

A Security Model for Full-Text File System Search in Multi-User Environments

Stefan Büttcher and Charles L. A. Clarke

*School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada*

Abstract

Most desktop search systems maintain per-user indices to keep track of file contents. In a multi-user environment, this is not a viable solution, because the same file has to be indexed many times, once for every user that may access the file, causing both space and performance problems. Having a single system-wide index for all users, on the other hand, allows for efficient indexing but requires special security mechanisms to guarantee that the search results do not violate any file permissions.

We present a security model for full-text file system search, based on the UNIX security model, and discuss two possible implementations of the model. We show that the first implementation, based on a postprocessing approach, allows an arbitrary user to obtain information about the content of files for which he does not have read permission. The second implementation does not share this problem. We give an experimental performance evaluation for both implementations and point out query optimization opportunities for the second one.

1 Introduction and Overview

With the advent of desktop and file system search tools by Google, Microsoft, Apple, and others, efficient file system search is becoming an integral component of future operating systems. These search systems are able to deliver the response to a search query within a fraction of a second because they index the file system ahead of time and keep an index that, for every term that appears in the file system, contains a list of all files in which the term occurs and the exact positions within those files (called the term's *posting list*).

While indexing the file system has the obvious advantage that queries can be answered much faster from the index than by an exhaustive disk scan, it also has the obvious disadvantage that a full-text index requires significant disk space, sometimes more than what is avail-

able. Therefore, it is important to keep the disk space consumption of the indexing system as low as possible. In particular, for a computer system with many users, it is infeasible to have an individual index for every user in the system. In a typical UNIX environment, for example, it is not unusual that about half of the file system is readable by all users in the system. In such a situation, even a single `chmod` operation – making a previously private file readable by everybody – would trigger a large number of index update operations if per-user indices were used. Similarly, due to the lack of information sharing among the individual per-user indices, multiple copies of the index information about the same file would need to be stored on disk, leading to a disk space consumption that could easily exceed that of the original file.

We investigated different desktop search tools, by Google¹, Microsoft², Apple³, Yahoo⁴, and Copernic⁵, and found that all but Apple's Spotlight maintain a separate index for every user (Google's search tool uses a system-wide index, but this index may only be accessed by users with administrator rights, which makes the software unusable in multi-user environments). While this is an unsatisfactory solution because of the increased disk space consumption, it is very secure because all file access permissions are automatically respected. Since the indexing process has the same privileges as the user that it belongs to, security restrictions cannot be violated, and the index accurately resembles the user's view of the file system.

If a single system-wide index is used instead, this index contains information about all files in the file system. Thus, whenever the search system processes a search query, care has to be taken that the results are consistent with the user's view of the file system. A search result is obviously inconsistent with the user's view of the file system if it contains files for which the user does not have read permission. However, there are more subtle cases of inconsistency. In general, we say that the result to a search query is inconsistent with the user's view of the

file system if some aspect of it (e.g., the order in which matching files are returned) depends on the content of files that cannot be read by the user. Examples of such inconsistencies are discussed in section 5.

An obvious way to address the consistency problem is the postprocessing approach: The same, system-wide index is used for all users, and every query is processed in the same way, regardless of which user submitted the query; after the query processor has computed the list of files matching the query, all file permissions are checked, and files that may not be searched by the user are removed from the final result. This approach, which is used by Apple's Spotlight search system (see the Apple Spotlight technology brief⁶ for details), works well for Boolean queries. However, pure Boolean queries are not always appropriate. If the number of files in a file system is large, the search system has to do some sort of relevance ranking in order to present the most likely relevant files first and help the user find the information he is looking for faster. Usually, a TF/IDF-based (term frequency / inverse document frequency) algorithm is used to perform this relevance ranking.

In this paper, we present a full-text search security model. We show that, if a TF/IDF-style ranking algorithm is used by the search system, an implementation of the security model must not follow the postprocessing approach. If it does, it produces search results that are inconsistent with the user's view of the file system. The inconsistencies can be exploited by the user in a systematic way and allow him to obtain information about the content of files which he is not allowed to search. While we do not know the exact ranking algorithm employed by Apple's Spotlight, we conjecture that it is at least in parts based on the TF/IDF paradigm (as TF/IDF-based algorithms are the most popular ranking techniques in information retrieval systems) and therefore amenable to the attacks described in this paper.

After discussing possible attacks on the postprocessing approach, we present a second approach to the inconsistency problem which guarantees that all search results are consistent with the user's view of the file system and which therefore does not allow a user to infer anything about the content of files which he may not search. This safe implementation of the file system search security model is part of the Wumpus⁷ file system search engine. The system is freely available under the terms of the GNU General Public License.

In the next two sections, we give a brief overview of previous work on security issues in multi-user environments (section 2) and an introduction to basic information retrieval techniques (section 3). This introduction covers the Okapi BM25 relevance ranking function (section 3.2) and the structural query language GCL (section 3.3) on which our retrieval framework and the safe im-

plementation of the security model are based.

In section 4, we present a general file system search security model and define what it means for a file to be *searchable* by a user. Section 5 discusses the first implementation of the security model, based on the postprocessing approach described above. We show how this implementation can be exploited in order to obtain the total number of files in the file system containing a certain term. This is done by systematically creating and deleting files, submitting search queries to the search system, and looking at either the relevance scores or the relative ranks of the files returned by the search engine.

In section 6, we present a second implementation of the security model. This implementation is immune against the attacks described in section 5. Its performance is evaluated experimentally in section 7 and compared to the performance of the postprocessing approach. Opportunities for query optimization are discussed in section 8, where we show that making an almost non-restrictive assumption about the independence of different files allows us to virtually nullify the overhead of the security mechanisms in the search system.

2 Related Work

While some research has been done in the area of high-performance dynamic indexing [BCC94] [LZW04], which is also very important for file system search, the security problems associated with full-text search in a multi-user environment have not yet been studied.

In his report on the major decisions in the design of Microsoft's Tripoli search engine, Peltonen [Pel97] demands that "full text indexing must never compromise operating or file system security". However, after this initial claim, the topic is not mentioned again in his paper. Turtle and Flood [TF95] touch the topic of text retrieval in multi-user environments, but only mention the special memory requirements, not the security requirements.

Griffiths and Wade [GW76] and Fagin [Fag78] were among the first who investigated security mechanisms and access control in relational database systems (System R). Both papers study discretionary access control with ownership-based administration, in some sense similar to the UNIX file system security model [RT74] [Rit78]. However, their work goes far beyond UNIX in some aspects. For example, in their model it is possible that a user grants the right to grant rights for file (table) access to other users, which is impossible in UNIX. Bertino et al. [BJS95] give an overview of database security models and access control mechanisms, such as group authorization [WL81] and authorization revocation [BSJ97].

While our work is closely related to existing research in database security, most results are not applicable to the file system search scenario because the requirements of a relational database are different from those of a file search system and because the security model of the search system is rather strictly predetermined by the security model of the underlying file system, UNIX in our case, which does not allow most of the operations database management systems support. Furthermore, the optimizations discussed in section 8 cannot be realized in a relational database systems.

3 Information Retrieval Basics

In this section, we give an introduction to basic information retrieval techniques. We start with the index structure used by our retrieval system (inverted files), explain Okapi BM25, one of the most popular relevance ranking techniques, and then present the structural query language GCL which can be used to express queries like

Find all documents in which “mad” and “cow” occur within a distance of 3 words from each other.

We also show how BM25 and GCL can be combined in order to compute collection term statistics on the fly. We chose GCL as the underlying query language because it offers very light-weight operators to define structural query constraints.

3.1 Index Structure: Inverted Files

Inverted files are the underlying index data structure in most information retrieval systems. An inverted file contains a set of *inverted lists* (also called *posting lists*). For each term in the index, its posting list contains the exact positions of all occurrences of this term in the text collection that the index refers to.

Techniques to efficiently construct an inverted file from a text collection and to maintain it (i.e., apply updates, such as document insertions and deletions, to the index) have been discussed elsewhere [HZ03, LZW04, BC05]. What is important is that the inverted index can be used to process search queries very efficiently. For every query term, its posting list is fetched from the index, and only the query terms’ posting lists are used to process the query. At query time, it is not necessary to actually look into the files that are being searched, as all the necessary information is stored in the index.

3.2 Relevance Ranking: TF/IDF and the Okapi BM25 Scoring Function

Most relevance ranking functions used in today’s information retrieval systems are based on the vector space model and the TF/IDF scoring paradigm. Other techniques, such as latent semantic indexing [DDL⁺90] or Google’s pagerank [PBMW98], do exist, but cannot be used for file system search:

- Latent semantic indexing is appropriate for information retrieval purposes, but not for the known-item search task associated with file system search; users are searching for the exact occurrence of query terms in files, not for semantic concepts.
- Pagerank cannot be used because there are usually no cross-references between the files in a file system.

Suppose a user sends a query to the search system requesting certain pieces of data matching the query (*files* in our scenario, *documents* in traditional information retrieval). The vector space model considers the query and all documents in the text collection (files in the file system) vectors in an n -dimensional vector space, where n is the number of different terms in the search system’s vocabulary (this essentially means that all terms are considered independent). The document vectors are ranked by their similarity to the query vector.

TF/IDF (*term frequency, inverse document frequency*) is one possible way to define the similarity between a document and the search query. It means that a document is more relevant if it contains more occurrences of a query term (has greater TF value), and that a query term is more important if it occurs in fewer documents (has greater IDF value). Many different TF/IDF scoring functions exist. One of the most prominent (and most sophisticated) is Okapi BM25 [RWJ⁺94] [RWHB98].

A BM25 query is a set of (T, q_T) pairs, where T is a query term and q_T is T ’s within-query weight. For example, when a user searches for “full text search” (not as a phrase, but as 3 individual terms), this results in the BM25 query

$$Q := \{ (“full”, 1), (“text”, 1), (“search”, 1) \}.$$

Given a query Q and a document D , the document’s BM25 relevance score is:

$$s(D) = \sum_{(T, q_T) \in Q} \frac{q_T \cdot w_T \cdot d_T \cdot (1 + k_1)}{d_T + k_1 \cdot ((1 - b) + b \cdot \frac{dl}{avgdl})}, \quad (1)$$

where d_T is the number of occurrences of the term T within D , dl is the length of the document D (number of tokens), and $avgdl$ is the average document length in the system. The free parameters are usually chosen as

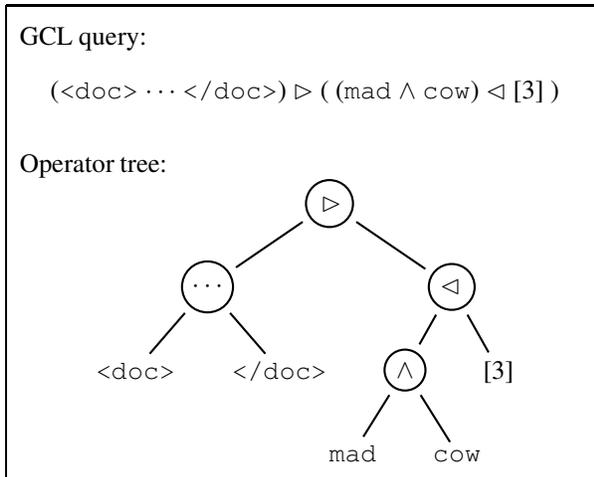


Figure 1: GCL query and resulting operator tree for the example query: *Find all documents in which “mad” and “cow” occur within a distance of 3 words from each other.* Leaf nodes in the operator tree correspond to posting lists in the index.

$k_1 = 1.2$ and $b = 0.75$. w_T is the IDF weight of the query term T :

$$w_T = \log\left(\frac{|\mathcal{D}|}{|\mathcal{D}_T|}\right), \quad (2)$$

where \mathcal{D} is the set of all documents in the text collection and \mathcal{D}_T is the set of all documents containing T . In our scenario, the documents are files, and we consequently denote \mathcal{D} as \mathcal{F} in the following sections.

From a Boolean point of view, a BM25 query is an OR query. Every document that contains at least one of the query terms matches the query. All matching documents are ranked according to their BM25 score. Roughly spoken (and therefore incorrect), this ranking makes documents containing all $|\mathcal{Q}|$ query terms appear at the top of the result list, followed by documents containing $|\mathcal{Q}| - 1$ different query terms, and so on.

3.3 Structural Queries: The GCL Query Language

The GCL (*generalized concordance lists*) query language proposed by Clarke et al. [CCB95] supports structural queries of various types. We give a brief introduction to the language because our safe implementation of the security model is based on GCL.

GCL assumes that the entire text collection (file contents) is indexed as a continuous stream of tokens. There is no explicit structure in this token stream. However, structural components, such as files or directories, can be added implicitly by inserting `<file>` and `</file>` tags (or `<dir>` and `</dir>`) into the token stream.

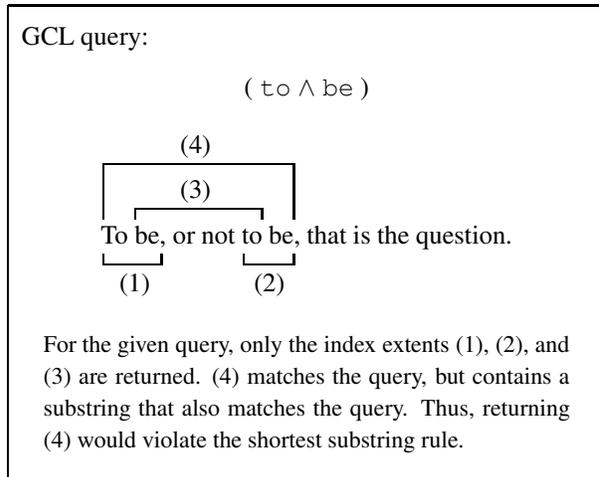


Figure 2: An example GCL query and the effect of the shortest substring rule: *Find all places where “to” and “be” co-occur.*

A GCL expression evaluates to a set of *index extents* (i.e., $[start, end]$ intervals of index positions). This is done by first producing an operator tree from the given GCL expression and then repeatedly asking the root node of the operator tree for the next matching index extent after the one that was seen last, until there are no more such extents.

GCL’s shortest substring paradigm demands that if two nested index extents satisfy a query condition, only the inner extent is part of the result set. This restriction limits the number of possible results to a query by the size of the text collection, whereas without it there could be $\binom{n}{2} \in \Theta(n^2)$ result extents for a text collection of size n . An example of how the application of the shortest substring rule affects the result to a GCL query is shown in Figure 2.

GCL operators are functions computing an output extent list from two or more input extent lists. This computation is performed on-demand in order to keep memory and CPU requirements low. The most basic type of extent lists are posting lists. As mentioned before, posting lists contain all occurrences of a given term within the indexed collection. They are found at the leaves of a GCL operator tree and can be combined using the various GCL operators.

The original GCL framework supports the following operators, which are only informally described here. Assume E is an index extent and A and B are GCL expressions. Then:

- E matches $(A \wedge B)$ if it matches both A and B ;
- E matches $(A \vee B)$ if it matches A or B ;
- E matches $(A \cdots B)$ if it has a prefix matching A and a suffix matching B ;

- E matches ($A \triangleright B$) if it matches A and contains an extent E_2 matching B ;
- E matches ($A \not\triangleright B$) if it matches A and does not contain an extent matching B ;
- E matches ($A \triangleleft B$) if it matches A and is contained in an extent E_2 matching B ;
- E matches ($A \not\triangleleft B$) if it matches A and is not contained in an extent matching B ;
- $[n]$ returns all extents of length n (i.e. all n -term sequences).

We have augmented the original GCL framework by two additional operators that let us compute collection statistics:

- $\#(A)$ returns the number of extents that match the expression A ;
- $\text{length}(A)$ returns the sum of the lengths of all extents that match A .

Throughout this paper, we use the same notation for both a GCL expression and the set of index extents that are represented by the expression.

3.4 BM25 and GCL

With the two new operators introduced in section 3.3, GCL can be employed to calculate all TF and IDF values that the query processor needs during a BM25 ranking process. Suppose the set of all documents inside the text collection is given by the GCL expression

`<doc>...</doc>`,

denoted as *documents*, and the document whose score is to be calculated is D . Then D 's relevance score is:

$$\sum_{T \in \mathcal{Q}} w_T \cdot \frac{\#(T \triangleleft D) \cdot (1 + k_1)}{\#(T \triangleleft D) + k_1 \cdot ((1 - b) + b \cdot \frac{dl}{avgdl})}, \quad (3)$$

where

$$w_T = \log \left(\frac{\#(documents)}{\#((documents) \triangleright T)} \right),$$

$$dl = \text{length}(D), \text{ and}$$

$$avgdl = \frac{\text{length}(documents)}{\#(documents)}.$$

This is very convenient because it allows us to compute all collection statistics necessary to rank the search results on the fly. Thus, integrating the necessary security restrictions into the GCL query processor in such a way that no file permissions are violated, automatically guarantees consistent search results for relevance queries. We will use this property in section 6.

	<i>owner</i>	<i>group</i>	<i>others</i>
Read	x	x	
Write	x		
eXecute	x		x

Figure 3: File permissions in UNIX (example). *Owner* permissions override *group* permissions; *group* permissions override *others*. Access is either granted or rejected explicitly (in the example, a member of the group would not be allowed to execute the file, even though everybody else is).

4 A File System Search Security Model

In section 1, we have used the term *searchable* to refer to a file whose content is accessible by a user through the search system. In this section, we give a definition of what it means for a file to be searchable by user. Before we do so, however, we have to briefly revisit the UNIX security model, on which our security model is based, and discuss the traditional UNIX search paradigm.

The UNIX security model [RT74] [Rit78] is an ownership-based model with discretionary access control that has been adopted by many operating systems. Every file is owned by a certain user. This user (the file *owner*) can associate the file with a certain *group* (a set of users) and grant access permissions to all members of that group. He can also grant access permissions to the group of all users in the system (*others*). Privileges granted to other users can be revoked by the owner later on.

Extensions to the basic UNIX security model, such as access control lists [FA88], have been implemented in various operating systems (e.g., Windows, Linux), but the simple *owner/group/others* model is still the dominant security paradigm.

UNIX file permissions can be represented as a 3×3 matrix, as shown in Figure 3. When a user wants to access a file, the operating system searches from left to right for an applicable permission set. If the user is the owner of the file, the leftmost column is taken. If the user is not the owner, but member of the group associated with the file, the second column is taken. Otherwise, the third column is taken. This can, for instance, be used to grant read access to all users in the system except for those who belong to a certain group.

File access privileges in UNIX fall into three different categories: **Read**, **Write**, and **eXecute**. Write permissions can be ignored for the purpose of this paper, which does not deal with file changes. The semantics of the read and execute privileges are different depending on

whether they are granted for a file or a directory. For files,

- the read privilege entitles a user to read the contents of a file;
- the execute privilege entitles a user to run the file as a program.

For directories,

- the read privilege allows a user to read the directory listing, which includes file names and attributes of all files and subdirectories within that directory;
- the execute privilege allows a user to access files and subdirectories within the directory.

In the traditional `find/grep` paradigm, a file can only be searched by a user if

1. the file can be found in the file system's directory tree and
2. its content may be read by the user.

In terms of file permissions, the first condition means that there has to be a path from the file system's root directory to the file in question, and the user needs to have both the read privilege and the execute privilege for every directory along this path, while the second condition requires the user to have the read privilege for the actual file in question. The same rules are used by `slocate`⁸ to decide whether a matching file may be displayed to the user or not.

While these rules seem to be appropriate in many scenarios, they have one significant shortcoming: It is not possible to grant *search* permission for a single file without revealing information about other files in the same directory. In order to make a file searchable by other users, the owner has to give them the read privilege for the file's parent directory, which reveals file names and attributes of all other files within the same directory.

A possible solution to this problem is to relax the definition of *searchable* and only insist that there is a path from the file system root to the file in question such that the user has the execution privilege for every directory along this path. Unfortunately, this conflicts with the traditional use of the read and execution privileges, in which this constellation is usually used to give read permission to all users who know the exact file name of the file in question (note that, even without read permission for a directory, a user can still access all files in it; he just cannot use `ls` to search for them). While we think this not as big a problem as the make-the-whole-directory-visible problem above, it still is somewhat unsatisfactory.

The only completely satisfying solution would be the introduction of an explicit fourth access privilege, the

search privilege, in addition to the existing three. Since this is very unlikely to happen, as it would probably break most existing UNIX software, we base our definition of *searchable* on a combination of read and execute. A file is searchable by a user if and only if

1. there is a path from the root directory to the file such that the user has the *execute* privilege for all directories along this path and
2. the user has the *read* privilege for the file.

Search permissions can be granted and revoked, just as any other permission types, by modifying the respective read and execute privileges.

While our security model is based on the simple *owner/group/other* UNIX security model, it can easily be extended to other security models, such as access control lists, as long as the set of privileges (R, W, X) stays the same, because it only requires a user to have certain privileges and does not make any assumptions about where these privileges come from.

This is our basic security model. In order to fully implement this model, a search system must not deliver query results that depend on files that are not searchable by the user who submitted the query. Two possible implementations of the model are discussed in the following sections. The first implementation does not meet this additional requirement, while the second does.

While our implementation of the security model is based on the above definition of *searchability*, the security problems we are discussing in the following sections are independent of this definition. They arise in any environment in which there are files that may not be searched by a given user.

It should be noted at this point that in most UNIX file systems the content of a file is actually associated with an i-node instead of the file itself, and there can be multiple files referring to the same i-node. This is taken into account by our search engine by assuming an i-node to be searchable if and only if there is at least one hard link to that i-node such that the above rules hold for the link.

5 A First Implementation of the Security Model and How to Exploit It: The Postprocessing Approach

One possible implementation of the security model is based on the postprocessing approach described in section 1. Whenever a query is processed, system-wide term statistics (IDF values) are used to rank all matching files by decreasing similarity to the query. This is always done in the same way, regardless of which user sent the search query. After the ranking process has finished, all files

for which the user does not have search permission (according to the rules described in the previous section) are removed from the final list of results.

Using system-wide statistics instead of user-specific data suggests itself because it allows the search system to precompute and store all IDF values, which, due to storage space requirements, is not possible for per-user IDF values in a multi-user environment. Precomputing term statistics is necessary for various query optimization techniques [WL93] [PZSD96].

In this section, we show how this approach can be exploited to calculate (or approximate) the number of files that contain a given term, even if the user sending the query does not have read permissions for those files. Depending on whether the search system returns the actual BM25 file scores or only a ranked list of files, without any scores, it is either possible to compute exact term statistics (if scores returned) or approximate them (if no scores returned).

One might argue that revealing to an unauthorized user the number of files that contain a certain term is only a minor problem. We disagree. It is a major problem, and we give two example scenarios in which the ability to infer term statistics can have disastrous effects on file system security:

- An industrial spy knows that the company he is spying on is developing a new chemical process. He starts monitoring term frequencies for certain chemical compounds that are likely to be involved in the process. After some time, this will have given him enough information to tell which chemicals are actually used in the process – without reading any files.
- The search system can be used as a covert channel to transfer information from one user account to another, circumventing security mechanisms like file access logging.

Throughout this section we assume that the number of files in the system is sufficiently large so that the addition of a single file does not modify the collection statistics significantly. This assumption is not necessary, but it simplifies the calculations.

5.1 Exploiting BM25 Relevance Scores

Suppose the search system uses a system-wide index and implements Okapi BM25 to perform relevance ranking on files matching a search query. After all files matching a user’s query have been ranked by BM25, all files that may not be searched by the user are removed from the list. The remaining files, along with their relevance scores, are presented to the user.

We will determine the total number of files in the file system containing the term T^* (the “*” is used to remind us that this is the term we are interested in). We start by computing the values of the unknown parameters $avgdl$ and $|\mathcal{F}|$ in the BM25 scoring function, as shown in equation (1). We start with $|\mathcal{F}|$, the number of files in the system. For this purpose, we generate two random terms T_2 and T_3 that do not appear in any file. We then create three files F_1 , F_2 , and F_3 :

- F_1 contains only the term T_2 ;
- F_2 consists of two occurrences of the term T_2 ;
- F_3 contains only the term T_3 .

Now, we send two queries to the search engine: $\{T_2\}$ and $\{T_3\}$. For the former, the engine returns F_1 and F_2 ; for the latter, it returns F_3 . For the scores of F_1 and F_3 , we know that

$$score(F_1) = \frac{(1 + k_1) \cdot \log(\frac{|\mathcal{F}|}{2})}{1 + k_1 \cdot ((1 - b) + \frac{b}{avgdl})} \quad (4)$$

and

$$score(F_3) = \frac{(1 + k_1) \cdot \log(\frac{|\mathcal{F}|}{1})}{1 + k_1 \cdot ((1 - b) + \frac{b}{avgdl})} \quad (5)$$

Dividing (4) by (5) results in

$$\frac{score(F_1)}{score(F_3)} = \frac{\log(\frac{|\mathcal{F}|}{2})}{\log(|\mathcal{F}|)} \quad (6)$$

and thus

$$|\mathcal{F}| = 2^{\frac{score(F_3)}{score(F_3) - score(F_1)}}. \quad (7)$$

Now that we know $|\mathcal{F}|$, we proceed and compute the only remaining unknown, $avgdl$. Using equation (5), we obtain

$$avgdl = \frac{b}{X - 1 + b}, \quad (8)$$

where

$$X = \frac{(1 + k_1) \cdot \log(|\mathcal{F}|) - score(F_3)}{score(F_3) \cdot k_1}. \quad (9)$$

Since now we know all parameters of the BM25 scoring function, we create a new file F_4 which contains the term T^* that we are interested in, and submit the query $\{T^*\}$. The search engine returns F_4 along with $score(F_4)$. This information is used to construct the equation

$$score(F_4) = \frac{(1 + k_1) \cdot \log(\frac{|\mathcal{F}|}{|\mathcal{F}_{T^*}|})}{1 + k_1 \cdot ((1 - b) + \frac{b}{avgdl})}, \quad (10)$$

in which $|\mathcal{F}_{T^*}|$ is the only unknown. We can therefore easily calculate its value and after only two queries know the number of files containing T^* :

$$|\mathcal{F}_{T^*}| = |\mathcal{F}| \cdot \left(\frac{1}{2}\right)^{\text{score}(F_4) \cdot Y}, \quad (11)$$

where

$$Y = \frac{1 + k_1 \cdot ((1 - b) + \frac{b}{\text{avgdl}})}{1 + k_1}. \quad (12)$$

To avoid small changes in \mathcal{F} and avgdl , the new file F_4 can be created before the first query is submitted to the system.

While this particular technique only works for BM25, similar methods can be used to obtain term statistics for most TF/IDF-based scoring functions.

5.2 Exploiting BM25 Ranking Results

An obvious countermeasure against this type of attack is to restrict the output a little further and not return relevance scores to the user. We now show that even if the response does not contain any relevance scores, it is still possible to compute an approximation of $|\mathcal{F}_{T^*}|$, or even its exact value, from the order in which matching files are returned by the query processor. The accuracy of the approximation obtained depends on the number of files F_{max} that a user may create. We assume $F_{max} = 2000$.

The basic observation is that most interesting terms are infrequent. This fact is used by the following strategy: After we have created a single file F_0 , containing only the term T^* , we generate a unique, random term T_2 and create 1000 files $F_1 \dots F_{1000}$, each containing the term T_2 . We then submit the query $\{T^*, T_2\}$ to the search system. Since BM25 performs a Boolean OR to determine the set of matching files, F_0 as well as $F_1 \dots F_{1000}$ match the query and are ranked according to their BM25 score. If in the response to the query the file F_0 appears before any of the other files ($F_1 \dots F_{1000}$), we know that $|\mathcal{D}_{T^*}| \leq 1000$ and can perform a binary search, varying the number of files containing T_2 , to compute the exact value of $|\mathcal{F}_{T^*}|$.

If instead F_0 appears after the other files ($F_1 \dots F_{1000}$), at least we know that $|\mathcal{F}_{T^*}| \geq 1000$. It might be that this information is enough evidence for our purpose. However, if for some reason we need a better approximation of $|\mathcal{F}_{T^*}|$, we can achieve that, too.

We first delete all files we have created so far. We then generate a second random term T_3 and create 1,000 files ($F'_1 \dots F'_{1000}$), each containing the two terms T_2 and T_3 . We generate a third random term T_4 and create 999 files ($F'_{1001} \dots F'_{1999}$) each of which contains T_4 . We finally create a last file F'_0 that contains the two terms T^* and T_4 .

After we have created all the files, we submit the query $\{T^*, T_2, T_3, T_4\}$ to the search system. The relevance scores of the files $F'_0 \dots F'_{1000}$ are:

$$\text{score}(F'_0) = C \cdot \left(\log\left(\frac{|\mathcal{F}|}{|\mathcal{F}_{T^*}|}\right) + \log\left(\frac{|\mathcal{F}|}{|\mathcal{F}_{T_4}|}\right)\right), \quad (13)$$

because F'_0 contains T^* and T_4 , and

$$\text{score}(F'_i) = C \cdot \left(\log\left(\frac{|\mathcal{F}|}{|\mathcal{F}_{T_2}|}\right) + \log\left(\frac{|\mathcal{F}|}{|\mathcal{F}_{T_3}|}\right)\right) \quad (14)$$

(for $1 \leq i \leq 1000$), because all the F'_i contain T_2 and T_3 . The constant C , which is the same for all files created, is the BM25 length normalization component for a document of length 2:

$$C = \frac{1 + k_1}{1 + k \cdot ((1 - b) + \frac{2 \cdot b}{\text{avgdl}})}. \quad (15)$$

We now subsequently delete one of the files $F'_{1001} \dots F'_{1999}$ at a time, starting with F'_{1999} , until $\text{score}(F'_0) \geq \text{score}(F'_1)$ (i.e. F'_0 appears before F'_1 in the list of matching files). Let us assume this happens after d deletions. Then we know that

$$\log\left(\frac{|\mathcal{F}|}{|\mathcal{F}_{T^*}|}\right) + \log\left(\frac{|\mathcal{F}|}{1000 - d}\right) \quad (16)$$

$$\geq 2 \cdot \log\left(\frac{|\mathcal{F}|}{1000}\right) \quad (17)$$

$$\geq \log\left(\frac{|\mathcal{F}|}{|\mathcal{F}_{T^*}|}\right) + \log\left(\frac{|\mathcal{F}|}{1000 - d + 1}\right) \quad (18)$$

and thus

$$-\log(|\mathcal{F}_{T^*}|) - \log(1000 - d) \quad (19)$$

$$\geq -2 \cdot \log(1000) \quad (20)$$

$$\geq -\log(|\mathcal{F}_{T^*}|) - \log(1000 - d + 1), \quad (21)$$

which implies

$$\frac{1000^2}{1000 - d} \geq |\mathcal{F}_{T^*}| \geq \frac{1000^2}{1000 - d + 1}. \quad (22)$$

If $|\mathcal{F}_{T^*}| = 11000$, for example, this technique would give us the following bounds:

$$10990 \leq |\mathcal{F}_{T^*}| \leq 11111.$$

The relative error here is about 0.5%. Again, binary search can be used to reduce the number of queries necessary from 1000 to around 10.

If it turns out that the approximation obtained is not good enough (e.g., if $|\mathcal{F}_{T^*}| > 40000$, we have a relative error of more than 2%), we repeat the process, this time with more than 2 terms per file. For $|\mathcal{F}_{T^*}| = 40001$ and 3 terms per file, for instance, this would give us the approximation

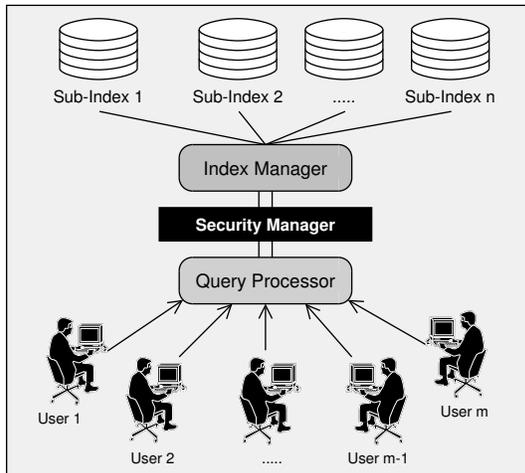


Figure 4: General layout of the Wumpus search system. The index manager maintains multiple sub-indices, one for every mount point. When the query processor requests a posting list, the index manager combines sub-lists from all indices into one large list and passes it to the security manager which applies user-specific security restrictions.

$$39556 \leq |\mathcal{F}_{T^*}| \leq 40057 \quad (\text{relative error: } 0.6\%)$$

instead of

$$40000 \leq |\mathcal{F}_{T^*}| \leq 41666 \quad (\text{relative error: } 2.1\%).$$

Thus, we have shown a way to systematically compute a very accurate approximation of the number of files in the file system containing a given term. Hence, if the postprocessing approach is taken to implement the search security model, it is possible for an arbitrary user to obtain information about the content of files for which he does not have read permission – by simply looking at the order in which files are returned by the search engine.

5.3 More Than Just Statistics

The above methods can be used to calculate the number of files that contain a particular term. While this already is undesirable, the situation is much worse if the search system allows relevance ranking for more complicated queries, such as boolean queries and phrases.

If, for instance, the search system allows phrase queries of arbitrary length, then it is possible to use the search system to obtain the whole content of a file. Assume we know that a certain file contains the phrase “*A B C*”. We then try all possible terms *D* and calculate the number of files which contain “*A B C D*” until we have found a *D* that gives a non-zero result. We then continue with the next term *E*. This way, it is possible to construct the entire content of a file using a finite number of search queries (although this might take a

long time). A simple *n*-gram language model [CGHH91] [GS95] can be used to predict the next word and thus increase the efficiency of this method significantly.

6 A Second Implementation of the Security Model: Query Integration

In this section, we describe how structured queries can be used to apply security restrictions to search results by integrating the security restrictions into the query processing instead of applying them in a postprocessing step. This implementation of the file system search security model is part of the Wumpus search system.

The general layout of the retrieval system is shown in Figure 4. A detailed description is given by [BC05]. All queries (Boolean and relevance queries) are handled by the query processor module. In the course of processing a query, it requests posting lists from the index manager. Every posting list that is sent back to the query processor has to pass the security manager, which applies user-specific restrictions to the list. As a result, the query processor only sees those parts of a posting list that lie within files that are searchable by the user who submitted the query. Since the query processor’s response is solely dependent on the posting lists it sees, the results are guaranteed to be consistent with the user’s view of the file system.

We now explain how security restrictions are applied within the security manager. In our implementation, every file in the file system is represented by an index extent satisfying the GCL expression

`<file> ... </file>`.

Whenever the search engine receives a query from a user *U*, the security manager is asked to compute a list F_U of all index extents that correspond to files whose content is searchable by *U* (using our security model’s definition of *searchable*). F_U represents the user’s view of the file system at the moment when the search engine received the query. Changes to the file system taking place while the query is being processed are ignored; the same list F_U is used to process the entire query.

While the query is being processed, every time the query processor asks for a term’s posting list (denoted as \mathcal{P}_T), the index manager generates \mathcal{P}_T and passes it to the security manager, which produces

$$\mathcal{P}_T^{(U)} \equiv (\mathcal{P}_T \triangleleft F_U),$$

the list of all occurrences of *T* within files searchable by *U*. The operator tree that results from adding these

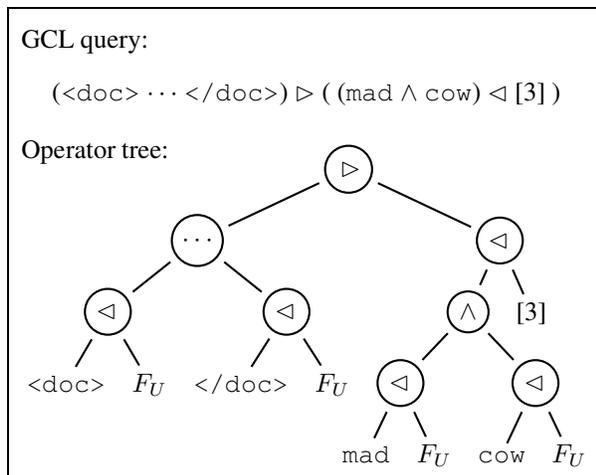


Figure 5: Integrating security restrictions into the query processing – GCL query and resulting operator tree with security restrictions applied to all posting lists (GCL containment operator “ \triangleleft ”).

security restrictions to a GCL query is shown in Figure 5. Their effect on query results is shown in Figure 6.

Since $\mathcal{P}_T^{(U)}$ is all the query processor ever sees, it is impossible for it to produce query results that depend on the content of files that are not searchable by U . Using the equations from section 3.4, all statistics necessary to perform BM25 relevance ranking can be generated from the user-specific posting lists, making it impossible to infer system-wide term statistics from the order in which matching files are returned to the user.

Because all operators in the GCL framework support lazy evaluation, it is not necessary to apply the security restrictions to the entire posting list when only a small portion of the list is used to process a query. This is important for query processing performance.

It is worth pointing out that this implementation of the security model has the nice property that it automatically supports index update operations. When a file is deleted from the file system, this file system change has to be reflected by the index immediately. Without a security model, every file deletion would either require an expensive physical update of the internal index structures, or a postprocessing step would be necessary in which all query results that refer to deleted files are removed from the final list of results [CH98]. The postprocessing approach would have the same problems as the one described in section 5: It would use term statistics that do not reflect the user’s actual view of the file system. With our implementation of the security model, file deletions are automatically supported because the

$\langle \text{file} \rangle \dots \langle / \text{file} \rangle$

extent associated with the deleted file is removed from

the security manager’s internal representation of the file system. This way, it is possible to keep the index up-to-date at minimal cost. Updates to the actual index data, which are very expensive, may be delayed and applied in batches. A more thorough discussion of index updates and their connection to security mechanisms is given by [BC05].

One drawback of our current implementation is that, in order to efficiently generate the list of index extents representing all files searchable by a given user, the security manager needs to keep some information about every indexed i-node in main memory. This information includes the i-nodes start and end address in the index address space, owner, permissions, etc. and comes to a total of 32 bytes per i-node. For file systems with a few million indexable files, this can become a problem. Keeping this information on disk, on the other hand, is not a satisfying solution, either, since it would make sub-second query times impossible. Unfortunately, we are not aware of a convincing solution to this problem.

7 Performance Evaluation

We evaluated the performance of both implementations of the security model – postprocessing and query integration – using a text collection known as TREC4+5-CR (TREC disks 4 and 5 without the Congressional Record). This collection contains 528,155 documents, which we split up into 528,155 different files in 5,282 directories. The index for this 2-GB text collection, with full positional information, requires about 615 MB, including 73 MB that are consumed by the search system’s internal representation of the directory tree, comprising file names for all files. The i-node table containing file access privileges has a total size of 16 MB and has to be kept in memory at all times to allow for fast query processing.

As query set, we used Okapi BM25 queries that were created by taking the 100 topics employed in the TREC 2003 Robust track and removing all stop words (using a moderately-sized set of 80 stop words). The original topics read like:

Identify positive accomplishments of the Hubble telescope since it was launched in 1991.

The topics were translated into queries that could be parsed and executed by our retrieval system:

```
@rank[bm25] "<doc>..</doc>" by
  "positive", "accomplishments", "hubble",
  "telescope", "since", "launched", "1991"
```

On average, a query contained 8.7 query terms, which is significantly more than the 2.2 terms found in an average web search query [JSBS98]. Nonetheless, our system

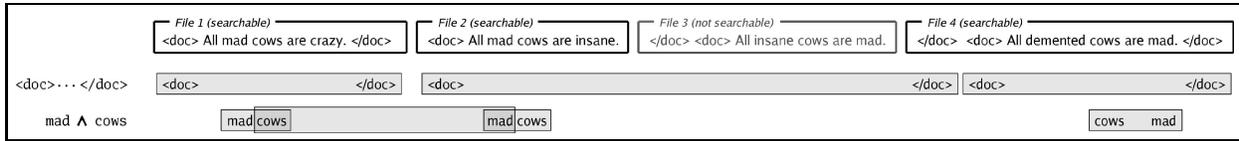
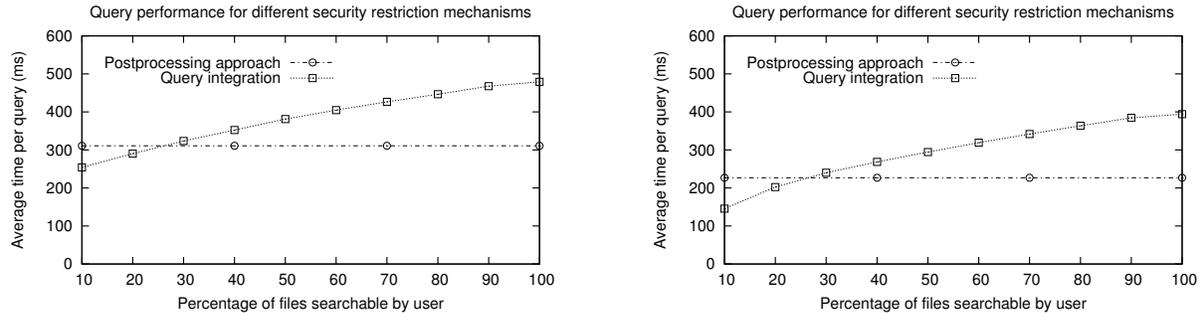


Figure 6: Query results for two example queries (“<doc>... </doc>” and “mad & cows”) with security restrictions applied. Only postings from files that are searchable by the user are considered by the query processor.



(a) Without cache effects: All posting lists have to be fetched from disk.

(b) With cache effects: All posting lists are fetched from the disk cache.

Figure 7: Performance comparison – query integration using GCL operators vs. postprocessing approach. When the number of files searchable is small, query integration is more efficient than postprocessing because relevance scores for fewer documents have to be computed.

can execute the queries in well below a second on the TREC4+5-CR text collection used in our experiments.

We ran several experiments for different percentages of searchable files in the entire collection. This was done by changing the file permissions of all files in the collection between two experiments, making a random $p\%$ readable and the other files unreadable. This way, we are able to see how the relative number of files searchable by the user submitting the search query affects the relative performance of postprocessing and query integration.

All experiments were conducted on a PC based on an AMD Athlon64 3500+ with 2 GB of main memory and a 7,200-rpm SATA hard drive.

The results depicted in Figure 7 show that the performance of the second implementation (query integration) is reasonably close to that of the postprocessing approach. Depending on whether the time that is necessary to fetch the postings for the query terms from disk is taken into account or not, the slowdown is either 54% (Figure 7(a)) or 74% (Figure 7(b)) – when 100% of the files in the index are searchable by the user submitting the query. Performance figures for both the cached and the uncached case are given because, in a realistic environment, system behavior is somewhere between these two extremes.

As the number of searchable files is decreased, query processing time drops for the query integration approach, since fewer documents have to be examined and fewer

relevance scores have to be computed, but remains constant for the postprocessing approach. This is, because in the postprocessing approach, the only part of the query processing is the postprocessing, which requires very little time compared to running BM25 on all documents. As a consequence, query integration is 18%/36% (uncached/cached) faster than postprocessing when only 10% of the files are searchable by the user who submitted the query.

8 Query Optimization

Although our GCL-based implementation of the security model does not exhibit an excessively decreased performance, it is still noticeably slower than the postprocessing approach if more than 50% of the files can be searched by the user (22-30% slowdown when 50% of the files are searchable). The slowdown is caused by applying the security restrictions ($\dots \triangleleft F_U$) not only to every query term but also to the document delimiters (<doc> and </doc>). Obviously, in order to guarantee consistent query results, it is only necessary to apply them to either the documents (in which case the query BM25 function will ignore all occurrences of query terms that lie outside searchable documents) or the query terms (in which case unsearchable documents would not contain any query terms and therefore receive

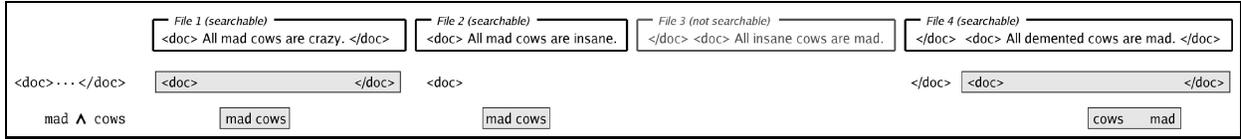


Figure 8: Query results for two example queries (“<doc> ... </doc>” and “mad ^ cows”) with revised security restrictions applied. Even though <doc> in file 2 and </doc> in file 4 are visible to the query processor, they are not considered valid query results, since they are in different files.

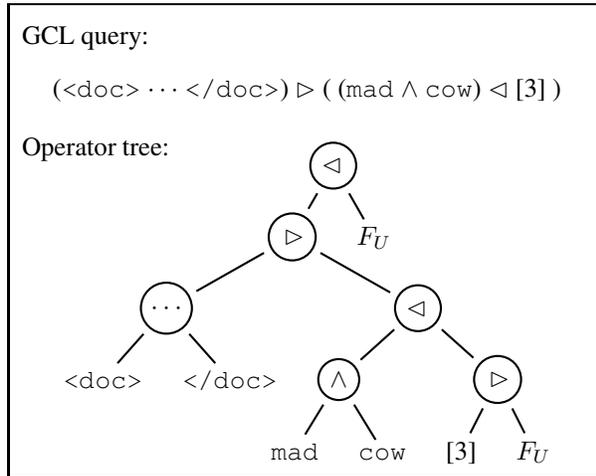


Figure 9: GCL query and resulting operator tree with optimized security restrictions applied to the operator tree.

a score of 0).

However, this optimization would be very specific to the type of the query (TF/IDF relevance ranking). More generally, we can see the equivalence of the following three GCL expressions:

$$\begin{aligned} & ((E_1 \triangleleft F_U) \triangleright (E_2 \triangleleft F_U)), \\ & ((E_1 \triangleleft F_U) \triangleright E_2), \text{ and} \\ & (E_1 \triangleright E_2) \triangleleft F_U, \end{aligned}$$

where the first expression is the result of the implementation from section 6 when run on the GCL expression

$$(E_1 \triangleright E_2).$$

The three expressions are equivalent because if an index extent E is contained in another index extent E' , and E' is contained in a searchable file, then E has to be contained in a searchable file as well.

If we make the (not very constrictive assumption) that every index extent produced by one of the GCL operators has to lie completely within a file searchable by the user that submitted the query, then we get additional equivalences:

$$((E_1 \triangleleft F_U) \wedge (E_2 \triangleleft F_U)) \equiv ((E_1 \wedge E_2) \triangleleft F_U),$$

$$\begin{aligned} ((E_1 \triangleleft F_U) \vee (E_2 \triangleleft F_U)) & \equiv ((E_1 \vee E_2) \triangleleft F_U), \\ ((E_1 \triangleleft F_U) \cdots (E_2 \triangleleft F_U)) & \equiv ((E_1 \cdots E_2) \triangleleft F_U), \end{aligned}$$

and so on. Limiting the list of extents returned by a GCL operator to those extents that lie entirely within a single file conceptually means that all files are completely independent. This is not an unrealistic assumption, since index update operations may be performed in an arbitrary order when processing events associated with changes in the file system. Thus, no ordering of the files in the index can be guaranteed, which renders extents spanning over multiple files somewhat useless. The effect that this assumption has on the query results is shown in Figure 8.

Note that if we did not make the file independence assumption, then the right-hand side of the above equivalences would be more restrictive than the left-hand side (in the case of “ \vee ”, for example, the right-hand side mandates that both extents lie within the same searchable file, whereas the left-hand side only requires that both extents lie within searchable files). If we make the assumption, then in all the cases shown above we can freely decide whether the security restrictions should be applied at the leaves of the operator tree or whether they should be moved up in the tree in order to achieve better query performance.

The only GCL operator that does not allow this type of optimization is the *contained-in* operator. The GCL expression

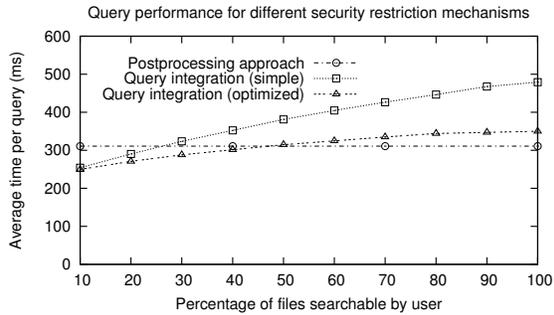
$$((E_1 \triangleleft F_U) \triangleleft (E_2 \triangleleft F_U))$$

is *not* equivalent to

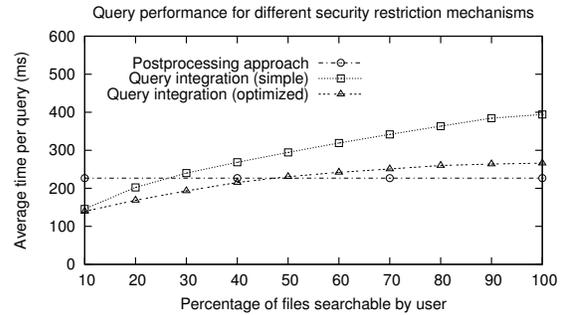
$$(E_1 \triangleleft E_2) \triangleleft F_U,$$

since in the second expression E_2 can refer to something outside the searchable files without the security restriction operator ($\triangleleft F_U$) “noticing” it. This would allow a user to infer things about terms outside the files searchable by him, so we cannot move the security restrictions up in the operator tree in this case.

At this point, it is not clear to us which operator arrangement leads to optimal query processing performance. Therefore, we follow the simple strategy of moving the security restrictions as far to the top of the operator tree as possible, as shown in Figure 9. Note that,



(a) Without cache effects: All posting lists have to be fetched from disk.



(b) With cache effects: All posting lists are fetched from the disk cache.

Figure 10: Performance comparison – query integration using GCL operators (simple and optimized) vs. postprocessing approach. Time per query for the optimized integration is between 61% and 117% compared to postprocessing.

in the figure, security restrictions are applied to the sub-expression “[3]” (all index extents of length 3), which, of course, does not make much sense, but is done by our implementation anyway.

Despite the possibility of other optimization strategies leading to better performance for certain queries, the move-to-top strategy works very well for “flat” relevance queries, such as the ones we used in our experiments:

$$\langle \text{doc} \rangle \dots \langle / \text{doc} \rangle \triangleright (T_1 \vee T_2 \vee \dots \vee T_n),$$

where the T_i are the query terms (remember that BM25 performs a Boolean OR). The performance gains caused by moving the security restrictions to the top of the tree are shown in Figure 10. With optimizations, the query integration is between 12-17% slower (100% files visible) and 20-39% faster (10% files visible) than the postprocessing approach. Even if most files in the file system are searchable by the user, this is only a minor slowdown that is probably acceptable, given the increased security.

9 Conclusion

Guided by the goal to reduce the overall index disk space consumption, we have investigated the security problems that arise if, instead of many per-user indices, a single system-wide index is used to process search queries from all users in a multi-user file system search environment.

If the same system-wide index is accessed by all users, appropriate mechanisms have to be employed in order to make sure that no search results violate any file permissions. Our full-text search security model specifies what it means for a user to have the privilege to search a file. It integrates into the UNIX security model and defines the search privilege as a combination of read and execution privileges.

For one possible implementation of the security model, based on the postprocessing approach, we have demonstrated how an arbitrary user can infer the number of files in the file system containing a given term, without having read access to these files. This represents a major security problem. The second implementation we presented, a query integration approach, does not share this problem, but may lead to a query processing slowdown of up to 75% in certain situations.

We have shown that, using appropriate query optimization techniques based on certain properties of the structural query operators in our retrieval system, this slowdown can be decreased to a point at which queries are processed between 39% faster and 17% slower by the query integration than by the postprocessing approach, depending on what percentage of the files in the file system is searchable by the user.

References

- [BC05] Stefan Büttcher and Charles L. A. Clarke. Indexing Time vs. Query Time Trade-offs in Dynamic Information Retrieval Systems. Wumpus Technical Report 2005-01. <http://www.wumpus-search.org/papers/wumpus-tr-2005-01.pdf>, July 2005.
- [BCC94] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast Incremental Indexing for Full-Text Information Retrieval. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB)*, pages 192–202, Santiago, Chile, September 1994.
- [BJS95] Elisa Bertino, Sushil Jajodia, and Pierangela Samarati. Database Security: Research and Practice. *Information Systems*, 20(7):537–556, 1995.
- [BSJ97] Elisa Bertino, Pierangela Samarati, and Sushil Jajodia. An Extended Authorization Model for Relational Databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(1):85–101, 1997.
- [CCB95] Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. An Algebra for Structured Text Search and a

- Framework for its Implementation. *The Computer Journal*, 38(1):43–56, 1995.
- [CGHH91] Kenneth Church, William Gale, Patrick Hanks, and Donald Hindle. Using Statistics in Lexical Analysis. In Uri Zernik, editor, *Lexical Acquisition: Exploiting On-Line Resources to Build a Lexicon*, pages 115–164, 1991.
- [CH98] Tzi-cker Chiueh and Lan Huang. Efficient Real-Time Index Updates in Text Retrieval Systems. Technical report, Stony Brook University, Stony Brook, New York, USA, August 1998.
- [DDL⁺90] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.
- [FA88] G. Fernandez and L. Allen. Extending the UNIX Protection Model with Access Control Lists. In *Proceedings of USENIX*, 1988.
- [Fag78] Ronald Fagin. On an Authorization Mechanism. *ACM Transactions on Database Systems*, 3(3):310–319, 1978.
- [GS95] William Gale and Geoffrey Sampson. Good-Turing Frequency Estimation Without Tears. *Journal of Quantitative Linguistics*, 2:217–237, 1995.
- [GW76] Patricia P. Griffiths and Bradford W. Wade. An Authorization Mechanism for a Relational Database System. *ACM Transactions on Database Systems*, 1(3):242–255, 1976.
- [HZ03] Steffen Heinz and Justin Zobel. Efficient Single-Pass Index Construction for Text Databases. *Journal of the American Society for Information Science and Technology*, 54(8):713–729, 2003.
- [JSBS98] Bernard J. Jansen, Amanda Spink, Judy Bateman, and Tefko Saracevic. Real Life Information Retrieval: A Study of User Queries on the Web. *SIGIR Forum*, 32(1):5–17, 1998.
- [LZW04] Nicholas Lester, Justin Zobel, and Hugh E. Williams. In-Place versus Re-Build versus Re-Merge: Index Maintenance Strategies for Text Retrieval Systems. In *Proceedings of the 27th Conference on Australasian Comp. Sci.*, pages 15–23. Australian Computer Society, Inc., 2004.
- [PBMW98] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford Digital Library Technologies Project, 1998.
- [Pel97] Kyle Peltonen. Adding Full Text Indexing to the Operating System. In *Proceedings of the Thirteenth International Conference on Data Engineering (ICDE 1997)*, pages 386–390. IEEE Computer Society, 1997.
- [PZSD96] Michael Persin, Justin Zobel, and Ron Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, October 1996.
- [Rit78] Dennis M. Ritchie. The UNIX Time-Sharing System: A Retrospective. *Bell Systems Technical Journal*, 57(6):1947–1969, 1978.
- [RT74] Dennis M. Ritchie and Ken Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, 1974.
- [RWHB98] Stephen E. Robertson, Steve Walker, and Micheline Hancock-Beaulieu. Okapi at TREC-7. In *Proceedings of the Seventh Text REtrieval Conference (TREC 1998)*, November 1998.
- [RWJ⁺94] Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. Okapi at TREC-3. In *Proceedings of the Third Text REtrieval Conference (TREC 1994)*, November 1994.
- [TF95] Howard Turtle and James Flood. Query Evaluation: Strategies and Optimization. *Information Processing & Management*, 31(1):831–850, November 1995.
- [WL81] Paul F. Wilms and Bruce G. Lindsay. A Database Authorization Mechanism Supporting Individual and Group Authorization. In *Distributed Data Sharing Systems*, pages 273–292, 1981.
- [WL93] Wai Y. P. Wong and Dik L. Lee. Implementations of Partial Document Ranking Using Inverted Files. *Information Processing & Management*, 29(5):647–669, 1993.

Notes

¹<http://desktop.google.com/>

²<http://toolbar.msn.com/>

³<http://www.apple.com/macosx/features/spotlight/>

⁴<http://desktop.yahoo.com/>

⁵<http://www.copernic.com/>

⁶http://images.apple.com/macosx/pdf/MacOSX_Spotlight_TB.pdf

⁷<http://www.wumpus-search.org/>

⁸<http://www.geekreview.org/slocate/>