side effect of our work, we show that grid files are more broadly applicable than previously thought; in particular, we show that by varying the policies used to construct the grid file we can provide good performance for a wide range of workloads even when storing highly skewed data.

1 Introduction

The last few years have seen an explosion in the amount of information that is available online. The falling costs of storage, processing, and communications have all contributed to this explosion, as has the emergence of the infrastructure provided by the World-Wide Web and its associated applications. Increasingly, the key issue is not *whether* some piece of information is available online, but *where*. As a result, an emerging area of research concerns *brokers*, systems that help users locate the text databases that are most likely to contain answers to their queries. To perform this service, brokers use summary information about the available databases. Brokers must be able both to query and to update this summary information. A central problem in broker design is to find a representation for summary information that is both effective in its ability to select appropriate information resources, and efficient to query and maintain.

*This work was partially supported by ARPA Contract F33615-93-1-1339.

†INRIA Rocquencourt, 78153 Le Chesnay, France. E-mail: `Anthony.Tomasic@inria.fr`

‡Computer Science Department, Stanford University, Stanford, CA 94305-9040, USA. E-mail: `gravano@cs.stanford.edu`

§Current address: Trident Systems, Sunnyvale, CA, USA. E-mail: `clue@tridmicr.com`

¶Department K55/801, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099, USA. E-mail: `schwarz@almaden.ibm.com, laura@almaden.ibm.com`

GlOSS (*Glossary-Of-Servers Server*) [16, 17] is one broker that keeps database summaries to choose the most promising databases for a given query. Initial studies of *GlOSS* are encouraging. Experiments with a small number of databases indicate that although the *GlOSS* summaries are orders of magnitude smaller than the information that they summarize, they contain enough information to select the best databases for a query. In this article, we show that the *GlOSS* summaries can be employed as the representation for summary information in a large scale system. In particular, we offer evidence that *GlOSS* can effectively locate databases of interest even in a system of hundreds of databases. Our metric for effectiveness is based on selecting databases that contain the largest number of matching documents for a simple Boolean query. Second, we suggest appropriate data structures for storing such large scale *GlOSS* summaries.

We experiment with two data structures, partitioned (multi-attribute) hashing and the grid file. Partitioned hashing offers the best average case performance for a wide range of workloads – if the number of hash buckets is chosen correctly. However, the grid file performs well, and grows more gracefully as the number or size of the summaries increases.

Grid files were developed to store data keyed in multiple dimensions, and are typically employed for data that is fairly uniformly distributed. The *GlOSS* summaries we store are highly skewed. We show that by varying the splitting policy used to construct a grid file we can provide good performance for a wide range of workloads even when storing such highly skewed data. Thus, as a side effect of our work, we demonstrate that grid files are more generally applicable than previously believed, and provide an exploration of the effect of different splitting policies on grid file performance.

In summary, this paper studies an emerging problem in the construction of distributed information retrieval systems, namely, the *performance of brokers* for accessing and updating summary information. Section 2 reviews the *GlOSS* representation of summary information. Section 3 discusses *GlOSS*'s effectiveness when there are large numbers of databases. The next four sections focus on choosing a storage method for the summary information. Section 4 discusses the issues involved in choosing a storage method, and describes some alternatives. Section 5 introduces the idea of using a grid file to store the *GlOSS* summaries, describes various splitting policies for managing grid file growth, and presents a simulation study of grid file performance over a range of workloads, for several splitting policies. Section 6 examines partitioned hashing as an alternative method for efficiently storing *GlOSS* summaries. Section 7 compares the results from the two storage methods, and explains why we recommend the grid file. Section 8 positions our work with respect to other work on brokers, and the last section summarizes our results and our conclusions, and provides some ideas for future work.

2 *GlOSS*–Glossary-Of-Servers Server

In this section we briefly describe how *GlOSS* helps users choose databases at which a query should be evaluated. Users first submit their query to *GlOSS* to obtain a ranking of the databases according to their potential usefulness for the given query. The information used by *GlOSS* to produce this ranking consists of a vector that indicates how many documents in the database contain each word in the database vocabulary, and a count of the total number of documents in the database [16]. This summary information is much smaller than the complete contents of the database, so this approach scales well as the number of available databases increases.

Table 1 shows a portion of the *GlOSS* summaries for two databases. Each row corresponds to a word and each column to a database. For example, the word "information" appears in 1234 documents in database db_1, and in 30 documents in database db_2. The last row of the table shows the total number of documents in each database: database db_1 has 1234 documents, while database

word	*database*	
	db_1	db_2
information	1234	30
retrieval	89	300
documents	1234	1000

Table 1: Part of the *GlOSS* summaries of two databases.

db_2 has 1000 documents.

To rank the databases for a given query, *GlOSS* estimates the number of documents that match the query at each database. *GlOSS* can produce these estimates from the *GlOSS* summaries in a variety of ways. One possibility for *GlOSS* is to assume that the query words appear in documents following independent and uniform probability distributions, and to estimate the number of documents matching a query at a database accordingly. For example, for query "*information AND retrieval*," the expected number of matches in db_1 (using the *GlOSS* summary information of Table 1) is $\frac{1234}{1234} \cdot \frac{89}{1234} \cdot 1234 = 89$, and the expected number of matches in db_2 is $\frac{30}{1000} \cdot \frac{300}{1000} \cdot 1000 = 9$. *GlOSS* would then return db_1 as the most promising database for the query, followed by db_2. Several other estimation functions are given in [17].

As mentioned in the introduction, *GlOSS* can be measured with respect to its effectiveness in locating the best databases for a given query, and it can be measured in terms of its computational performance. In the next section we study the effectiveness of *GlOSS*. The rest of the article is devoted to computational performance.

3 Effectiveness of *GlOSS*

Given a set of candidate databases and a set of queries, we explored the ability of *GlOSS* to suggest appropriate databases for each query. The original *GlOSS* studies [16, 17] tested *GlOSS*'s ability to select among six databases. To be sure that *GlOSS* would be useful as a large scale broker, we scaled up the number of databases by about two orders of magnitude. In this section, we describe a set of experiments that demonstrate that *GlOSS* can select relevant databases effectively from among a large set of candidates. We present a metric for evaluating how closely the list of databases suggested by *GlOSS* corresponds to an "optimal" list, and evaluate *GlOSS* based on this metric.

For our experiments, we used as data the complete set of United States patents for 1991. Each patent issued is described by an entry that includes various attributes (e.g., names of the patent owners, issuing date) as well as a text description of the patent. The total size of the patent data is 3.4 gigabytes. We divided the patents into 500 databases by first partitioning them into fifty groups based on date of issue, and then dividing each of these groups into ten subgroups, based on the high order digit of a subject-related patent classification code. This partitioning scheme gave databases that ranged in size by an order of magnitude, and were at least somewhat differentiated by subject. Both properties are ones we would expect to see in a real distributed environment.

For test queries, we used a set of 3,719 queries submitted against the INSPEC database offered by Stanford University through its FOLIO boolean information retrieval system.[1] INSPEC is not a patent database, but it covers a similar range of technical subjects, so we expected a fair number of hits against our patent data. Each query is a boolean conjunction of one or more words, e.g.,

[1]For more information on the query traces, see reference [39], which provides detailed statistics for similar traces from the same system.

N	Mean	Std. Dev.
1	0.712	0.392
2	0.725	0.350
3	0.730	0.336
4	0.736	0.326
5	0.744	0.319
6	0.750	0.312
7	0.755	0.307
8	0.758	0.303
9	0.764	0.299
10	0.769	0.294

Table 2: Normalized cumulative recall for 500 databases for the INSPEC trace.

"*microwave AND interferometer*." A document is considered to match a query if it contains all the words in the conjunction.

To test *GlOSS*'s ability to locate the databases with the greatest number of matching documents, we compared its recommendations to those of an "omniscient" database selection mechanism implemented using a full-text index of the contents of our 500 patent databases. For each query, we found the exact number of matching documents in each database, using the full-text index, and ranked the databases accordingly. We compared this ranking with the ranking suggested by *GlOSS* by calculating, for various values of N, the ratio between the total number of matching documents in the top N databases recommended by *GlOSS* and the total number of matching documents in the N best databases according to the ideal ranking. This metric, the *normalized cumulative recall*, approaches 1.0 as N approaches 500, the number of databases, but is most interesting when N is small. Because this metric is not meaningful for queries with no matching documents in any database, we eliminated such queries, reducing the number of queries in our sample to 3,286.

Table 2 shows the results of this experiment. The table suggests that compared to an omniscient selector, *GlOSS* does a reasonable job of selecting relevant databases, on average finding over seventy percent of the documents that could be found by examining an equal number of databases under ideal circumstances, with gradual improvement as the number of databases examined increases. The large standard deviations arise because although *GlOSS* performs very well for the majority of queries, there remains a stubborn minority for which performance is very poor. Nevertheless, using *GlOSS* gives a dramatic improvement over randomly selecting databases to search, for a fraction of the storage cost of a full-text index.

We felt these initial results were promising enough to pursue the use of *GlOSS*'s representation for summary information. A more rigorous investigation is in progress. Ideally, we would like to use a real set of test databases instead of one constructed by partitioning, and a matching set of queries submitted against these same databases, including boolean disjunctions as well as conjunctions. We will try to characterize those queries for which *GlOSS* performs poorly, and to study the impact of the number of query terms on effectiveness. Other metrics will be included. For example, a metric that revealed whether the matching documents were scattered thinly across many databases or concentrated in a few large clumps would allow us to measure the corresponding impact on effectiveness. Effectiveness can also be measured using information retrieval metrics [7]. In this case, *GlOSS* would be measured in terms of its effectiveness in retrieving relevant *documents*, irrespective of the document location in one database or another [19].

4 Alternative Data Structures for *GlOSS* Summaries

The choice of a good data structure to store the *GlOSS* summaries depends on the type and frequency of operations at the *GlOSS* servers. A *GlOSS* server needs to support two types of operations efficiently: query processing and summary updates. When a query arrives, *GlOSS* has to access the complete set of document frequencies associated with each query keyword. When new or updated summaries arrive, *GlOSS* has to update its data structure, operating on the frequencies associated with a single database. Efficient access by database might also be needed if different brokers exchange database summaries to develop "expertise" [34], or if we allow users to do *relevance feedback* [33] and ask for databases "similar" to some given database. The two types of operations pose conflicting requirements on the *GlOSS* data structure: to process queries, *GlOSS* needs fast access to the table by word, whereas to handle frequency updates, *GlOSS* needs fast access to the table by database.

Thus, our problem requires efficient access to multi-dimensional data. For multi-dimensional data structures, queries and updates are generally expressed as selections on the possible data values of each dimension. Selections come in the form of constants, ranges of values, or a selection of all values along a dimension. A *point query* selects constants across all dimensions. For example, retrieving the *GlOSS* summary of the word *information* in db_1 from Table 1 is a point query. A *region query* selects ranges of values across all dimensions. Retrieving summaries for all words between *data* and *base* from databases 2 through 5 is a region query. *GlOSS* demands efficient *partial match* queries and updates: one dimension has a constant selected and the other dimension selects all values across the dimension. Partial match queries occur because we need to access the *GlOSS* records by word and by database. A workload constructed entirely of partial match queries creates unique demands on the data structure used to implement *GlOSS*.

Ideally we would like to simultaneously minimize the access cost in both dimensions. In general, however, the costs of word and database access trade off. Consequently, one must consider the relative frequencies of these operations, and try to find a policy that minimizes overall cost. Unfortunately, the relative frequencies of word and database access are difficult to estimate. They depend on other parameters, such as the number of databases covered by *GlOSS*, the intensity of query traffic, the actual frequency of summary updates, etc.

Just to illustrate the tradeoffs, let us assume that query processing is the most frequent operation, and that a *GlOSS* server receives 200,000 query requests per day. Likewise, let us assume that we update each database summary once a day. Given this scenario, and if *GlOSS* covers 500 databases, the ratio of accesses by word to accesses by database would be about 400:1, and our data structure might therefore favor the performance of accesses by word over that by database in the same proportion. However, if the server received 350,000 queries a day, or covered a different number of databases, or received updates more frequently, a vastly different ratio could occur. Therefore, *GlOSS* needs a data structure that can be tuned to adapt to the actual conditions observed in practice.

A simple data organization for *GlOSS* is to cluster the records according to their associated word, and to build a tree based directory on the words (e.g., a sparse B^+ tree), to provide efficient access by word [16], thus yielding fast query processing. To implement *GlOSS* using this approach, we could adapt any of the techniques for building inverted files for documents (e.g.,[8], [47], [40], [6]). However, this approach does not support fast access by database, for updating summaries or exchanging them with other brokers. To access all the words for a database, the entire directory tree must be searched.

Organizations for "spatial" data provide a variety of techniques that we can apply for *GlOSS*. In particular, we are interested in techniques that efficiently support partial match queries. Ap-

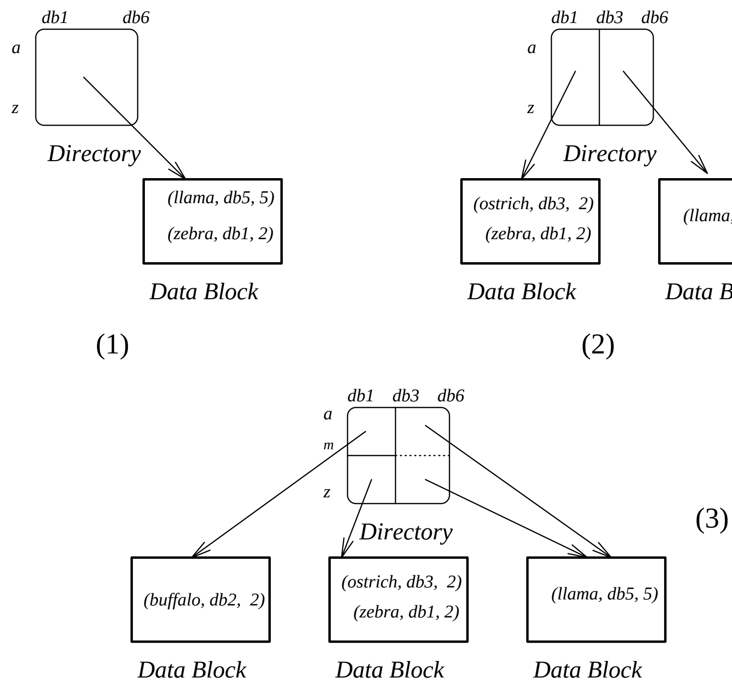

Figure 1: The successive configurations of a grid file during record insertion.

proaches that index multiple dimensions using a tree-based directory, including quad trees, *k-d* trees, *K-D-B* trees [42], R trees [18], R^+ trees [36], and BV trees [14], are not well suited for this type of access. To answer a partial-match query, typically a significant portion of the directory tree must be searched. A similar problem arises with techniques like the ones based on the "z order" [31]. In contrast, the directory structure of grid files [28] and the addressing scheme for partitioned or multi-attribute hashing [24] make them well suited for answering partial-match queries.

5 Using Grid Files for *GlOSS*

In this section we describe how grid files [28] can be used to store the *GlOSS* summaries, and describe a series of experiments that explore their performance. We show how to tune the grid file to favor access to the summary information by word or by database.

5.1 Grid File Basics

A grid file consists of data blocks, stored on disk and containing the actual data records, and a directory that maps multi-dimensional keys to data blocks. For *GlOSS*, the (two-dimensional) keys are (word, database identifier) pairs. Initially there is only one data block, and the directory consists of a single entry pointing to the only data block. Records are inserted in this data block until it becomes full and has to be split into two blocks. The grid file directory changes to reflect the splitting of the data block.

Figure 1 shows a grid file where the data blocks have capacity for two records. In (1), we have inserted two records into the grid file: (*llama*, db_5, *5*) and (*zebra*, db_1, *2*). There is only one data block (filled to capacity) containing the two records, and only one directory entry pointing to the only data block.

To insert record (*ostrich*, db_3, *2*), we locate the data block where the record belongs by first reading the directory entry corresponding to word *ostrich* and database db_3. Since the data block is full, we have to split it. We can split the data block between different databases, or between different words. In (2), we split the data block between databases: all records with databases in the (db_1, db_3) range go to one block, and all records with databases in the (db_4, db_6) range go to the other block. We also split the grid file directory to contain two entries, one pointing to each of the data blocks.

To insert record (*buffalo*, db_2, *2*), we first locate the data block where the record belongs: by looking at the directory, we find the pointer associated with range (db_1, db_3) and (a, z), and the corresponding data block. This data block already has two records in it, (*ostrich*, db_3, *2*) and (*zebra*, db_1, *2*), so the insertion of the new tuple causes the data block to overflow. In (3), we split the data block between words, and we reflect this splitting in the directory by creating a new row in it. The first row of the directory corresponds to word range (a, m), and the second to word range (n, z). Thus, the overflowed data block is split into one block with record (*buffalo*, db_2, *2*), and another block with records (*ostrich*, db_3, *2*) and (*zebra*, db_1, *2*). Note that both directory entries corresponding to database range (db_4, db_6) point to the same data block, which has not overflowed and thus does not need to be split yet. These two directory entries form a *region*. Regions may contain any number of directory entries, but are always convex in our grid files. We will refer to a division between directory entries in a region as a *partition* of the region. The region in the example directory contains a single partition. We define *directory utilization* as the ratio of directory regions to directory entries. In this example, the directory utilization is $\frac{3}{4}$ or 75%.

To locate the portion of the directory that corresponds to the record we are looking for, we keep one *scale* per dimension of the grid file. These scales are one-dimensional arrays that indicate what partitions have taken place in each dimension. For example, the *word scale* for the grid file configuration in (3) is (a, m, z), and the corresponding *database scale* is (db_1, db_3, db_6).

Consider for a moment the behavior of grid files for highly skewed data. For example, suppose the sequence of records (a, db_1, 1), (a, db_2, 1), (b, db_1, 1), (b, db_2, 1), (c, db_1, 1), (c, db_2, 1), etc. is inserted into the grid file of Figure 1 (3) and we continue to split between words. The resulting directory would exhibit very low utilization, since, for example, all the directory entries on the database dimension for the database db_5 would point to the same data block. In our application, the data is highly skewed and we attack the problem of low directory utilization by adjusting the way data blocks are split to account for the skew of the data.

Figure 2 shows the successive configurations for a different grid file for the same input as Figure 1. In this case, the grid file directory has been prepartitioned along one dimension. Prepartitioning the grid file directory has resulted in a larger directory for the same input. The directory utilization in this example is $\frac{3}{6}$ or 50%. We defer further discussion of the effect of prepartioning until Section 5.5.

5.2 Splitting Blocks

The rule that is used to decide how to split a data block is called the *splitting policy*. The splitting policy can be used to adjust the overall cost of using a grid file to store our summary information. Our goal is to find and evaluate splitting policies that are easily parameterized to support an observed ratio between the frequency of word and database accesses. We describe two extreme splitting policies that characterize the endpoints of the spectrum of splitting behavior, and then introduce three additional parameterized policies that can be adjusted to minimize overall cost.

To insert a record into the *GlOSS* grid file, we first find the block where the record belongs,

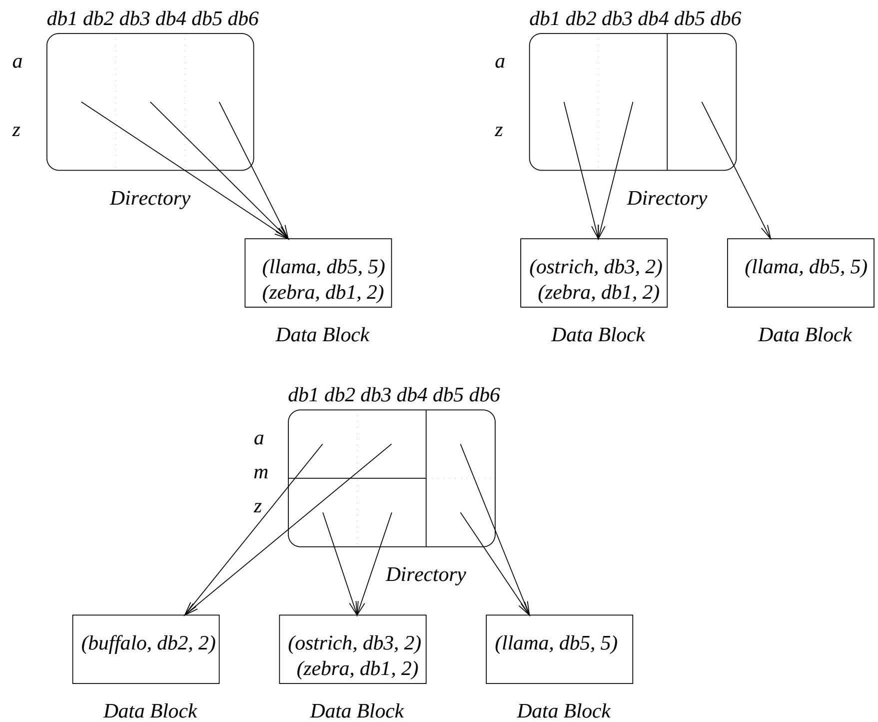

Figure 2: The successive configurations of a grid file during record insertion for a prepartitioned grid file.

using the grid file directory. If the record fits in this block, then we insert it. [2] Otherwise, the block must be split, either by dividing it between two words or by dividing it between two databases. Splitting between databases tends to benefit access by database, whereas splitting between words tends to benefit access by word. This choice of *splitting dimension* is therefore the basic tool for controlling relative access costs.

To limit the growth of the grid file directory, however, we always look for ways to split the block that take advantage of pre-existing partitions in the directory. [3] If more than one entry in the grid file directory maps to the overflowed block, then the collection of directory entries pointing to the overflowed block define a region. This region of the directory contains at least one partition (either between words or between databases). If the data corresponding to the entries on either side of the partition form non-empty blocks, then we can use one such pre-existing partition to split the block without introducing new entries into the directory. That is, the partition becomes a division between two new, smaller, regions. For example, the insertion of the records ($panther$, db_5, 2) and ($penguin$, db_5, 3) into Figure 1 (3) causes the rightmost data block to overflow and a new block to be created without changing the size of the directory. The single partition becomes two regions for a total of four regions in the directory. If more than one such partition exists, we favor those between databases over those between words. If multiple partitions exist in a single dimension, we choose the one that splits the block most nearly in half. (See Section 5.4 for a variation of this

[2] We can compress the contents of each block of the grid file by applying methods used for storing sparse matrices efficiently [32], or by using the methods in [47] for compressing inverted files, for example. Any of these methods will effectively increase the capacity of the disk blocks in terms of the number of records that they can hold.

[3] Several alternative organizations for the grid file directory control its growth and make it proportional to the data size. These alternative organizations include the *region-representation* directory and the BR^2 directory [3]. The *2-level* directory organization [22] shows how to implement the directory on disk. We have not yet explored how these techniques would work in our environment.

```
1.  Compute region and block for record
2.  If    Record fits in block
3.        Insert record
4.  Else
5.        If      Usable partitions in database scale
6.                Divide region in half on database scale
7.        Else If Usable partitions in word scale
8.                Divide region in half on word scale
9.        Else
10.               Split directory
11.               Divide region on chosen scale
12.       Insert record
```

Figure 3: Algorithm for inserting a record in a grid file for *GlOSS*.

Policy	Splitting dimension
`DB-always`	Database
`Word-always`	Word
`Bounded`	If *DB-splits* $<$ *bound* then Database else Word
`Probabilistic`	If *Random()* $<$ *prob-bound* then Database else Word
`Prepartition`	Like `Word-always`, after prepartitioning on Database

Table 3: Different policies for choosing the splitting dimension.

policy that reduces the amount of unused space in blocks.)

To be precise, Figure 3 shows the basic algorithm for inserting a record into the grid file. Line 1 computes the region and block where the record should be inserted according to the database and word scales for the grid file directory. Line 2 attempts to insert the record. If there is no overflow, the insertion succeeds. Otherwise, there is overflow in the block. Line 5 checks the region in which the record is being inserted for a partition in the database scale. If there is a partition, the region is divided in half along a partition line, and the records in the block of the region are redistributed between the old and new block. The new block is assigned to the new region. This process eliminates a partition of the region by creating a new region. Lines 7-8 do the same for the word scale. If there are no qualifying partitions (line 10), we need to create one by introducing a new row (or column) in the directory.

Table 3 describes several policies for choosing a splitting dimension. The `DB-always` policy always attempts to split the block between databases, thus favoring access by database over access by word. Conversely, the `Word-always` policy always attempts to split between words, thus favoring access by word over access by database. In between these two extremes lies a spectrum of other possibilities. The `Bounded` policy allows the database scale of the grid file directory to be split up to *bound* times, and then resorts to splitting between words. Thus, it allows some splits between databases (which favor access by database), while putting an upper bound on the number of block reads that might be needed to access all the records for a word. If *bound* is set to infinity, then `Bounded` behaves as `DB-always`, whereas if *bound* is set to zero, then `Bounded` behaves as

`Word-always`. The `Probabilistic` policy splits between databases with probability *prob-bound*. Unlike the `Bounded` policy, which favors splitting between databases initially, this policy allows the choice of splitting dimension to be made independently at each split. The `Prepartition` policy works like `Word-always`, except that the database scale of the directory is prepartitioned into m regions before any databases are inserted, to see if "seeding" the database scale with evenly-spaced partitions improves performance. The size of each region is $\lfloor \frac{m}{db} \rfloor$, where db is the number of available databases.

Note that once a scale has been chosen, it may not be possible to split the block on that scale. For instance, we may choose to split a block on the database scale, but the scale may have only a single value associated with that block (and consequently, every record in the block has the same database value). In this case, we automatically split on the other scale.

5.3 Metrics for Evaluation

To evaluate the policies of Table 3, we implemented a simulation of the grid file in C++ on an IBM RISC/6000 workstation and ran experiments using 200 of the 500 patent databases described in Section 3 (around 1.3 gigabytes of data). The resulting grid file had 200 columns (one for each of the 200 databases) and 402,044 rows (one for each distinct word appearing in the patent records). The file contained 2,999,676 total records. At four bytes per entry, we assumed that each disk block could hold 512 records.

Our evaluation of the various policies is based on the following metrics (cf. Section 5.1):

DB Splits Number of splits that occurred in the database scale.

Word Splits Number of splits that occurred in the word scale.

Total Blocks The total number of blocks in the grid file (excluding the scales and the directory).

Block-fill factor The ratio of used block space to total block space. This measure indicates how effectively data is packed into blocks.

Directory Size The number of entries in the database scale of the grid file directory times number of entries in the word scale. This measure indicates the overhead cost of the grid file directory. About four bytes would be needed for each directory entry in an actual implementation.

Average Word (or Database) Access Cost The number of blocks accessed in reading all the records for a single word (or database), i.e., an entire row (or column) of the grid file, averaged over all words (or databases) on the corresponding scale.

Expansion Factor for Words (or Databases) The ratio between the number of blocks accessed in reading all the records for a single word or database and the minimum number of blocks that would be required to store that many records, averaged over all words or databases on the corresponding scale. This metric compares the access cost using the grid file to the best possible access cost that could be achieved. Note that since we assume that 512 records can be stored in a block, and there are only 200 databases, all the records for a single word can always fit in one block. Thus the minimum number of blocks required for each word is one, and the expansion factor for words is always equal to the average word access cost.

Average Trace Word Access Cost and **Expansion Factor for Trace Words** Similar to the word scale metrics, but averaged over the words occurring in a representative set of patent queries, instead of over all words. For this measurement, we used 1767 queries issued by real users in November 1994, against a full-text patent database accessible at `http://-town.hall.org/patent/patent.html`. These queries contain 2828 words that appear in at least one of our 200 databases (counting repetitions). This represents less than one percent of the total vocabulary, but query words from the trace (trace words) occur on average in 99.96 databases compared to the average of 7.46 for all words. Thus trace words occur with relatively high frequency in the databases.

Weighted (Trace) Average Cost This metric gives the overall cost of using the grid file, given an observed ratio between word and database accesses. It is calculated by multiplying the word-to-database access ratio by the average (trace) word access cost, and adding the average database access cost. For example, if the ratio of word to database accesses is observed to be 100:1, the weighted average cost is (100 * average word access cost) + average database access cost.

Although the choice of splitting policy is the major factor in determining the behavior of the grid file, performance is also sensitive to a number of other, more subtle, variations in how the *GlOSS* summaries are mapped onto the grid file. We therefore discuss these variants before moving on to the main body of our results.

5.4 Mapping *GlOSS* to a Grid File

To insert the *GlOSS* summary data for a database into a grid file, one must first define a mapping from each word to an integer that corresponds to a row in the grid file. We explored two alternatives for this mapping, `alpha` and `freq`. In the `alpha` mapping, all the words in all the databases are gathered into a single alphabetically ordered list, and then assigned sequential integer identifiers. In the `freq` mapping, the same set of words is ordered by frequency, instead of alphabetically, where the frequency for a word is the sum of the frequencies for that word across all summaries. [4]

This difference in mapping has two effects. First, although the vast majority of rows have exactly one record, the `freq` map clusters those rows having multiple records in the upper part of the grid file, and the top rows of the grid file contain a record for every database. In the `alpha` map, the rows with multiple records are spread throughout the grid file. (By contrast, the distribution of records across the columns of the grid file is fairly uniform.)

The second effect is due to the fact that, as an artifact of its construction, the summary for each database is ordered alphabetically. For the `alpha` mapping, therefore, (word id, frequency) pairs are inserted in increasing, but non-sequential, word-identifier order. For example, db_1 might insert records (1,28) (2,15) (4,11) and db_2 might insert records (1,25) (3,20) (4,11). In each case, the word identifiers are increasing, but they are non-sequential. By contrast, with the `freq` mapping, (word id, frequency) pairs are inserted in essentially random order, since the words are ordered alphabetically but the identifiers are ordered by frequency.

Similar considerations pertain to the order in which databases are inserted into the grid file. We considered sequential ordering (`seq`) and random ordering (`random`). In the `seq` case, database 1 is inserted with database identifier 1, database 2 with database identifier 2, etc. In the `random` ordering, the mapping is permuted randomly. The `seq` ordering corresponds to statically loading the grid file from a collection of summaries. The `random` ordering corresponds to the dynamic addition and deletion of summaries as information is updated or exchanged among brokers.

A consequence of the `seq` ordering is that insertion of data into the grid file is very deterministic. In particular, we noticed that our default means of choosing a partition in the case of overflow was

[4] In practice, one could approximate the `freq` mapping by using a predefined mapping table for relatively common words, and assigning identifiers in order for the remaining (infrequent) words.

Mapping		Policy	Splits		Average Cost		Total	Block-Fill	Directory
Word	Database		Word	DB	Word	DB	Blocks	Factor	Size
alpha	seq	middle	119	117	104.92	109.69	10859	.54	13923
alpha	seq	right	138	115	83.55	115.89	8584	.68	15870
alpha	random	middle	204	85	61.03	159.40	9258	.63	17340
alpha	random	right	187	120	87.34	133.44	10586	.55	22440
freq	seq	middle	118	172	93.04	108.66	12601	.46	20296
freq	seq	right	118	167	58.41	108.36	8750	.67	19706
freq	random	middle	194	127	69.14	128.30	9737	.60	24638
freq	random	right	167	142	83.29	118.77	10398	.56	23714

Table 4: The `DB-always` policy for the different mapping and splitting options.

a bad one. Since databases are inserted left to right, the left-hand member of a pair of split blocks is never revisited: subsequent insertions will always insert into the right-hand block. Thus, when the database scale is split (in line 11 of the algorithm in Figure 3), it would be advantageous to choose the rightmost value in the block as the value to split on. Furthermore, if given a choice of pre-existing partitions to use in splitting a block, it would be advantageous to choose the rightmost partition for splitting (in line 6 of the algorithm). To examine this effect, we parameterized the algorithm in Figure 3 to choose either the rightmost value or partition (`right`) or the middle-most value or partition (`middle`), as per the original algorithm.

We ran experiments for the eight combinations of mapping and splitting options above for the `DB-always`, `Word-always`, `Bounded`, and `Probabilistic` policies. Table 4 shows the results for the `DB-always` policy, but the conclusions we draw here apply to the other policies as well. Note that the combination of options chosen can have a significant effect on performance. The average word access cost for the worst combination of options is 1.8 times the average word access cost for the best combination. For average database access cost, this factor is about 1.5. Block-fill factor varies from a worst case of 0.46 to a best case of 0.68.

The table shows that the combination of frequency ordering for assignment of word identifiers, sequential insertion of databases, and the `right` option for block splitting achieves the lowest access costs, both by word and by database, and has a block-fill factor only slightly poorer than the best observed. Therefore, we used this combination for our subsequent experiments. The base parameters for these experiments are summarized in Table 5.

5.5 Comparison of Splitting Policies

We begin our comparison of the splitting policies by examining the basic behavior of each of the five policies. Tables 6 and 7 provide performance measurements for each policy; for the parameterized policies (`Probabilistic`, `Prepartition` and `Bounded`) we present data for a single representative parameter value, and defer discussion of other parameter values to later in this section.

We start with the `Word-always` policy, since its behavior is very regular. At the start of the experiment, there is a single empty block. As databases are inserted, the block overflows, and the word scale is split at a point that balances the resulting blocks. By the time all 200 databases have been inserted, the word scale has been split 8878 times. In the resulting grid file, each data block therefore contains complete frequency information for some number of words, i.e., multiple rows of the grid file. The number of words in a data block depends on the number of databases in which the corresponding words appear. As expected, the average word access cost is one block read. Clearly,

Parameter	Value
Databases (columns)	200
Words (rows)	402,044
Records	2,999,676
Records per block	512
Database Insertion	`seq`
Word Insertion	`freq`
Block division	`right`

Table 5: Parameter values for the base set of experiments. See Section 5.4 for a description of the parameters.

Policy	Splits		Total	Block-Fill	Directory
	Word	DB	Blocks	Factor	Size
`Word-always`	8878	1	8878	.66	8878
`DB-always`	118	167	8750	.67	19706
`Probabilistic` (0.5)	202	101	8887	.66	20402
`Prepartition` (10)	2361	10	8779	.67	23610
`Bounded` (10)	8252	10	8694	.67	82520

Table 6: Performance measurements for the base experiment for the five policies introduced in Section 5.2.

this policy is the most favorable one possible for access by word. To access all the records for a database, however, every block must be read. The average database access cost therefore equals the total number of blocks in the file. This policy minimizes the size of the grid file directory, since it reduces the directory to a one-dimensional vector of pointers to the data blocks.

Next, consider the `DB-always` policy. Our measurements show that the database scale was split 167 times. However, the size of the grid file far exceeds the capacity of 167 blocks, so splitting must occur between words as well (cf. Section 5.2). Such splits will take advantage of existing partitions of the word scale, if they exist, otherwise the word scale of the directory will be split. Such splitting of the word scale occurred 118 times during our experiment, leading to an average

Policy	Avg. Word Cost (Dev.)	Avg. DB Cost (Dev.)	Expansion Factor for DB (Dev.)	Avg. Trace Word Cost (Dev.)
`Word-always`	1.00 (0.00)	8878.00 (0.00)	710.09 (1152.57)	1.00 (0.00)
`DB-always`	58.41 (18.32)	108.36 (11.79)	8.84 (15.23)	104.70 (27.75)
`Probabilistic` (0.5)	53.08 (12.19)	153.63 (38.79)	13.13 (25.63)	70.51 (13.09)
`Prepartition` (10)	10.00 (0.00)	2180.81 (238.88)	167.09 (248.87)	10.00 (0.00)
`Bounded` (10)	7.09 (0.74)	7883.91 (1695.71)	640.99 (1076.49)	8.35 (1.24)

Table 7: Performance measurements for the base experiment for the five policies introduced in Section 5.2.

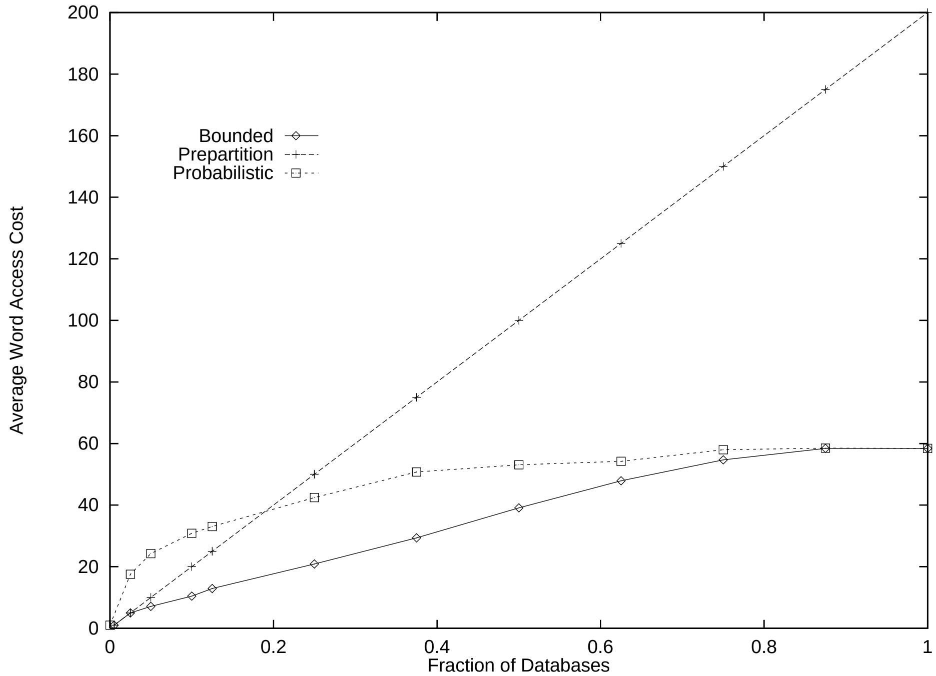

Figure 4: The average word access cost as the bound changes.

database access cost of 108.36 for this policy. At 8.84 times the minimum number of blocks that must be read, this is the best average database access cost of the policies we measured. However, 58.41 is the worst average word access cost, and for the frequently occurring words of the trace queries, the cost is even higher.

As a point of comparison for the two extremes of the `DB-always` and `Word-always` policies, we measured the `Probabilistic` policy, parameterized so that the word and database scales would be chosen with equal probability. If the distribution of data were uniform between the two scales, this policy would on average split each scale the same number of times. As the table shows, however, for our data this policy behaves very much like the `DB-always` policy. For both of these policies, the skewed nature of the data (i.e., the vastly larger number of distinct values on the word scale) makes many attempts to split on the database scale unsuccessful. In effect, the database scale quickly becomes "saturated" and large numbers of splits must occur in the word scale. For this parameter value, the `Probabilistic` policy gives poorer average database access cost and slightly better average word access cost, when compared to the `DB-always` policy. The difference is more pronounced for the average trace word access cost. Block-fill factor varies very little.

Figures 4 and 5 show how the three tunable policies, `Bounded`, `Prepartition` and `Probabilistic`, behave as the tuning parameter varies. In order to graph these results on a common set of axes, we express the parameter as a fraction of the total number of databases. Thus, for our study of 200 databases, an X-axis value of .05 in these figures represents parameter values of 10, 10, and .05 for the `Bounded`, `Prepartition` and `Probabilistic` policies, respectively.

Figure 6 reveals a hidden cost of the `Bounded` policy: an up-to-tenfold inflation in the size of the grid-file directory for parameter values midway between the extremes. Consider the parameter value 0.4 in this figure. The `Bounded` policy forces splits between databases to occur for the first 40% of the databases. That is, as early as possible in the insertion process, whereas the other two policies distribute them evenly. Under the `Bounded` policy, therefore, relatively few splits between words occur early in the insertion process (because the regions being split are typically only one

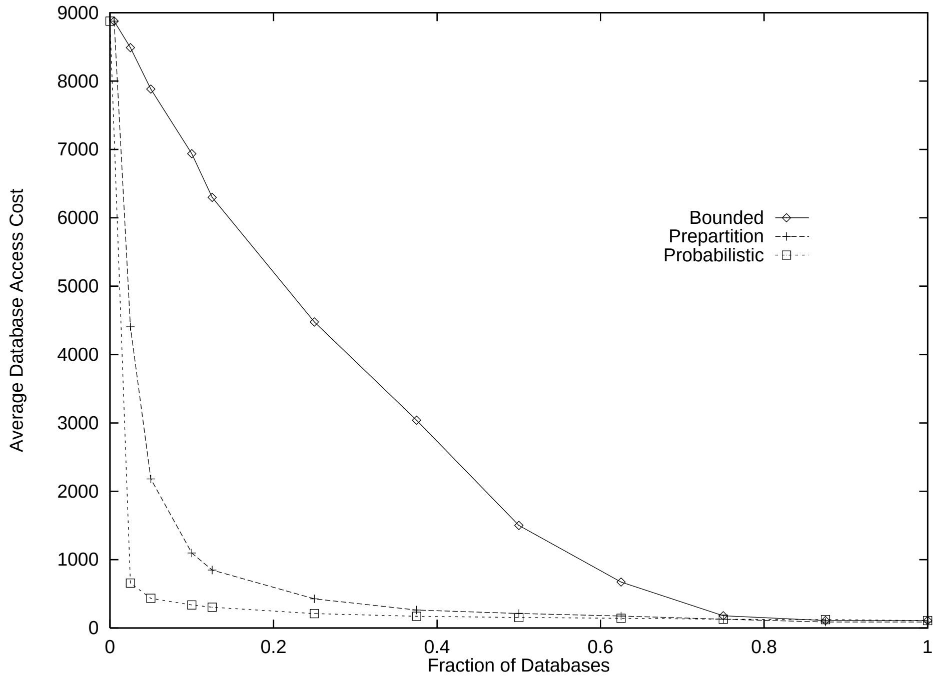

Figure 5: The average database access cost as the bound changes.

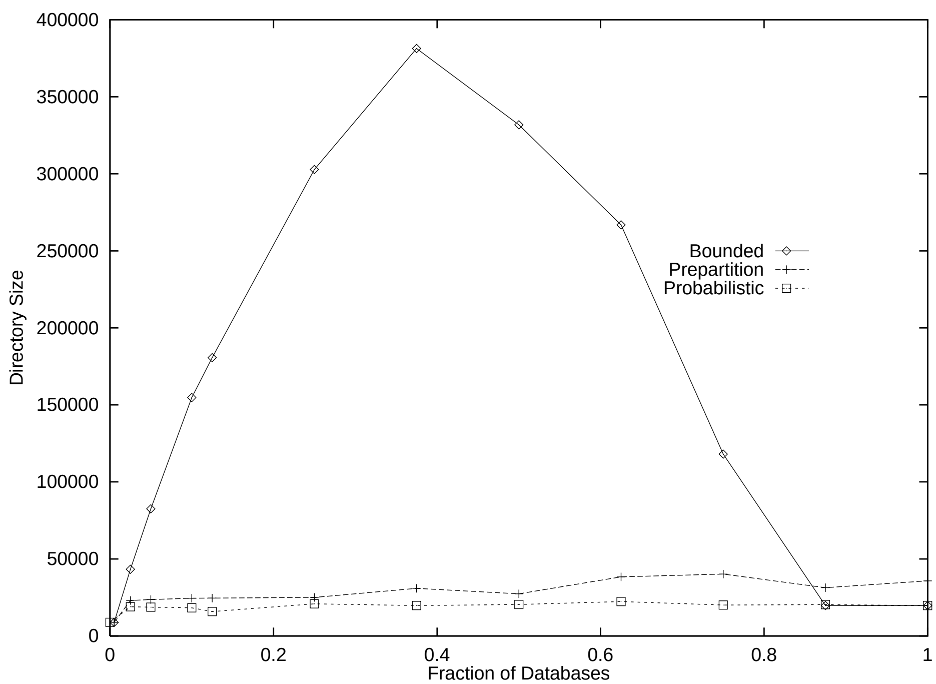

Figure 6: Directory size as a function of policy parameter.

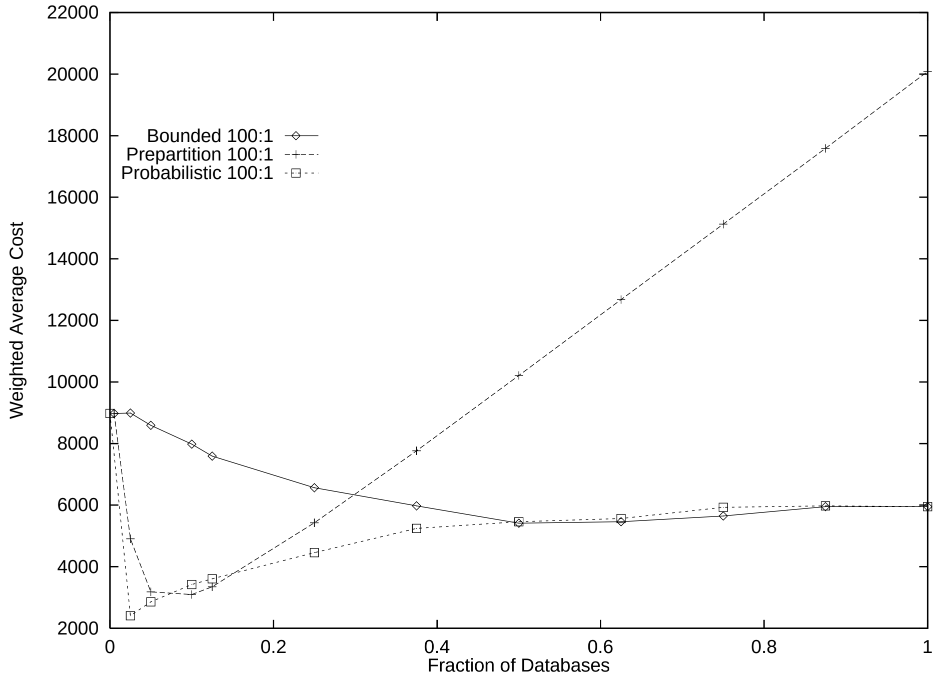

Figure 7: Weighted average cost for a word-to-database access ratio of 100:1.

database wide) but, once the bound has been reached, many splits between words are required to subdivide the remaining portion of the grid file. Each of these latter splits introduces a number of additional directory entries equal to the bound value. With a low bound value, there are few splits between databases, with a high bound value, there are many splits between databases, but few splits between words to introduction the additional directories entries. With the bound near the middle, these two effects complement each other to produce a huge directory. With the other policies, the number of splits between words for each group of databases is fairly constant across the width of the grid file, and the total number of splits between words (and hence the directory size) is much smaller.

5.6 Weighted Average Costs

Table 7 presents no clear winner in terms of an overall policy choice, because the performance of a policy can only be reduced to a single number once the ratio of accesses by word to accesses by database has been determined. Only then can an appropriately weighted overall access cost be calculated. For a word-to-database access ratio of 100:1, Figure 7 shows the weighted average cost for each of the policies, across the entire parameter range. [5] The lowest point on this set of curves represents the best choice of policy and parameter for this access ratio, and corresponds to the `Probabilistic` policy with a parameter of about .025.

The best selections for various ratios are given in Tables 8 and 9, for the weighted average cost and weighted trace average cost, respectively. When access by word predominates, `Word-always` gives the best performance. When access by database is as common as access by word (or more common), `DB-always` is the preferred policy. In between, the `Probabilistic` policy with an appropriate parameter dominates the other choices.

[5] The `Word-always` and `DB-always` policies are represented by the points for `Probabilistic`(0) and `Probabilistic`(1), respectively.

Ratio	Policy	Weighted Avg. Cost	Block-Fill Factor
1:1	`DB-always`	167	.67
10:1	`Probabilistic`(.125)	634	.67
100:1	`Probabilistic`(.025)	2409	.66
1000:1	`Word-always`	9878	.66

Table 8: The policy choices that minimize the weighted average cost, for different word-to-database access ratios.

Ratio	Policy	Weighted Trace Avg. Cost	Block-Fill Factor
1:1	`DB-always`	213	.67
10:1	`Probabilistic`(.125)	659	.67
100:1	`Probabilistic`(.025)	2506	.66
1000:1	`Word-always`	9878	.66

Table 9: The policy choices that minimize the weighted trace average cost, for different word-to-database access ratios.

5.7 Bounded Access Costs

If the databases summarized by *GlOSS* grow gradually over time, the weighted access costs for the grid file must grow as well. Using the recommended policies of Tables 8 and 9, this increasing cost will be distributed between word and database access costs so as to minimize the weighted average cost. The response time for a given query, however, depends only on the word access costs for the terms it contains, and will increase without bound as the grid file grows. If such response time growth is unacceptable, the `Bounded` and `Prepartition` policies can be used to put an upper limit on word access cost, in which case query cost will depend only on the number of terms in the query.

The upper limit on word access cost for these policies is determined by the parameter value. With the `Prepartition` policy, the word access cost is exactly the parameter value: e.g., the cost is 10 block accesses for any word for `Prepartition`(10). The `Bounded`(10) policy gives the same upper limit, but the average cost is lower (about 7) because for many words, the cost does not reach the bound. However, Tables 6 and 7 in Section 5.5 show the penalty for the improved average word access cost: about a fourfold increase in both directory size and database average access cost. The corresponding tradeoffs for other values of the parameter can be deduced from Figures 4, 5 and 6 in Section 5.5.

5.8 Other Experiments

We did a number of other experiments to complete our evaluation of grid files as a storage method for *GlOSS* summaries. In particular, since we must be able to maintain (update) the summaries efficiently, we tested each of the policies under simulated updates. We also ran our experiments with a smaller block size to see how that affected our results. Details can be found in [41]. The results were generally acceptable and did not serve to differentiate the various policies, hence they are not repeated here.

6 Using Partitioned Hashing for *GlOSS*

In this section we analyze partitioned (or multi-attribute) hashing [24] as an alternative technique for *GlOSS* to access its records efficiently both by word and by database. We first describe how partitioned hashing handles the *GlOSS* summaries, and then we show experimental results on its performance using the data of Section 5.

6.1 Partitioned-Hashing Basics

With partitioned hashing, the *GlOSS* records are stored in a hash table consisting of $B = 2^b$ buckets. Each bucket is identified by a string of b bits. b_w of these b bits are associated with the word attribute of the records, and the $b_{db} = b - b_w$ remaining bits with the database attribute of the records. Hash functions h_w and h_{db} map words and databases into strings of b_w and b_{db} bits, respectively. A record (w, db, f), with word w and database db, is stored in the bucket with address $h_w(w)h_{db}(db)$, formed by the concatenation of the $h_w(w)$ and $h_{db}(db)$ bit strings. To access all the records with word w, we search all the buckets whose address starts with $h_w(w)$. To access all the records with database db, we search all the buckets whose address ends with $h_{db}(db)$ [6].

The h_w hash function maps words into integers between 0 and $2^{b_w} - 1$. Given a word $w = a_n \ldots a_0$, h_w does this mapping by first translating word w into integer $i_w = \sum_{i=0}^{n} \mathit{lettervalue}(a_i) \times 36^i$ [46], and then taking $\lfloor (i_w A \bmod 1) 2^{b_w} \rfloor$, where $A = 0.6180339887$[24]. Similarly, the h_{db} hash function maps database numbers into integers between 0 and $2^{b_{db}} - 1$. Given a database number i_{db}, h_{db} maps it into integer $\lfloor (i_{db} A \bmod 1) 2^{b_{db}} \rfloor$. We initially assign one disk block per hash-table bucket. If a bucket overflows, we assign more disk blocks to it.

Given a fixed value for b, we vary the values for b_w and b_{db}. By letting b_w be greater than b_{db}, we favor access to the *GlOSS* records by word, since there will be fewer buckets associated with each word than with each database. In general we just consider configurations where b_w is not less than b_{db}, since the number of words is much higher than the number of databases, and in our model the records will be accessed more frequently by word than by database. In the following section, we analyze experimentally the impact of the b_w and b_{db} parameters on the performance of partitioned hashing for *GlOSS*.

6.2 Experimental Results

To analyze the performance of partitioned hashing for *GlOSS*, we ran experiments using the 2,999,676 records for the 200 databases of Section 5. For these experiments, we assumed that 512 records fit in one disk block, that each bucket should span one block on average, and that we want each bucket to be 70% full on average. Therefore, we should have around $B = \lceil \frac{2{,}999{,}676}{0.7 \times 512} \rceil = 8370$ buckets, and we can dedicate approximately $b = 13$ bits for the bucket addresses. (Section 7 shows results for other values of b.)

To access all the records for a word w we must access all of the $2^{b_{db}}$ buckets with address prefix $h_w(w)$. Accessing each of these buckets involves accessing one or more disk blocks, depending on whether the buckets have overflowed or not. Figure 8 shows the average word access cost as a function of b_w ($b = 13$ and $b_{db} = b - b_w$). As expected, the number of blocks per word decreases as b_w increases, since the number of buckets per word decreases. Conversely, Figure 9 shows that the average database access cost increases steeply as b_w increases. In the extreme case when $b_w = 13$ and $b_{db} = 0$, we need to access every block in every bucket of the hash table, resulting in an expansion factor for databases of around 773.28. [7] In contrast, when $b_w = 7$ and $b_{db} = 6$ we access,

[6] An improvement over this scheme is to apply the methodology of [12] and use Gray codes to achieve better performance of partial-match queries.

[7] Smarter bucket organizations can help alleviate this situation by sorting the records by database inside each bucket, for example. However, all buckets of the hash table would still have to be examined to get the records for a database.

Ratio	b_w	b_{db}	Weighted Avg. Cost	Block-Fill Factor
1:1	7	6	217	.64
10:1	8	5	587	.70
100:1	10	3	1845	.71
1000:1	11	2	6323	.69

Table 10: The choices for b_w and b_{db} that minimize the weighted average cost, for different word-to-database access ratios.

on average, around 11.24 times as many blocks for a database as we would need if the records were clustered by database.

Partitioned hashing does not distribute records uniformly across the different buckets. For example, all the records corresponding to database db belong in buckets with address suffix $h_{db}(db)$. Surprisingly, this characteristic of partitioned hashing does not lead to a poor block-fill factor: the average block-fill factor for $b = 13$ and the different values of b_w and b_{db} is mostly higher than 0.6, meaning that on average blocks were at least 60% full. These high values of block-fill factor are partly due to the fact that only the last block of each bucket can be partially empty: all of the other blocks of a bucket are completely full.

To measure the performance of partitioned hashing for access by word, we have so far computed the average value of various parameters over all the words in the combined vocabulary of the 200 databases. Figure 8 also shows a curve using the words in the query trace of Section 5. The average trace word access cost is very similar to the average word access cost. Two aspects of partitioned hashing and our experiments explain this behavior. Firstly, the number of blocks read for a word w does not depend on the number of databases associated with w: we access all the $2^{b_{db}}$ buckets with prefix $h_w(w)$. Consequently, we access a similar number of blocks for each word. (For example, when $b_w = 7$ and $b_{db} = 6$ the number of blocks we access per word ranges between 66 and 82.) Secondly, there are only 2^{b_w} possible different word access costs, because the hash function h_w maps the words into 2^{b_w} different values. Each trace word w will contribute a "random sample" $(h_w(w))$ of this set of 2^{b_w} possible costs. Furthermore, the number of words in the query trace (2828 word occurrences from a set of 1304 different words) is significant with respect to the number of different access costs, for the values of b that we used in our experiments. In summary, each hashed value $h_w(w)$ acts as a random sample of a limited set of different access costs, and we consider a high number of such samples.

To determine the best values for b_w and b_{db} for an observed word-to-database access ratio, we computed the weighted average cost for different access ratios, as in Section 5.6. Figure 10 shows the results for a 100:1 word-to-database access ratio (i.e., when accesses by word are 100 times as frequent as accesses by database). For this ratio, the best choice is $b_w = 10$ and $b_{db} = 3$, with a weighted average cost of around 1844.51. Table 10 summarizes the results for the different access ratios. Table 11 shows the corresponding results for the trace words. [8]

7 Comparing Grid Files to Partitioned Hashing for Storing Summaries

A comparison of Tables 8 and 9 with Tables 10 and 11 shows that with an ideal choice of parameters for either structure, partitioned hashing and the grid file are competitive data structures for storing

[8]References [1] and [25] study how to analytically derive the values of b_w and b_{db} that would minimize the number of *buckets* accessed for a given query distribution.

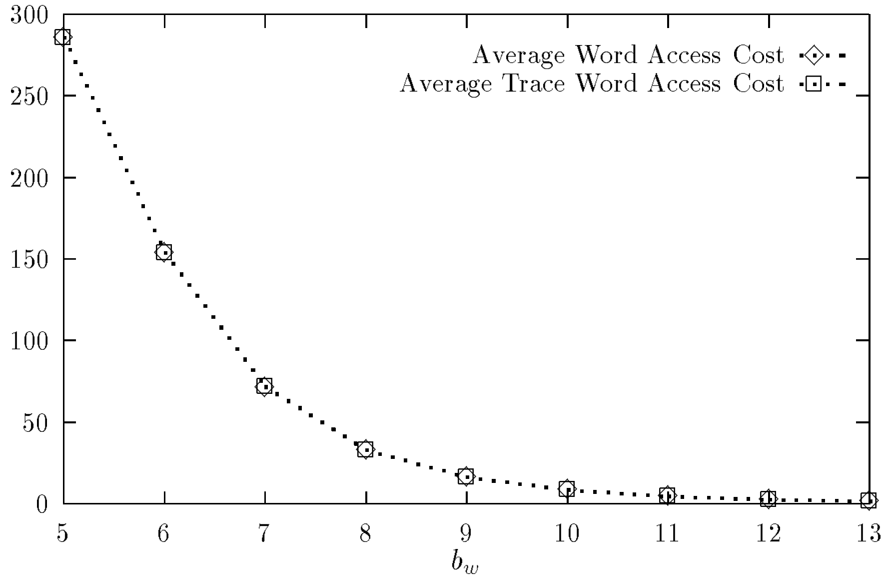

Figure 8: Average word access costs as a function of b_w.

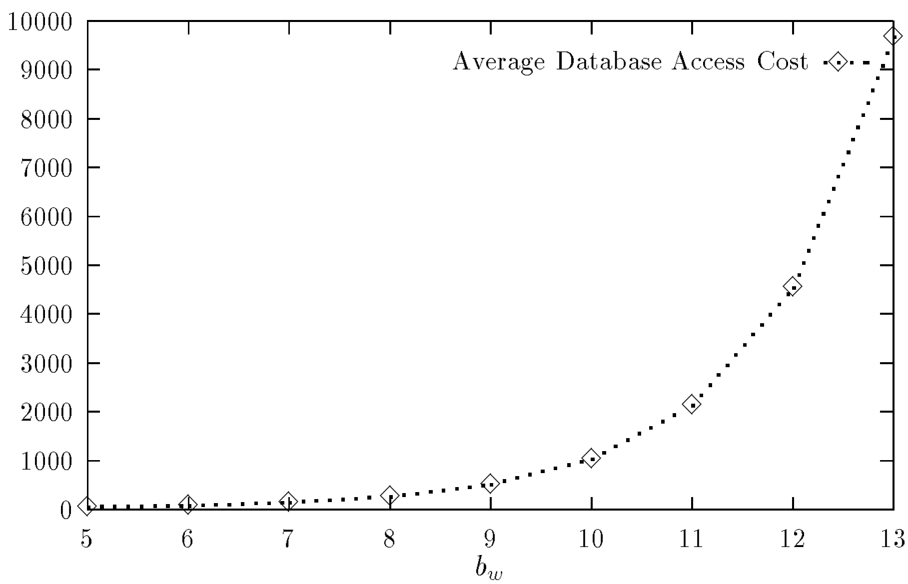

Figure 9: Average database access cost as a function of b_w.

Ratio	b_w	b_{db}	Weighted Trace Avg. Cost	Block-Fill Factor
1:1	7	6	218	.64
10:1	8	5	588	.70
100:1	10	3	1851	.71
1000:1	11	2	6456	.69

Table 11: The choices for b_w and b_{db} that minimize the weighted trace average cost, for different word-to-database access ratios.

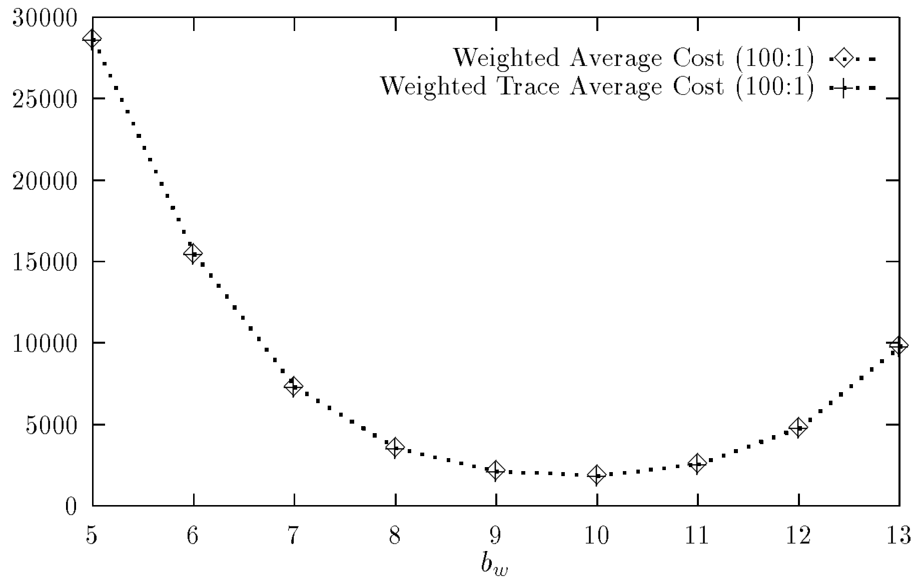

Figure 10: Weighted average cost for a word-to-database access ratio of 100:1, as a function of b_w.

GlOSS summaries. The grid file outperforms partitioned hashing only when word and database accesses are equally frequent. Additionally, the implementation of partitioned hashing is generally simpler than grid files. However, for a number of practical reasons, we believe that the grid file is better suited to this application.

Firstly, to get optimum performance from partitioned hashing, it is critical to choose the total number of buckets correctly. For instance, suppose that we overestimate the number of records of the *GlOSS* summaries and set the number of bits that identify each bucket to $b = 14$ instead of to $b = 13$ for the experiments of Section 6. Table 12 shows that, as expected, the average block-fill factor drops to about half the values for $b = 13$ (see Table 10), because there are twice as many buckets now for the same number of records. The best average access costs also deteriorate: for example, the weighted average cost for the 100:1 word-to-database access ratio grows to 2624 (from 1844.51 for the $b = 13$ case). Alternatively, if we underestimate the number of records of the *GlOSS* summaries and set $b = 12$, we obtain the results in Table 13. In this case, the average block-fill factor is higher than in the $b = 13$ case. However, all of the average access costs are significantly higher. For example, for the 100:1 word-to-database access ratio, the weighted average cost for $b = 12$ is 2623.05, whereas it is 1844.51 for $b = 13$. These experiments show that it is crucial for the performance of partitioned hashing to choose the right number of buckets in the hash table. Since we expect databases to grow over time, even an initially optimal choice will degrade as database size increases. By contrast, the grid file grows gracefully. Dynamic versions of multi-attribute hashing like the ones in [26] solve this problem at the expense of more complicated algorithms, resulting in techniques that are closely related to the grid files.

Secondly, with partitioned hashing, the tradeoff between word and database access cost is fixed for all time once a division of hash-value bits has been made. The only way to correct for an error is to rebuild the hash table. By contrast, the value of the probabilistic splitting parameter for the grid file can be dynamically tuned. Although changing the parameter may not be able to correct for unfortunate splits in the existing grid file, at least future splitting decisions will be improved.

Finally, partitioned hashing treats all words the same, regardless of how many or how few databases they occur in, and likewise treats all databases the same, regardless of the number of words they contain. By contrast, the cost of reading a row or column of the grid file tends to be

Ratio	b_w	b_{db}	Weighted Avg. Cost	Block-Fill Factor
1:1	7	7	255	.36
10:1	9	5	832	.36
100:1	10	4	2624	.36
1000:1	12	2	8096	.36

Table 12: The choices for b_w and b_{db} that minimize the weighted average cost, for different word-to-database access ratios and for $b = 14$.

Ratio	b_w	b_{db}	Weighted Avg. Cost	Block-Fill Factor
1:1	6	6	253	.74
10:1	8	4	832	.72
100:1	9	3	2623	.72
1000:1	11	1	7968	.73

Table 13: The choices for b_w and b_{db} that minimize the weighted average cost, for different word-to-database access ratios and for $b = 12$.

proportional to the number of records it contains.

8 Related Work

Many approaches to solving the text database discovery problem have been proposed. These fall into two groups: distributed browsing systems and query systems. In distributed browsing systems (e.g., [4], [27]), users follow pre-defined links between data items. While a wealth of information is accessible this way, links must be maintained by hand, and are therefore frequently out of date (or non-existent). Finding information can be frustrating to say the least.

To address this problem, an increasing number of systems allow users to query a collection of "meta-information" about available databases. The meta-information typically provides some summary of the contents of each database; thus, these systems fit our generic concept of a broker. Of course, different systems use different types of summaries, and their implementation varies substantially (e.g., [23], [10], [38], [2], [30], [37, 11], and [29, 35]).

Some systems provide manual mechanisms to specify meta-information (e.g., WAIS [23], Yahoo[9], and ALIWEB[10]) and attach human-generated text summaries to data sources. Given a query, these systems search for matching text summaries and return the attached data sources. Unfortunately, human-generated summaries are often out of date. In effect, as the database changes the summaries generally do not. In addition, English text summaries generally do not capture the information the user wants.

Other systems construct meta-information automatically. These systems are generally of two types: *document-based* systems generate meta-information from the documents at the data sources; *cooperative* systems provide a broker architecture that the data source administrator maps to the data. Two important issues that these systems have to address are retrieval effectiveness and scalability.

Document-based systems face a retrieval effectiveness problem, since each document is equal to every other document. Given two databases that are equally relevant to a query, the database with more documents will be more highly represented in the answer. We believe that *GlOSS*

[9]Yahoo is accessible at `http://www.yahoo.com`.

[10]ALIWEB is accessible at `http://web.nexor.co.uk/aliweb/doc/aliweb.html`.

representations of databases could improve retrieval effectiveness for this class of system. This application of *GlOSS* is an area of future research.

Document-based systems are faced with a scalability problem when the number of documents grows. The architecture of document-based systems generally comprises a centralized server (e.g., Lycos[11] and AltaVista[12]). In such a centralized architecture, a robot usually scans the collection of documents in a distributed way, fetching every document to the central server. One solution to the scalability problem is to index only document titles or, more generally, just a small fraction of each document (e.g., the World-Wide Web Worm[13]). This approach sacrifices important information about the contents of each database. We believe that the retrieval effectiveness and scalability of document-based systems would be dramatically improved if each data source generated a *GlOSS* summary instead of transmitting the entire contents of each data source to the central server. As an example, a *GlOSS*-based meta-information query facility has been implemented for WAIS servers.[14] Recently, reference [7] describes the application of inference networks (from traditional information retrieval) to the text database discovery problem. Their approach summarizes databases using document frequency information for each term (the same type of information that *GlOSS* keeps about the databases), together with the "inverse collection frequency" of the different terms.

Another approach to solving the effectiveness and performance problems is to design a more sophisticated broker architecture. In principle, we can achieve greater effectiveness by creating brokers that specialize in a certain topic. Scalability comes from removing the central server bottleneck. In Indie (shorthand for "Distributed Indexing") [10, 9] and Harvest [5], each broker knows about some subset of the data sources, with a special broker that keeps information about all other brokers. Reference [34] and WHOIS++ [45] allow brokers (index-servers in WHOIS++) to exchange information about sources they index, and to forward queries they receive to other knowledgeable brokers. Reference [13] describes a system that allows sites to forward queries to likely sources (based, in this case, on what information has been received from that source in the past).

While there have been many proposals for how to summarize database contents and how to use the summaries to answer queries, there have been very few performance studies in this area. Reference [34] includes a simulation study of the effectiveness of having brokers exchange content summaries, but is not concerned with what these content summaries are, nor with the costs of storing and exchanging them. Reference [7] studies the inference network approach experimentally. Likewise, [16, 17, 15] examine the effectiveness and storage efficiency of *GlOSS* without worrying about costs of access and update.

On a related topic, the fusion track of the TREC conference [20, 21] has produced papers on the "collection fusion" problem [44, 43]. These papers study how to merge query results from multiple data sources into a single query result so as to maximize the number of relevant documents that users get from the distributed search. Reference [7] also studies this problem.

The representation of summary information for distributed text databases is clearly important for a broad range of query systems. Our article goes beyond existing work in addressing the storage of this information, and in studying the performance of accesses and updates to this information.

---

[11]Lycos is accessible at `http://www.lycos.com`.
[12]AltaVista is accessible at `http://www.altavista.digital.com`.
[13]The World-Wide Web Worm is accessible at `http://wwww.cs.colorado.edu/wwww`.
[14]*GlOSS* is accessible at `http://gloss.stanford.edu`.

9 Conclusion

The investigation reported in this paper represents an important step toward making *GlOSS* a useful tool for large scale information discovery. We showed that *GlOSS* can, in fact, effectively distinguish among text databases in a large system with hundreds of databases. We further identified partitioned hashing and grid files as useful data structures for storing the summary information that *GlOSS* requires. We showed that partitioned hashing offers the best average case performance for a wide range of workloads, but that performance can degrade dramatically as the amount of data stored grows beyond initial estimates. The grid file can be tuned to perform well, and does not require any initial assumption about the ultimate size of the summary information. Our work on tuning grid files demonstrates that good performance can be achieved even for highly skewed data.

We examined how the characteristics of the *GlOSS* summaries make the policy for splitting blocks of the grid file a critical factor in determining the ultimate row and column access costs, and evaluated several specific policies using databases containing U.S. Patent Office data. Our investigation showed that if the expected ratio of row accesses to column accesses is very high (greater than about 1000:1 in our experiment), the best policy is to always split between words. Some existing distributed information retrieval services exceed this high ratio. If the ratio is very low or if updates exceed queries, the best policy is to split between databases whenever possible. Between these extremes, a policy of splitting between databases with a given probability can be used to achieve the desired balance between row and column access costs. For a given probability (and expected number of database splits, ds), this policy performs better than policies that prepartition the database scale ds times, or that always divide on the database scale up to ds times. If it is important to have a firm bound on query costs, policies that prepartition or divide the database scale a fixed number of times can be used.

More work is needed to explore the utility of the *GlOSS* summaries as a representation of summary information for brokers. Their effectiveness should be studied in a more realistic environment with real databases and matching queries, where the queries involve disjunction as well as conjunction. There is more work to be done on the storage of these summaries as well. An unfortunate aspect of the grid files is their need for a relatively large directory. Techniques have been reported for controlling directory size [3]; we must examine whether those techniques are applicable to the highly-skewed grid files generated by the *GlOSS* summaries. Compression techniques [47] would have a significant impact on the performance figures reported here. Finally, building an operational *GlOSS* server for a large number of real databases is the only way to truly determine the right ratio between word and database access costs.

On a broader front, many other issues remain to be studied. The vastly expanding number and scope of online information sources make it clear that a centralized solution to the database discovery problem will never be satisfactory, showing the need to further explore architectures based on hierarchies [15] or networks of brokers.

References

[1] Alfred V. Aho and Jeffrey D. Ullman. Optimal partial-match retrieval when fields are independently specified. *ACM Transactions on Database Systems*, 4(2):168–179, June 1979.

[2] Daniel Barbará and Chris Clifton. Information Brokers: Sharing knowledge in a heterogeneous distributed system. Technical Report MITL-TR-31-92, Matsushita Information Technology Laboratory, October 1992.

[3] Ludger Becker, Klaus Hinrichs, and Ulrich Finke. A new algorithm for computing joins with grid files. In *Proceedings of the* 9^{th} *International Conference on Data Engineering*, pages 190–197, 1993.

[4] Tim Berners-Lee, Robert Cailliau, Jean-F. Groff, and Bernd Pollermann. World-Wide Web: The Information Universe. *Electronic Networking: Research, Applications and Policy*, 1(2), 1992.

[5] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado-Boulder, August 1994.

[6] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast incremental indexing for full-text information retrieval. In *Proceedings of the* 20^{th} *International Conference on Very Large Data Bases*, pages 192–202, 1994.

[7] James P. Callan, Zhihong Lu, and W. Bruce Croft. Searching distributed collections with inference networks. In *Proceedings of the* 18^{th} *Annual SIGIR Conference*, 1995. To appear.

[8] Doug Cutting and Jan Pedersen. Optimizations for dynamic inverted index maintenance. In *Proceedings of the* 13^{th} *International Conference on Research and Development in Information Retrieval*, pages 405–411, 1990.

[9] Peter B. Danzig, Jongsuk Ahn, John Noll, and Katia Obraczka. Distributed indexing: a scalable mechanism for distributed information retrieval. In *Proceedings of the* 14^{th} *Annual SIGIR Conference*, October 1991.

[10] Peter B. Danzig, Shih-Hao Li, and Katia Obraczka. Distributed indexing of autonomous Internet services. *Computer Systems*, 5(4), 1992.

[11] Andrzej Duda and Mark A. Sheldon. Content routing in a network of WAIS servers. In *14th IEEE International Conference on Distributed Computing Systems*, 1994.

[12] Christos Faloutsos. Multiattribute hashing using Gray codes. In *Proceedings of the 1986 ACM SIGMOD Conference*, pages 227–238, 1986.

[13] David W. Flater and Yelena Yesha. An information retrieval system for network resources. In *Proceedings of the International Workshop on Next Generation Information Technologies and Systems*, June 1993.

[14] Michael Freeston. A general solution of the n-dimensional b-tree problem. In *Proceedings of the 1995 ACM SIGMOD Conference*, pages 80–91, May 1995.

[15] Luis Gravano and Héctor García-Molina. Generalizing *GlOSS* for vector-space databases and broker hierarchies. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 78–89, September 1995.

[16] Luis Gravano, Héctor García-Molina, and Anthony Tomasic. The effectiveness of *GlOSS* for the text-database discovery problem. In *Proceedings of the 1994 ACM SIGMOD Conference*, May 1994. Also available as `ftp://db.stanford.edu/pub/gravano/1994/-stan.cs.tn.93.002.sigmod94.ps`.

[17] Luis Gravano, Héctor García-Molina, and Anthony Tomasic. Precision and recall of *GlOSS* estimators for database discovery. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems (PDIS'94)*, September 1994. Also available as `ftp://db.stanford.edu/pub/gravano/1994/stan.cs.tn.94.010.pdis94.ps`.

[18] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD Conference*, pages 47–57, June 1984.

[19] D. K. Harman, editor. *Overview of the Third Text REtrieval Conference (TREC-3)*. U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology (NIST), 1995. NIST Special Publication 500-225, Coden: NSPUE2.

[20] Donna Harman, editor. *Proceedings of the Third Text Retrieval Conference (TREC-3)*. National Institute of Standards and Technology, Special Publication 500-225, 1995.

[21] Donna Harman, editor. *Proceedings of the Fourth Text Retrieval Conference (TREC-4)*. National Institute of Standards and Technology, 1996.

[22] Klaus Hinrichs. Implementation of the grid file: design concepts and experience. *BIT*, 25:569–592, 1985.

[23] Brewster Kahle and Art Medlar. An information system for corporate users: Wide Area Information Servers. Technical Report TMC199, Thinking Machines Corporation, April 1991.

[24] Donald E. Knuth. *The art of computer programming: Volume 3/Sorting and searching*. Addison-Wesley, 1973.

[25] John W. Lloyd. Optimal partial-match retrieval. *BIT*, 20:406–413, 1980.

[26] John W. Lloyd and K. Ramamohanarao. Partial-match retrieval for dynamic files. *BIT*, 22:150–168, 1982.

[27] B. Clifford Neuman. The Prospero File System: A global file system based on the Virtual System model. *Computer Systems*, 5(4), 1992.

[28] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[29] Katia Obraczka, Peter B. Danzig, and Shih-Hao Li. Internet resource discovery services. *IEEE Computer*, September 1993.

[30] Joann J. Ordille and Barton P. Miller. Distributed active catalogs and meta-data caching in descriptive name services. Technical Report #1118, University of Wisconsin-Madison, November 1992.

[31] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *3rd ACM Symposium on Principles of Database Systems*, pages 181–190, April 1984.

[32] Sergio Pissanetzky. *Sparse matrix technology*. Academic Press, 1984.

[33] Gerard Salton and Michael J. McGill. *Introduction to modern information retrieval*. McGraw-Hill, 1983.

[34] Michael F. Schwartz. A scalable, non-hierarchical resource discovery mechanism based on probabilistic protocols. Technical Report CU-CS-474-90, Dept. of Computer Science, University of Colorado at Boulder, June 1990.

[35] Michael F. Schwartz, Alan Emtage, Brewster Kahle, and B. Clifford Neuman. A comparison of Internet resource discovery approaches. *Computer Systems*, 5(4), 1992.

[36] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th Conference on Very Large Databases*, pages 507–518, September 1987.

[37] Mark A. Sheldon, Andrzej Duda, Ron Weiss, James W. O'Toole, and David K. Gifford. A content routing system for distributed information servers. In *Proceedings Fourth International Conference on Extending Database Technology*, 1994.

[38] Patricia Simpson and Rafael Alonso. Querying a network of autonomous databases. Technical Report CS-TR-202-89, Dept. of Computer Science, Princeton University, January 1989.

[39] Anthony Tomasic and Hector Garcia-Molina. Performance issues in distributed shared-nothing information retrieval systems. *Information Processing and Management*, 1996. To Appear.

[40] Anthony Tomasic, Héctor García-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. In *Proceedings of the 1994 ACM SIGMOD Conference*, pages 289–300, 1994.

[41] Anthony Tomasic, Luis Gravano, Calvin Lue, Peter Schwarz, and Laura Haas. Data structures for efficient broker implementation. Technical report, IBM Almaden Research Center, June 1995. Also available as `ftp://db.stanford.edu/pub/gravano/1995/ibm_rj.ps`.

[42] Jeffrey D. Ullman. *Principles of database and knowledge-base systems*, volume I. Computer Science Press, 1988.

[43] Ellen M. Voorhees. Siemens TREC-4 report: Further experiments with database merging. In *Proceedings of the* 4^{th} *Text Retrieval Conference (TREC-4)*, 1996.

[44] Ellen M. Voorhees, Narendra K. Gupta, and Ben Johnson-Laird. The collection fusion problem. In *Proceedings of the* 3^{rd} *Text Retrieval Conference (TREC-3)*, 1995.

[45] Chris Weider and Simon Spero. Architecture of the WHOIS++ Index Service, October 1993. Working draft.

[46] Gio Wiederhold. *File organization for database design*. McGraw-Hill, 1987.

[47] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. An efficient indexing technique for full-text database systems. In *Proceedings of the* 18^{th} *International Conference on Very Large Data Bases*, pages 352–362, 1992.