Focusing Search in Hierarchical Structures with Directory Sets

Guy Jacobson
AT&T Labs–Research
guy@research.att.com

Balachander Krishnamurthy
AT&T Labs–Research
bala@research.att.com

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

Dan Suciu
AT&T Labs–Research
suciu@research.att.com

Abstract

Keyword-based searches on the World Wide Web are often of limited use, because they return too many uninteresting matches. We propose here a novel mechanism that permits the user to specify *directory sets* to restrict the space of documents searched, and, at the same time, increase the speed of the search. We view the Web as a single, huge hierarchy, reflected in the structure of the URLs. We refer to each subtree in this hierarchy as a *directory*, and group semantically related documents from multiple directories into a *directory set*. Starting from a collection of pre-defined directory sets, a user can dynamically generate new directory sets by using operators in a *directory set algebra*, and focus keyword-based searches to documents that belong to a (pre-defined or dynamically generated) directory set. We design algorithms for efficiently evaluating expressions in the directory set algebra, and describe a technique for tightly integrating directory sets into keyword-based search engines. Finally, we validate our ideas using a realistic case study for a significant fragment of the AT&T intranet, and 211 pre-defined directory sets.

1 Motivation

The World Wide Web is a very large and rapidly growing repository of documents. The utility of the Web is considerably enhanced by the presence of search services (e.g., Alta Vista[1], Yahoo[2]) that locate documents matching user queries, and the explosion of search services (see, e.g., the search services listed under the "Internet Search" feature provided by Netscape Navigator and other Web browsers) is testimony to the increasing importance of the problem of searching for documents on the Web. There are two main approaches that have been used with considerable success on the Web: *search engines* and *web guides*.

Search engines such as Alta Vista essentially use text indexing techniques to index a large fraction of the documents on the Web. Queries to these search engines are typically specified by providing keywords, and the query answers are pointers (URLs) to matching documents. There are many advantages of this approach: several search engines do an effective job of finding many possible matching documents, the index can be kept (reasonably) up-to-date in an automatic fashion, and the approach is scalable. The effectiveness of these search engines is, however, limited by the fact that searching by keywords alone returns many matches that may not be of interest to the user, and the large number of matches makes it difficult to locate the few documents really being sought.

Web guides such as Yahoo provide a topic-based hierarchical classification that allow users to navigate the topic hierarchy and pose keyword-based queries to locate documents classified under any specific topic of interest. The principal advantage of this approach is that the manual classification mechanism used to populate the topic-based hierarchy results in good quality matches, and such Web guides are very useful to locate a few matching documents. The utility of such Web guides is restricted because of two distinct reasons. First, the problem of having to manually classify a rapidly growing collection of documents into specific topics is a human/computationally intensive task; hence, the documents that can be located using such Web guides may miss many of the documents that would be of interest to the user, because they had not been manually classified. Second, navigating the topic/sub-topic hierarchy is a very rigid mechanism of focusing the search and does not allow users to pose keyword-based queries to locate documents classified under, e.g., both `Education` and `Environment`, in the Yahoo topic/sub-topic hierarchy.

In this paper, we propose a novel mechanism for focusing searches that combines elements of each of these two approaches in a scalable fashion:

> Keywords are used to locate documents of interest, and focusing is provided by overlaying the notion of *directories of interest* on the physical structure of the document space, and restricting matching documents to be from these directories of interest.

Our mechanism is light-weight, and relies on the following two complementary observations. First, documents at Web sites are typically organized using a hierarchical structure, as in the UNIX file system, and Web sites can be organized into a hierarchical structure, based on the domain name hierarchy. Second, the hierarchical file system structure is a natural, decentralized and scalable mechanism for

[1] http://altavista.digital.com/.
[2] http://www.yahoo.com/.

organizing a large space of documents, and for grouping related documents together. Keeping these two observations in mind, we develop the presentation of our mechanism for focusing keyword-based searches as follows:

- We introduce the notion of a *directory set* to group a set of related directories, where each *directory* is the set of all documents contained in a single sub-tree in the hierarchical structure (Section 2). Starting from a pre-defined collection of named directory sets, users can dynamically create new directory sets using operators in a *directory set algebra*, that is adapted from an algebra introduced in [10] for indexing text databases.

- We have designed and implemented a uniform and efficient table driven evaluation algorithm for expressions in the directory set algebra (Section 3.1). In our implementation, the named directory sets need to be sorted lexicographically, and the resulting directory set is computed with a single scan of each of the named directory sets.

  We also discuss two practical implementation issues: (a) using B-trees over directory sets to compute the result of an algebraic expression in an I/O efficient manner (Section 3.2); and (b) a tight integration of the keyword search process, based on the use of inverted indexes, with directory sets filtering (Section 3.3).

- We have performed a realistic case study by applying directory sets to searching in a fragment of the documents in the AT&T intranet (Section 4). Our experiments show: (1) it is possible to compile a meaningful collection of directory sets for a large intranet like AT&T's, from existing resources, with minimal effort; (2) the use of directory sets substantially improved the search quality, by reducing the number of documents retrieved by a factor of 4 to 18; (3) the running time decreased too, by a factor of 1.5 to 3.2.

Example 1.1 Consider the following companies-related directory sets: `Telecom-Companies`, containing directories corresponding to Web pages of telecommunication companies like AT&T, MCI, Sprint, etc.; `Fortune-500-Companies`, containing the directories for the Fortune 500 companies, e.g., American Express, AT&T, IBM, etc.; and `Research-Labs`, containing directories for research labs and organizations, e.g., AT&T Labs, IBM Almaden, IBM TJ Watson, and university research labs like the University of Pennsylvania's IRCS.

If a user wants to find documents about the AT&T split, currently he can use a search engine to look for the phrase "AT&T split"; Alta Vista finds several hundred document matches.[3] In our system, the user can focus the keyword-based search by additionally specifying the directory set `Telecom-Companies`, containing URLs of Web sites classified as belonging to telecommunication companies; if this still returns too many documents, he can narrow the search even further, by restricting to a dynamically generated directory set, e.g., only to those `Telecom-Companies` that are also `Fortune-500-Companies`.

The intent is for directory sets to be compiled manually, or compiled from existing resources, by inserting the root of each directory. However, directory sets are rather robust with respect to changes; for example, we do not need to change any of the above when new documents are created in IBM Almaden. Note that different directory sets may have common directories, e.g., `Telecom-Companies` and `Fortune-500-Companies` both contain AT&T's directory; also, they may have overlapping directories, like AT&T Labs' directory (in `Research-Labs`) which is properly contained in AT&T's directory.

Finally, we would like to emphasize the flexibility of our notion of *directory*; this can be either a complete domain name (e.g., `www.research.att.com`), a complete domain name augmented with a path name in that domain (e.g., `www.cis.upenn.edu/~ircs`), or a partial domain name (e.g., `att.com`). Each directory should be understood as the collection of all documents "underneath". □

[3] But returns only 200; the rest are lost to the user.

2 Directory Sets and the Directory Set Algebra

In this section, we first introduce the notions of directories and directory sets in hierarchical structures, and then describe our directory set algebra, which can be used to create dynamically defined directory sets. Where it adds intuition, we discuss analogies between our concepts with the work on manipulation of boolean queries in text retrieval systems.

2.1 Hierarchical Structures

We assume a finite set of hierarchically structured elements to be given. We denote this set by $\mathcal{U}$, for *universe*. Each element $u \in \mathcal{U}$ has a set of successors (or children), at most one predecessor (or parent), and the resulting graph forms a forest or a tree. For each element $u \in \mathcal{U}$, we denote with $u\!\downarrow\, \subseteq \mathcal{U}$ the set of descendants of u, i.e., u itself, its children, its grandchildren, etc. A *directory* is any set r, $r \subseteq \mathcal{U}$, of the form $r = u\!\downarrow$, for some $u \in \mathcal{U}$. A *directory set* is a set of directories, i.e., $s = \{r_1, \ldots, r_n\}$, where $r_1, \ldots, r_n \subseteq \mathcal{U}$ are directories. Finally, we associate names to certain directory sets, and then call them *named* directory sets.

Example 2.1 Consider the Web example. Document URLs have two main components, the complete domain name of the Web site (e.g., `www.cis.upenn.edu`), and the path name of the document within the file system at the Web site (e.g., `~ircs/index.html`). Each of these two components is hierarchically organized. The domain name hierarchy consists of domain names and parts thereof (like `edu`, `upenn.edu`, etc), and the path name hierarchy at a Web site consists of a fully qualified document name and parts thereof (like `~ircs`, `~ircs/index.html`). The composition of these two hierarchies, obtained by composing the hierarchical space of path names under the domain name hierarchy can be viewed as a single huge hierarchy. Here `edu` has `upenn.edu` and `wisc.edu` as successors; `upenn.edu` has `cis.upenn.edu` as a successor, which in turn has `www.cis.upenn.edu` as a successor; and the latter has `www.cis.upenn.edu/~ircs` as a successor, etc.

A directory on the Web consists of a sub-tree in this hierarchy. For example, AT&T's directory is $\texttt{att.com}\!\downarrow$, the `Telecom-Companies` directory set from Example 1.1 could be defined as $\{\texttt{att.com}\!\downarrow, \texttt{mci.com}\!\downarrow, \ldots\}$, and the directory set `Research-Labs` could be defined as $\{\texttt{research.att.com}\!\downarrow, \texttt{almaden.ibm.com}\!\downarrow, \texttt{research.microsoft.com}\!\downarrow, \ldots\}$.

An example use of directory sets in the Web context might be to search for the keywords "unified messaging" in the `Telecom-Companies` directory set. □

The documents in a hierarchical structure correspond to the leaves in the hierarchy $\mathcal{U}$ (files in the UNIX file system, and Web documents in the WWW). One should understand

$\mathcal{U}$ as a constantly changing universe; files are being added and deleted, the set of URLs is continuously changing. We would like a directory set's definition to be valid for a longer period of time. Hence, it is a bad idea to specify a directory by enumerating all the leaves in its sub-tree, since this enumeration will quickly become obsolete. We overcome this problem by *specifying* a directory $u\downarrow$ by its root u. Thus a directory set $s = \{u_1\downarrow, \ldots, u_n\downarrow\}$ will be specified by enumerating its roots: $\{u_1, \ldots, u_n\}$. The hope is that nodes higher in the hierarchy change less often than nodes lower in the hierarchy. For example, we would specify the directory set `Research-Labs` by enumerating only the roots of the research labs' directories. Additions and deletions of URLs inside each research lab directory will not affect this specification. Of course, when a new research lab is created, our specification has to be modified.

Note that this view differs considerably from the approach taken by Yahoo; there, each *topic* is an enumeration of all documents that have been manually classified as belonging to that topic, hence any small change in a Web site's structure requires human intervention to keep the classification up to date. Another key difference between our directory sets and Yahoo's topics is that we can dynamically generate new directory sets, using a *directory set algebra*, to be described next.

Discussion Text retrieval systems are based on the manipulation of sets of documents and of intensional descriptions of sets of documents. A boolean query system, for example, interprets a word in a query as standing for a primitive set of documents containing that word, and many boolean query systems let one use, in addition to words, truncation or wildcard operators to form expressions of the form "`comput*`", which is taken to mean "all documents containing a word whose prefix is `comput`".

In this context, a directory can be viewed as a wildcard expression describing the set of documents in a sub-tree of the hierarchy, and a named directory set can be viewed as a *standing query* (or, *hedge*) that is the logical disjunction of the wildcard expressions corresponding to its constituent directories. Consider again Example 2.1. The AT&T directory `att.com`$\downarrow$ can be specified using a wildcard expression of the form "`URL:*.att.com/*`", and the `Telecom-Companies` directory set could be specified as "`URL:*.att.com/*` OR `URL:*.mci.com/*` OR ...".

We prefer the hierarchical model over the (syntactic) wildcard expression approach for specification of directories and directory sets in order to (a) support the right level of abstraction (thus, the wildcard approach would result in different expressions for directory sets on the Web from those in the UNIX file system); and (b) design appropriate data structures and efficient algorithms for the manipulation of directory sets.

2.2 A Directory Set Algebra

Once we have specified a fixed number of named directory sets, we can dynamically create new directory sets by applying certain operators on the named ones. The operators form an algebra of directory sets in a hierarchy, which we call **H**. It is derived from the PAT algebra for indexing text databases, introduced in [10] and further discussed in [3, 4]. We assume a fixed number of named directory sets to be given, $R_1, R_2, \ldots, R_k$. Then the expressions in **H** are:

$$\begin{aligned} E \;\; ::= \;\; & R_i \mid E \cup E \mid E \cap E \mid E - E \mid \\ & E \subset E \mid E \subseteq E \mid E \supset E \mid E \supseteq E \end{aligned}$$

Here E denotes an algebra expression, while R_i denotes the name of a named directory set. All expressions denote directory sets. For the meaning of these operators, recall that every directory set is a *set of sets of elements*. First $\cup, \cap$ and $-$ have obvious set-theoretic meanings:

$$\begin{aligned} s \cup s' &\stackrel{\text{def}}{=} \{r \mid r \in s \vee r \in s'\} \\ s \cap s' &\stackrel{\text{def}}{=} \{r \mid r \in s \wedge r \in s'\} \\ s - s' &\stackrel{\text{def}}{=} \{r \mid r \in s \wedge r \notin s'\} \end{aligned}$$

Next:

$$\begin{aligned} s \subset s' &\stackrel{\text{def}}{=} \{r \mid r \in s \wedge \exists r' \in s', r \subset r'\} \\ s \subseteq s' &\stackrel{\text{def}}{=} \{r \mid r \in s \wedge \exists r' \in s', r \subseteq r'\} \\ s \supset s' &\stackrel{\text{def}}{=} \{r \mid r \in s \wedge \exists r' \in s', r \supset r'\} \\ s \supseteq s' &\stackrel{\text{def}}{=} \{r \mid r \in s \wedge \exists r' \in s', r \supseteq r'\} \end{aligned}$$

Example 2.2 Consider Example 2.1, and the search for documents matching keywords "unified messaging". Even if we restrict the search to the `Telecom-Companies` directory set, we may get too many matching documents. The key observation here is that the documents most likely to be of interest to us (as researchers) are not just in *arbitrary* telecommunication companies, but in ones that have a research lab. Hence, we construct a directory set dynamically:

$$s_1 \stackrel{\text{def}}{=} \texttt{Telecom-Companies} \supset \texttt{Research-Labs}$$

and use it to filter the search. If the Web pages of a research lab are strictly contained in the Web pages of its parent company, then s_1 is equivalent to the following dynamically generated directory set:

$$s_2 \stackrel{\text{def}}{=} \texttt{Telecom-Companies} \supseteq \texttt{Research-Labs}$$

But suppose that there exists some telecommunication company that has a Web site only for its research lab; then s_1 would not contain that company, while s_2 would. This explains why we need both variations: $\supset$ and $\supseteq$. □

We emphasize that the operators in this algebra generate new *directory sets*, not new *directories*. The maximal set of directories is the union of all named directory sets, and algebra operations return various subsets of this maximal set. This is analogous to the case of databases and database queries; the atomic elements of the database are fixed (and form the *active domain*), and each query returns some relation built from these elements.

The reader may be puzzled by the apparently ad-hoc choice of operators in **H**. In the full version of this paper [6], we present a formal study of all possible choices for such operators, and analyze their properties. We show that there exist exactly 16 "positive" and 16 corresponding "negative" possible operators. Out of these, **H** can express 9 positive and 9 negative ones. We then show that exactly 8 of the positive operators (and 8 corresponding negative ones) admit efficient table driven evaluation[4] and show how to derive the table for each of them: we give the table for the operators in **H** in Section 3. Efficiency of evaluation thus provides an a posteriori justification of our choice of operators in **H**.

[4] The ninth positive operator is set union $\cup$, which can be efficiently evaluated using list merging, although not by the table in Section 3.

Discussion Consider again our analogy where directories and named directory sets can be viewed as akin to wildcard expressions and standing queries, respectively, in the intensional descriptions of sets of documents. In boolean query systems, expressions in the boolean algebra are interpreted as standing for sets of documents built up by intersection and union of the primitive sets. Various manipulations can be done on the intensional descriptions without knowing the extension of the primitive sets, with the goal of producing a final form whose extension can be computed efficiently.

In a boolean query system, the truncated form "`comput* AND computer*`", e.g., can be simplified to "`computer*`". This is similar to the simplification that can be achieved in our case when computing the set of documents in the intersection of the IBM directory (`ibm.com`$\downarrow$) and the IBM Almaden directory (`almaden.ibm.com`$\downarrow$); the result is precisely the IBM Almaden directory. A crucial difference between our notions of intensional descriptions of sets of documents and those in boolean query systems is that while one document can contain different words with different prefixes, one document can *only live in one position in a hierarchy*. Thus, while "`comput* AND communic*`" is not the empty set, there are no documents in the intersection of the AT&T directory (`att.com`$\downarrow$) and the MCI directory (`mci.com`$\downarrow$).

This difference between directories and truncated terms provides two key advantages for our approach: (a) It allows us to perform many more simplifications of our intensional descriptions (which are tailored to the hierarchical model, and much richer than boolean expressions); and (b) The manipulations on the intensional descriptions can be performed very efficiently (we discuss this in Section 3).

2.3 Additional Operators

For practical purposes we found it convenient to introduce additional derived operators. They are handy in compactly expressing queries and, as we shall see in Section 3, they are more efficiently computed directly, rather than using their definition. These derived operators are:

$$\begin{aligned}
s \nabla s' &\stackrel{\text{def}}{=} \{r \mid r \in s \wedge \exists r' \in s', r \cap r' \neq \emptyset\} \\
&= (s \subseteq s') \cup (s \supseteq s') \\
s \not\subset s' &\stackrel{\text{def}}{=} s - (s \subset s') \\
s \not\subseteq s' &\stackrel{\text{def}}{=} s - (s \subseteq s') \\
s \not\supset s' &\stackrel{\text{def}}{=} s - (s \supset s') \\
s \not\supseteq s' &\stackrel{\text{def}}{=} s - (s \supseteq s')
\end{aligned}$$

Example 2.3 Suppose that we have the two directory sets `Research-Labs` and `Computer-Science-Depts`, and we want to focus a keyword search to only those `Research-Labs` that are affiliated with `Computer-Science-Depts`. The problem is that we may expect two kinds of relationships between a Research Lab and a Computer Science Department: either the Research Lab is included in the Computer Science Department (like `www.cis.upenn.edu/~ircs` which is included in `www.cis.upenn.edu`), or conversely, the Computer Science Department may be included in the Research Lab (like in many industry research labs). A concise expression that denotes the directory set of interest is:

$$s_3 \stackrel{\text{def}}{=} \texttt{Research-Labs} \nabla \texttt{Computer-Science-Depts}$$

Suppose now that we are interested in obtaining documents from research labs that are not contained in telecommunication companies, we can dynamically construct the directory set:

$$s_4 \stackrel{\text{def}}{=} \texttt{Research-Labs} \not\subseteq \texttt{Telecom-Companies}$$

and use it to filter the search. □

2.4 Ordered Hierarchies

So far our hierarchical universe $\mathcal{U}$ is unsorted; for element $u \in \mathcal{U}$, there is no order on its successors. In practice, nodes in $\mathcal{U}$ are denoted by strings, which leads naturally to an order. We believe this order to have no semantic meaning, so we did not include in **H** any operators referring to this order (unlike [3, 4], where order in a text *has* semantic meaning, and there are operators in their algebra whose semantics depends on this order). However we *do* exploit this order to derive efficient implementations.

Assume that each node in the tree has a name, which is a string. We allow several nodes to have the same name, but we require that each node in the hierarchy be uniquely identified by the path leading to it; that is, it can be uniquely identified by concatenating the names of each node along the path from the root of the tree to that node, and separating them with some special character like "/". This is exactly like in the UNIX file system, where files may share common names, but each file must be uniquely identified by its fully qualified path name.

Whenever $\mathcal{U}$ is represented in this way we can sort its elements lexicographically. We say in this case that $\mathcal{U}$ is an *ordered hierarchy*.

3 Efficient Evaluation of Focused Queries

Our approach of focusing keyword-based search requires the user to augment a keyword-based query Q with an expression E in our directory set algebra: a typical such query is "find all documents matching keyword w whose URLs are in directory set generated from expression E". Answering such an augmented query efficiently consists of two parts: (1) Evaluating/simplifying the expression E, resulting in a (dynamically generated) directory set s_E; and (2) Optimizing the evaluation of the keyword-based query Q against the directory set s_E. We address the first part in Sections 3.1 and 3.2, and the second part in Section 3.3.

3.1 Operator Driven Evaluation of Directory Set Expressions

The evaluation of expressions in the directory set algebra relies on a generalization of the algorithm for computing the intersection of two sorted lists. The algorithm uniformly evaluates all operators in **H**, except set union $\cup$ (for the latter we used a standard merge algorithm). In terms of complexity, the algorithm is linear in the sizes of the input named directory sets. This may be too high, when the named directory sets are large, and in Section 3.2, we describe how our uniform algorithm can be extended to take advantage of the presence of B-trees over large named directory sets.

Our directory set algebra is defined over hierarchies, and there are four possible relationships between two nodes u_1 and u_2 in the hierarchy that are of interest: u_1 is a proper ancestor of u_2, u_1 and u_2 are identical, u_2 is a proper ancestor of u_1, or none of these. Only one of these four relationships can hold between u_1 and u_2. Evaluating an expression in our directory set algebra essentially requires the ability to determine these relationships between *sets of nodes*. When our hierarchical structure is ordered, and the hierarchy nodes are represented as strings (as described in Section 2.4), these four relationships have string counterparts: the string representation of u_1 is a prefix of the string representation of u_2, the string representations of u_1 and u_2 are identical, the string representation of u_2 is a prefix of the string representation of u_1, or none of these.

We now present an algorithm, `TableDrivenEval`, that takes two lexicographically sorted directory sets s_1 and s_2, and one of the operators $\cap, \supset, \supseteq, \subset, \subseteq, \nabla$ or their negative counterparts, and computes a lexicographically sorted directory set s_1 `op` s_2. While `TableDrivenEval` does not handle $\cup$, the set union of two lexicographically sorted directory sets can be easily computed by merging the two directory sets. Hence, given any expression in the directory set algebra, the result of an inner operator in the expression can be pipelined to an outer operator in the expression.

`TableDrivenEval` uses two pointers, P_i for directory set $s_i, i = 1, 2$, to scan the elements of the two directory sets. Initially, pointer $P_i, i = 1, 2$ is positioned at the first element of directory set s_i. For simplicity of presentation, assume that each of the input directory sets is terminated by a special string that is lexicographically after all the strings in the two directory sets. Suppose at some point in the computation, pointer P_1 is positioned at element u_1 in directory set s_1, and pointer P_2 is positioned at element u_2 in directory set s_2. For the evaluation of the operators considered by `TableDrivenEval`, five possible relationships between strings u_1 and u_2 are worth distinguishing:

1. u_1 is lexicographically before u_2, while not being a proper prefix of u_2, denoted by $u_1 <_l u_2$.
2. u_1 is a proper prefix of u_2, denoted by $u_1 <_p u_2$.
3. u_1 and u_2 are identical, denoted by $u_1 = u_2$.
4. u_2 is a proper prefix of u_1, denoted by $u_1 >_p u_2$.
5. u_2 is lexicographically before u_1, while not being a proper prefix of u_1, denoted by $u_1 >_l u_2$.

Since each of the operators considered by the algorithm `TableDrivenEval` can be viewed as selecting elements from their left operand based on some property satisfied by the selected element with respect to elements of the right operand, three possible actions can be taken when comparing u_1 and u_2:

Advance-P1 : Nothing is output and pointer P_1 is advanced.

Print-P1 : Element u_1 is output and pointer P_1 is advanced.

Advance-P2 : Nothing is output and pointer P_2 is advanced.

Algorithm `TableDrivenEval` terminates when P_1 is positioned beyond the end of directory set s_1.

Table 1 summarizes the action taken by the algorithm `TableDrivenEval`, for the various possible operators considered by the algorithm, for the five relationships between strings u_1 and u_2 mentioned above. For example, consider the case when the operator is $\subset$. A directory represented by string u_1 in directory set s_1 is in the output of $s_1 \subset s_2$, provided there is a directory represented by string u_2 in directory set s_2 such that u_2 is a proper prefix of u_1. This is indicated by the entry Print-P1 in the table corresponding to the column marked $u_1 >_p u_2$ and the row marked $\subset$. When u_1 is lexicographically before or identical to u_2, we are guaranteed that no element in s_2 after u_2 can be a witness to the presence of u_1 in the result of $s_1 \subset s_2$; hence, pointer P_1 can be advanced. When u_2 is lexicographically before but not a proper prefix of u_1, a subsequent element in s_2 could still be a witness to the presence of u_1 in the result of $s_1 \subset s_2$; hence, pointer P_2 can be advanced. We prove the correctness of this algorithm in the full version of the paper [6].

Proposition 3.1 `TableDrivenEval` *correctly evaluates an expression in the directory set algebra, using the operators* $\cap, \subset, \subseteq, \supset, \supseteq, \nabla, -, \not\subset, \not\subseteq, \not\supset$ *and* $\not\supseteq$.

Since each of the input named directory sets is scanned once by `TableDrivenEval`, and each string operation performed results in one of the pointers being advanced, we have the following result.

Proposition 3.2 `TableDrivenEval` *takes time linear in the sum of the sizes of the input named directory sets.*

3.2 Using B-trees Over Large Directory Sets

When named directory sets have large numbers of directories, `TableDrivenEval` may be too slow. We consider here how it can be sped up with the aid of B-trees.

Consider some operator in $\mathbf{H}$, s_1 `op` s_2. The algorithm `TableDrivenEval` scans both arguments s_1 and s_2 entirely. Since s_1 `op` s_2 is a subset of s_1, when s_2 is much larger than s_1, it makes sense to use an index on s_2 to visit only those few elements that are relevant to the result. It is less obvious how to do a similar improvement when s_1 is much larger than s_2, because in operators like $s_1 \subseteq s_2$, a "small" s_2 may cause the entire s_1 to be part of the result. Here we observe that the result consists of the union of all directories in a small number of "intervals" in s_1, whose endpoints can be computed quickly. For example, when s_2 has one directory, $s_2 = \{v\}$ and $s_1 = \{u_1, u_2, \ldots, u_m\}$, then $s_1 \subseteq s_2$ consists of one interval, $\{u_i, u_{i+1}, \ldots, u_j\}$, whose endpoints u_i, u_j can be located using an index on s_1.

We illustrate some more details for the operator $s_1 \subset s_2$. Suppose that at some point in the computation, pointer P_1 is positioned at element u_1 in directory set s_1, and pointer P_2 is positioned at element u_2 in directory set s_2. If $u_2 <_l u_1$, then we advance P_2 several positions at once, namely up to the first element after u_2 that is either a proper prefix of u_1, or lexicographically after u_1; this can be quickly located using a B-tree on s_2. Similarly, if $u_1 <_l u_2$, then we advance P_1 several positions, up to the first element which is either $<_p u_2$, or lexicographically after u_2. Again, we can use a B-tree on s_1 in this step. Of course, if B-trees were not available a priori, constructing them on the fly could negate any performance benefits due to their subsequent use.

To analyze the the algorithms sketched above, we use techniques described in [7] in the context of *finger trees*. These are *level-linked* trees, i.e., trees in which all nodes at the same level are chained together in a doubly-linked list. In addition, all tree edges are traversable in both directions. A *finger* is a pointer to some key in the tree. The finger tree has the property that a search for a key which is d keys away from a finger can be done in $O(1 + \log d)$ steps. Using two finger trees s_1, s_2, all binary set operations like set

Operator	u_1 *stringop* u_2				
	$u_1 <_l u_2$	$u_1 <_p u_2$	$u_1 = u_2$	$u_1 >_p u_2$	$u_1 >_l u_2$
$\cap$	Advance-P1	Advance-P1	Print-P1	Advance-P2	Advance-P2
$\subset$	Advance-P1	Advance-P1	Advance-P1	Print-P1	Advance-P2
$\subseteq$	Advance-P1	Advance-P1	Print-P1	Print-P1	Advance-P2
$\supset$	Advance-P1	Print-P1	Advance-P2	Advance-P2	Advance-P2
$\supseteq$	Advance-P1	Print-P1	Print-P1	Advance-P2	Advance-P2
∇	Advance-P1	Print-P1	Print-P1	Print-P1	Advance-P2
$-$	Print-P1	Print-P1	Advance-P1	Advance-P2	Advance-P2
$\not\subset$	Print-P1	Print-P1	Print-P1	Advance-P1	Advance-P2
$\not\subseteq$	Print-P1	Print-P1	Advance-P1	Advance-P1	Advance-P2
$\not\supset$	Print-P1	Advance-P1	Advance-P2	Advance-P2	Advance-P2
$\not\supseteq$	Print-P1	Advance-P1	Advance-P1	Advance-P2	Advance-P2

Table 1: Actions Taken By `TableDrivenEval`

intersection, set difference, symmetric difference, etc. can be computed in $O(\binom{m+n}{n})$, where $\binom{m+n}{n} = \frac{(m+n)!}{m!n!}$ [7, pp.240].

In order to achieve this complexity, finger trees require the following change over B-trees: they are (A, B)-trees rather than B-trees. The difference is that each non-leaf node has $\geq A$ and $\leq B$ successors, and the root has ≥ 2 and $\leq B$ successors. Here A and B are two parameters of the tree with $A \geq 2$. Note that a B-tree is precisely an (A, B)-tree with $B = 2A - 1$. Finger trees require that $B \geq 2A$, a condition which ensures that the amortized insertion and deletion cost in a sequence of insertions and deletions is $O(1)$ (it can be as high as $O(\log n)$ for a B-tree).

Our algorithm described above for computing $s_1 \subset s_2$ is similar to binary set operations on finger trees. There is a catch however: the result of such an operation is another finger tree, not a directory set. We need an additional $O(m')$ steps to actually construct the result, where m' is the size of $s_1 \subset s_2$. To see the problem, recall the example above where $s_2 = \{v\}$: then $s_1 \subset s_2$ consists of an interval $\{u_i, u_{i+1}, \ldots, u_j\}$ of s_1. The algorithms on finger trees will construct the result in time $O(\log \frac{(m+1)!}{m!1!}) = O(\log m)$ by re-building a subtree of s_1. But the actual result has size $m' = j - i + 1$ and can be as large as m, hence the total running time becomes $O(m' + \log m)$. In general we have:

Proposition 3.3 *All operators in* $\mathbf{H}$, s_1 op s_2, *can be computed in time* $O(m' + \log\binom{m+n}{m})$, *where* m, n, m' *are the sizes of* s_1, s_2, s_1 op s_2, *and* $\binom{m+n}{n} = \frac{(m+n)!}{m!n!}$.

When $n = O(1)$, then the running time is $O(m' + \log m)$, which is better than `TableDrivenEval`; similarly when $m = O(1)$. But when $n \approx m$, then the running time becomes $O(m+n)$, offering no theoretical improvement over the algorithm `TableDrivenEval`. In practice, however, an improvement can be achieved, except when s_1 and s_2 are fully interleaved.

In general, the entries in Table 1, for each given operator, indicate how B-trees could be used to advance the pointers in the directory sets that are the two operands of the operator. A pointer in a directory set needs to be advanced to the first element after the current element in the directory set where performing the appropriate string operation between (the new values of) u_1 and u_2 could possibly result in a value of Print-P1 in some entry of the row corresponding to that operator in the table.

3.3 Speeding Up Search Engines with Directory Sets

So far we have discussed how to compute the directory set expression E in isolation: the result is a directory set we denote s_E. Now we consider the keyword-based query Q and propose an optimized evaluation of Q against the directory set s_E. This optimized evaluation requires the integration of directory sets into the keyword-based search engine.

Before doing so, we note that it may be considered acceptable for focusing techniques (like our directory sets) to cause performance penalties over pure keyword searches. This is because the simplest way to integrate focusing techniques with search engines is to do post-processing (first do the keyword search Q, then filter the result against s_E) which, of course, is slower than just keyword search. For example, the PAT text search system [5, 10] runs slower when the search is narrowed to text components [5, pp.63].

Web search engines use inverted indexes [9] on keywords for query processing. (Inverted indexes, which are lists of document identifiers in a document collection that contain a term, are pre-computed off-line for each term in the document collection.) The inverted index associated with one term is usually large, since it contains the identifiers of all documents on the Web containing that term. We propose using directory sets to prune portions of the inverted index, during the search process. Since the construction of these inverted lists is under the control of the search engine builder, he can order the lists to follow the same lexicographic order as that of the directory mechanism, and provide a B-tree on top of each of them. Of course, such an ordering of the inverted lists would require new documents to be inserted into B-trees; this would require more in the way of additional incremental indexes and more frequent rebuilds. Since Web search engines such as Alta Vista update their indexes infrequently, but answer a very large number of queries daily, trading slower index update time for faster query response time is a worthwhile tradeoff to make.

Consider a typical directory set enhanced query in the vector-space model, "find all documents matching keywords $\{w_1, \ldots, w_n\}$ whose URLs are in the directory set generated from expression E". There are many query evaluation algorithms in the literature (see, e.g., [11]) for processing keyword-based queries in the vector-space model. Instead of describing how directory sets can be tightly integrated into specific evaluation algorithms, we outline (in a generic way) how directory sets can be used to prune portions of the inverted lists that are processed by the various query evaluation algorithms. While processing inverted list s_i associated with keyword w_i, first filter its entries against the directory set s_E. This requires the computation of $s_i \subseteq s_E$, for which we use the B-trees available on the inverted list s_i, as explained in Section 3.2. Since s_E is expected to be much smaller than s_i, using a B-tree on the inverted list s_i can help locate the relevant "intervals" in the inverted index quickly. This is faster than a search *without* a directory set, because the search engine does not have to scan the entire inverted list s_i for w_i. For large inverted lists, the additional cost involved in traversing the non-leaf nodes of the B-tree in this technique is expected to be small.

4 Intranet Directory Sets: A Realistic Case Study

In this section, we investigate the benefits of using directory sets and the directory set algebra to restrict keyword-based searches over a significant fragment of documents in the AT&T intranet.

Creating Directory Sets The AT&T intranet consists of a large collection of sites corresponding to, e.g., different organizations and projects within AT&T. To facilitate easy browsing and searching of the AT&T intranet sites by category, AT&T librarians created a few hundred categories, some based on geography (e.g., Ohio, Midwest), some based on organizational units (e.g., Human Resources), and some based on AT&T service offerings (e.g., AT&T WorldNet, AT&T Universal Card Services). The various sites within AT&T were then manually classified into these various categories, and individual sites were often classified into multiple categories (e.g., a site may be classified into a geographical category as well as into appropriate service offering categories).

From this database of sites, categories and mapping of sites to categories [1], we created 211 directory sets, one per category, using UNIX tools and scripts. Each directory set consists of an average of about 31 directories, one directory per AT&T intranet site; each directory is contained in an average of about 10 directory sets.

AT&T is certainly not unique in this respect, and most large organizations either already have, or can easily assemble, a listing of various sites in their intranets and a mapping of these sites to various categories of interest. Similar categories can be found for the Web. Thus, the first lesson that we learn from our case study is that *it is possible to compile a meaningful collection of directory sets for a large intranet like AT&T's, with minimal effort, from existing resources.*

Gathering and Indexing Intranet Documents For the purpose of the experiments reported here we collected 385,398 documents from within the AT&T intranet, comprising a little over 6 Gigabytes of textual data. We organized this collection of documents into a single, huge UNIX hierarchy, reflected in the structure of the document URLs, obtained by composing the hierarchical space of documents at each Web site with the site name hierarchy, as described in Example 2.1.

We attempted to index this collection of documents using the Glimpse 4.0B1 text indexing system from the University of Arizona ([8] and `http://glimpse.cs.arizona.edu/`). However, we ran into the 32-bit problem: Glimpse cannot correctly index a document collection over 2 Gigabytes in size.[5] Consequently, we had to *thin* our collection of documents to under 2 Gigabytes.

We achieved this in an automated manner, by examining the total space occupied by all documents in each of the top-level directories under `com/att`, i.e., in the domains directly below the `att.com` domain. We sorted this list of (top-level) directories based on the total space occupied, and retained every fourth directory. Using this approach we ended up with 105,964 documents comprising about 1.5 Gigabytes of textual data, and were able to successfully index it using Glimpse 4.0B1.

Glimpse Indexing Glimpse is a popular utility to speed up searches within text documents in file systems. Glimpse partitions the entire document collection into 256 blocks[6], with the total size of files in each block roughly the same [8]. It then creates an index on blocks, rather than files. Given a search pattern, Glimpse operates in two stages: (1) It looks up the index and retrieves a set of blocks (each block contains at least one file matching the pattern); let us denote with D_p the total number of files in these blocks. (2) Glimpse runs a modified `grep` on all D_p files, and retrieves the final answer; we denote with D_a the total number of files retrieved, $D_a \leq D_p$.

We integrated directory sets into Glimpse by interposing another stage between (1) and (2) — call it (1a). This stage computes the directory set expression and filters the D_p documents against the directory sets; we will denote with D_d the number of documents after the filtering. Only these D_d documents are passed to `grep`. In general, $D_p \geq D_d \geq D_a$. Note that D_p is just D_d for the case without directory sets.

Experimental Study We ran several experiments with three types of keyword queries (single keyword, disjunction of keywords, conjunction of keywords) both without and with directory sets. All experiments reported in this section were run on the thinned fragment of the AT&T intranet consisting of 105,964 documents comprising about 1.5 Gigabytes of textual data, as described previously. The performance measurements given below were generated as the average of 10 runs on a 20 processor MIPS R4400 (250 MHz) running IRIX 6.2. (Only one processor was used, though, in our study.) The machine used had 3GB of 4-way interleaved RAM. The reported time is the CPU time reported by the operating system. In our experiments, we measured two types of parameters:

- The number of documents which are potential matches D_d (these are passed to `grep`), and the number of documents in the answer, D_a.
- The execution time without and with the use of directory sets to restrict the search. More precisely, the time for stage (1) of Glimpse: T_p; the time for stage (1a) (our directory set processing): T_d; and the time for Glimpse's stage (2): T_a.

We report on three representative sets of experiments.

Single Keyword: Consider a programmer, with a geographic preference for the midwest, searching for jobs as an Oracle programmer in AT&T's information technology related organizations. We search for documents

[5] The largest positive integer that can be represented using 32 bits is $2^{31} - 1$.

[6] Each block is addressable with 1 byte.

Key-words	Para-meter	Directory Sets		
		ALL	IT	IT $\subseteq$ MW
Single	D_d	462	55	25
	D_a	462	55	25
Dis-junction	D_d	10286	1525	992
	D_a	785	177	143
		ALL	Qual	Qual $\cap$ SR
Con-junction	D_d	2143	160	21
	D_a	123	27	21

Table 2: Focusing achieved with directory sets

Key-words	Para-meter	Directory Sets		
		ALL	IT	IT $\subseteq$ MW
Single	T_p	1.75	1.75	1.75
	T_d	—	0.06	0.05
	T_a	1.42	0.21	0.14
	T_{tot}	3.17	2.02	1.94
Dis-junc-tion	T_p	5.32	5.32	5.32
	T_d	—	0.78	0.76
	T_a	18.18	2.30	1.20
	T_{tot}	23.50	8.40	7.28
		ALL	Qual	Qual $\cap$ SR
Con-junc-tion	T_p	2.11	2.11	2.11
	T_d	—	0.16	0.16
	T_a	5.32	0.32	0.10
	T_{tot}	7.43	2.59	2.37

Table 3: Performance improvements with directory sets. Times are in seconds. $T_{tot} \stackrel{\text{def}}{=} T_p + T_d + T_a$.

containing the single keyword "Oracle" in (a) the entire collection of documents (ALL), (b) restricted to the predefined directory set `Information-Technology`, and (c) restricted to the dynamically defined directory set `Information-Technology` $\subseteq$ `Midwest`.

Disjunction of Keywords: For the above directory sets, we search for documents containing the disjunction of keywords "Oracle,Sybase,Informix,DB2".

Conjunction of Keywords: We consider the case of a software engineer searching for approaches and techniques to the Year 2000 problem within AT&T. We search for documents containing the conjunction of keywords "Year;2000" in (a) the entire collection of documents (ALL), (b) restricted to the `Quality` directory set, and (c) restricted to the dynamically defined directory set `Quality` $\cap$ `Software-Repositories`.

For evaluating expressions in the directory set algebra, we used the table driven algorithm described in the previous section.

Experimental Results The results are shown in Tables 2 and 3. We comment on these results next.

First, the results reveal a better search focus with directory sets as opposed to keyword search only. For example, in the first experiment ("Single"), in Table 2, the keyword search alone returns 462 documents; directory sets narrow them to 55 and 25 respectively. (The fact that $D_a = D_d$ in this experiment is accidental.) The second lesson that we learn from our case study is that *keyword-based searches can be significantly focused by using directory sets; they can be focused even further by using dynamically generated directory sets.*

Second, the results reveal a reduction in execution time. Referring again to the first experiment, in Table 3, the total execution time decreased from 3.17 to 2.02 and 1.94 seconds respectively; the decrease is more dramatic for the other experiments. When we started this work we were willing to accept a performance deterioration for the sake of better focusing. In fact, as our experiments show, even with a simple implementation like ours, the performance improves. A tighter integration of directory sets into search engines, as described in Section 3.3, could lead to even better performance.

The explanation for the improved performance lies in the parameter D_d from the previous table. Much of the time for processing a keyword query is spent in the second stage (the `grep` search, T_a) of Glimpse, and this time is dependent on the number (and size) of documents processed during this second stage. A dramatic reduction in the value of D_d due to the use of directory sets results in a similar reduction in the execution time.

To measure the overhead of invoking separately the two stages of Glimpse, we also measured T_g, the time taken by Glimpse to return the set of documents that match the keyword search pattern in the entire collection of documents. This overhead varied from about 6% for the single keyword query, to about 20% both for the conjunction and for the disjunction of keywords. Even considering this overhead, the improved performance due to the use of directory sets is substantial.

Thus, the third lesson we learn from our case study is that *the execution time for a keyword-based search is reduced when the search is focused using a directory set; the execution time is often reduced even further by using dynamically generated directory sets.* A consequence is that, in a multi-user environment, the use of directory sets in a search can also significantly increase the throughput of the search system.

5 Related Work

Alta Vista currently supports focusing keyword-based search using partial specification of the URLs of matching documents with the `url:string` operator. Only one string can be specified, and this string can match anywhere in the pathname, which results in a very different semantics from that obtained using directory sets.

While current Web guides do not allow boolean combinations of subject categories, most large online text databases, e.g., DIALOG and Lexis-Nexis, do allow queries to contain combinations of subject categories, in addition to keywords. Our approach is much more flexible in the kinds of ways subject categories (or, directory sets) can be combined.

Directory operators were used in the PAT text searching system [10] for the New Oxford English Dictionary at the University of Waterloo. Consens and Milo consider a fragment of the PAT operators for indexing text databases [3, 4], and discuss optimization techniques for the algebra; our operators in $\mathbf{H}$ are derived from those in [3, 4]. The main difference between our approach and theirs is that our domain is a hierarchy, while theirs is a flat file; hence, in our context, a directory is a sub-tree in this hierarchy (collecting everything below), while in their context it is a continuous subset of a text file, represented by two pointers in that file. The application requirements are different too: the file is given in advance, and is not expected to change (else the directories need to be recomputed), while in our application the entire "file" is either unavailable (Web), or updates and deletions are frequent (Web and UNIX).

In the Web, related work has begun to show up in clustering answers of search queries. For example, the inference find search engine at `http://m5.inference.com/ifind` claims to farm the query out to multiple search engines in parallel and clusters answers into understandable groupings. However, the grouping appears to be based on section headings, domains (commercial sites, educational sites and European/foreign sites etc.). The answers are *not* grouped according to directories that are beyond just what is present in the names of the first portion of the URL string. Any commercial site that has "`.com`" as part of its name is treated to be the same as any other commercial site; thus for example, no distinction is made between a Fortune 500 company and a startup company. Likewise, all educational institutions are treated alike.

6 Conclusions and Future Work

We addressed the dual problems of the limited effectiveness of Web search engines (e.g., Alta Vista), and the inflexibility of current Web guides (e.g., Yahoo) in locating documents of interest to a user. Our novel mechanism enables the user to augment the query keywords with a flexible specification of the directory sets that restrict the space of documents searched. Starting from a pre-defined collection of directory sets, users can dynamically create new directory sets using operators in a hierarchy-based directory set algebra. We have designed and implemented a uniform and efficient table driven evaluation of expressions in this algebra. We presented a study that demonstrates both the improved search focus (in that the use of directory sets considerably narrows the number of documents retrieved), and performance gains. This study also demonstrates that directories are useful in other hierarchically structured information spaces as well, such as the UNIX file system.

We are currently exploring more flexible mechanisms of specifying directories. In some cases, directories are more appropriately defined by a *union* of a few sub-trees, instead of a single sub-tree. For example, if two existing companies merge, the directory that contains their Web pages after the merger would be the union of the two sub-trees that defined their respective directories before the merger. Such situations currently exist among the Fortune 500 companies, for example, Philip Morris is a holding company that owns many companies each of whose directories is a separate sub-tree in the Web document space. When directories are unions of sub-trees, the operations in the directory set algebra are still meaningful and many of our results extend in a straightforward manner. The evaluation of expressions in the algebra, however, is not as efficient since the directory sets can no longer be sorted; we are examining this issue currently.

Future work includes deploying directory sets and expressions in the directory set algebra to augment keyword-based queries in real-world search engines.

Acknowledgments

The authors would like to thank David Lewis for carefully reading the paper, and for his very useful comments.

References

[1] William A. Boyajian. Personal communication. August 1997.

[2] Ashok Chandra and David Harel. Computable queries for relational databases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[3] Mariano P. Consens and Tova Milo. Optimizing queries on files. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 301–312, Minneapolis, MN, May 1994.

[4] Mariano P. Consens and Tova Milo. Algebras for querying text regions. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 11–22, San Jose, CA, May 1995.

[5] Heather Fawcett. *A Text Searching Document, PAT 3.3. User's Guide*. Center for the New Oxford English Dictionary, University of Waterloo, 1989.

[6] Guy Jacobson, Balachander Krishnamurthy, Divesh Srivastava and Dan Suciu. Focusing search in hierarchical structures with directory sets. Available from http://www.research.att.com/~divesh/papers/jkss98-ds-full.ps, 1998.

[7] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1984.

[8] Udi Manber and Sun Wu. GLIMPSE: A tool to search through entire file systems. Technical Report 34, University of Arizona, Tucson, 1993.

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

[10] A. Salminen and F. W. Tompa. PAT expressions: An algebra for text search. In *Papers in Computational Lexicography: COMPLEX'92*, pages 309–332, 1992.

[11] Howard Turtle and James Flood. Query evaluation: Strategies and optimization. *Information Processing & Management*, November 1995.