

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

**Stefan Büttcher** Charles L. A. Clarke



University of Waterloo, Canada

December 15, 2005

- 1 Introduction and Motivation
- 2 Information Retrieval Basics
- 3 A File System Search Security Model
- 4 Implementation I: Postprocessing
- 5 Implementation II: Query Integration
- 6 Summary

# Introduction and Motivation

## Desktop search

Virtually all desktop search systems (Copernic, Google, MSN, Yahoo, ...) maintain one index per user.

- **Security perspective:** Per-user indexing processes run with user privileges. Very secure.
- **Efficiency perspective:** Files that can be read by  $n$  users are indexed  $n$  times. Waste of space and time.

Efficiency considerations force us to use one single index that is accessed by all users in the system.

**But then make sure that no file permissions are violated!**

# The Wumpus Search Engine

## What is Wumpus?

- A multi-user file system search engine.
- A multi-purpose information retrieval system.

## What are the main features?

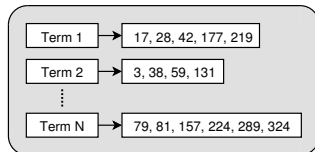
- Multi-user support based on the UNIX security model: Each user can only search files for which she has permission.
- Fully dynamic: File changes, creations, and deletions are immediately reflected by the index and the search results.

**Wumpus is Free Software according to the GPL.**

<http://www.wumpus-search.org/>

## Index Structure: Inverted Files

An **inverted file** is a dictionary data structure that tells us **for every term** in the text collection (here: file system) the exact location of **all its occurrences**.



Building an inverted file (mathematically): Transposing the term/document matrix. Can be done very efficiently.

All information necessary to process a search query can be found inside the inverted file. The actual files being searched are only accessed at the very end, to produce snippet summaries etc.

# The Vector Space Model and TF/IDF (1)

Boolean searches (AND, OR) are easy to implement and very helpful. But only if the number of matching documents is small.

For large text collections (e.g., file systems): Relevance ranking!

Three main assumptions behind relevance ranking:

- All query terms are independent (**vector space** assumption).
- A file is more likely to be relevant if has more occurrences of query terms (**term frequency** assumption).
- A query term that appears in many documents is less important than a query term that appears in few documents (**inverse document frequency** assumption).

## The Vector Space Model and TF/IDF (2)

### State-of-the-art relevance ranking: Okapi BM25

Given a document collection  $\mathcal{D}$ , a query  $Q := \{T_1, \dots, T_n\}$ , and a document  $D$ . Then

$$S_{\text{BM25}}(D) = \sum_{T \in Q} \log \left( \frac{|\mathcal{D}|}{|\mathcal{D}_T|} \right) \cdot \frac{f_{T,D} \cdot (k_1 + 1)}{f_{T,D} + k_1 \cdot (1 - b + b \cdot \frac{dl}{\text{avgdl}})},$$

where  $\mathcal{D}_T$  is the set of documents containing  $T$ , and  $f_{T,D}$  is the number of occurrences of  $T$  within  $D$ .

BM25 can be thought of as a Boolean OR. It finds all documents containing at least one query term and ranks all matching documents by their BM25 score.

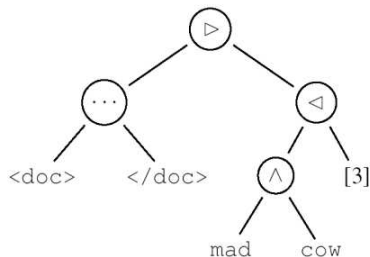
## Structural Queries: The GCL Query Language

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

**GCL query:**

$(\langle \text{doc} \rangle \dots \langle / \text{doc} \rangle) \triangleright ((\text{mad} \wedge \text{cow}) \triangleleft [3])$

**GCL operator tree:**



## Combining GCL and TF/IDF Ranking

Using GCL to compute BM25 relevance scores:

- Assume the search query contains the three terms  $T_1$ ,  $T_2$ ,  $T_3$ .
- Find all documents matching the expression:

$$(\langle \text{doc} \rangle \cdots \langle / \text{doc} \rangle) \triangleright (T_1 \vee T_2 \vee T_3)$$

- Rank all matching documents using IDF weights:

$$w_{T_i} = \log \left( \frac{\#(\langle \text{doc} \rangle \cdots \langle / \text{doc} \rangle)}{\#(\langle \text{doc} \rangle \cdots \langle / \text{doc} \rangle \triangleright T_i)} \right)$$

Other components of BM25 can be expressed in a similar way.

# The UNIX Security Model

The UNIX security model knows 3 different types of file access privileges:

	<i>owner</i>	<i>group</i>	<i>others</i>
<b>R</b> ead	x	x	
<b>W</b> rite	x		
<b>eX</b> ecute	x		x

*R, W, X, ... But where is the search bit???*

⇒ Need to work with implicit search permissions. Define the *search* privilege as a combination of *read* and *execute*.

## The find/grep/slocate Security Model

In the traditional `find/grep` UNIX search environment, a user can search a file if (and only if) she has

- $R$  and  $X$  permission for a path leading to the parent directory (in order to `find` the file) and
- $R$  permission for the actual file (in order to `grep` it).

The same rules are used by `slocate` when it tries to find out what search results it may return to the user.

**Good:** It is possible to grant file access to a specific group of users by making the dir  $X$  only (not  $R$ ) and telling them the file name.

**Bad:** To make a file `find/grep`-searchable, the entire directory has to be made readable, revealing information about other files.

# The Wumpus Security Model

In the security model used by Wumpus, a user can search a file if (and only if) she has

- $X$  permission for a path leading to the parent directory and
- $R$  permission for the actual file.

**Good:** Individual files can be made searchable without revealing any information about other files in the same directory.

**Bad:** Inconsistent with the existing search infrastructure and possible counter-intuitive.

Neither definition of *searchable* is completely satisfactory. Maybe we really need an explicit *search* bit.

# Summary

## Where are we?

- There are different ways to define what it means for a file to be *searchable* by a user.
- Regardless of the actual definition, the result is a user-specific partition of the set of all files in the file system into 2 subsets:  $F_U^\oplus$  (files that may be searched by user  $U$ ) and  $F_U^\ominus$  (files that may not be searched by user  $U$ ).
- We need to make sure that the response to a search query by user  $U$  does not reveal any information about files in  $F_U^\ominus$ .
- In particular: Search results must not contain files from  $F_U^\ominus$ .

# Implementation I: The Postprocessing Approach

The most obvious way to address the security requirements is based on a postprocessing approach:

- Find all files matching the query and rank them using global, user-independent term weights (IDF values).
- Before the search results are returned to the user, remove all files for which the user does not have search permission.

**Rationale:** In a system with a large number of users, it is infeasible to precompute per-user term weights.

*This implementation is actually part of several multi-user search systems. Its problems are real problems.*

## How to Exploit the Postprocessing Approach (1)

**Target:** Analyze the results to a search query in order to determine the number of files  $|\mathcal{F}_T|$  containing a given term  $T$ .

Is this a real problem? Yes. And it does not stop with terms. Boolean queries, structural queries, phrases, ...

How to proceed?

- If the search engine returns relevance scores along with the matching files, it is straightforward to compute  $|\mathcal{F}_T|$  (see paper for details).
- Make it more challenging: The search engine does not return relevance scores. Use relative file ranks to approximate  $|\mathcal{F}_T|$ .

## How to Exploit the Postprocessing Approach (2)

Assume we want to approximate  $|\mathcal{F}_T|$ .

- Create single file  $F_0$  containing only the term  $T$ .
- Generate unique, random term  $T_2$ .
- Create 1000 files  $F_1 \dots F_{1000}$ , each containing the term  $T_2$ .
- Submit search query  $\{T, T_2\}$ .
- Since BM25 performs OR, all files  $F_0 \dots F_{1000}$  match the query and are ranked according to their BM25 score.
- If  $F_0$  appears before any of the other files ( $F_1 \dots F_{1000}$ ), we know that  $|\mathcal{F}_T| \leq 1000 = |\mathcal{F}_{T_2}|$ . Exact value: *Binary search!*
- ... but what if not???

## How to Exploit the Postprocessing Approach (3)

If  $F_0$  appears after the  $F_1 \dots F_{1000}$ , then:

- Delete all files created so far.
- Generate two more random terms  $T_3$  and  $T_4$ .
- Create 1000 files  $\bar{F}_1 \dots \bar{F}_{1000}$  containing  $T_2$  and  $T_3$ .
- Create 999 files  $\hat{F}_1 \dots \hat{F}_{999}$ , each containing only  $T_4$ .
- Create one last file  $\hat{F}_{1000}$ , containing  $T$  and  $T_4$ .
- Send query  $\{T, T_2, T_3, T_4\}$  to the search systems.
- All 2,000 files match the query. But what is the ranking?  
What are the scores?

## How to Exploit the Postprocessing Approach (4)

Remember the BM25 score of a file  $F$ :

$$S_{\text{BM25}}(F) = \sum_{T \in Q} \log \left( \frac{|\mathcal{F}|}{|\mathcal{F}_T|} \right) \cdot \frac{f_{T,F} \cdot (k_1 + 1)}{f_{T,F} + k_1 \cdot (1 - b + b \cdot \frac{dl}{\text{avgdl}})}.$$

Thus, we have:

$$S_{\text{BM25}}(\hat{F}_{1000}) = C \cdot \left( \log \frac{|\mathcal{F}|}{|\mathcal{F}_T|} + \log \frac{|\mathcal{F}|}{|\mathcal{F}_{T_4}|} \right) \quad \text{and}$$
$$S_{\text{BM25}}(\hat{F}_i) = C \cdot \left( \log \frac{|\mathcal{F}|}{|\mathcal{F}_{T_2}|} + \log \frac{|\mathcal{F}|}{|\mathcal{F}_{T_3}|} \right) \quad \text{for } 1 \leq i \leq 999,$$

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

## How to Exploit the Postprocessing Approach (5)

BM25 score components:



Document frequencies:

- $|\mathcal{F}_{T_2}| = |\mathcal{F}_{T_3}| = 1000$
- $|\mathcal{F}_{T_4}| = 1000$
- $|\mathcal{F}_T| \geq 1000$

## How to Exploit the Postprocessing Approach (5)

**BM25 score components:**



**Document frequencies:**

- $|\mathcal{F}_{T_2}| = |\mathcal{F}_{T_3}| = 1000$
- $|\mathcal{F}_{T_4}| = 1000 \dots$  decrease!
- $|\mathcal{F}_T| \geq 1000$

## How to Exploit the Postprocessing Approach (5)

BM25 score components:



Document frequencies:

- $|\mathcal{F}_{T_2}| = |\mathcal{F}_{T_3}| = 1000$
- $|\mathcal{F}_{T_4}| = 1000 \dots \text{decrease!} \dots \text{decrease!}$
- $|\mathcal{F}_T| \geq 1000$

## How to Exploit the Postprocessing Approach (5)

### BM25 score components:



### Document frequencies:

- $|\mathcal{F}_{T_2}| = |\mathcal{F}_{T_3}| = 1000$
- $|\mathcal{F}_{T_4}| = 1000 \dots \text{decrease!} \dots \text{decrease!} \dots \text{decrease!}$
- $|\mathcal{F}_T| \geq 1000 \dots \log \frac{|\mathcal{F}|}{|\mathcal{F}_T|} + \log \frac{|\mathcal{F}|}{|\mathcal{F}_{T_4}|} \approx \log \frac{|\mathcal{F}|}{|\mathcal{F}_{T_2}|} + \log \frac{|\mathcal{F}|}{|\mathcal{F}_{T_3}|}$

## How to Exploit the Postprocessing Approach (6)

### Example

Assume  $T$  appears in  $|\mathcal{F}_T| = 11,000$  files.

Using the technique just described, we obtain

$$\log \frac{|\mathcal{F}|}{|\mathcal{F}_T|} + \log \frac{|\mathcal{F}|}{90} \geq 2 \cdot \log \frac{|\mathcal{F}|}{1000} \geq \log \frac{|\mathcal{F}|}{|\mathcal{F}_T|} + \log \frac{|\mathcal{F}|}{91},$$

which yields

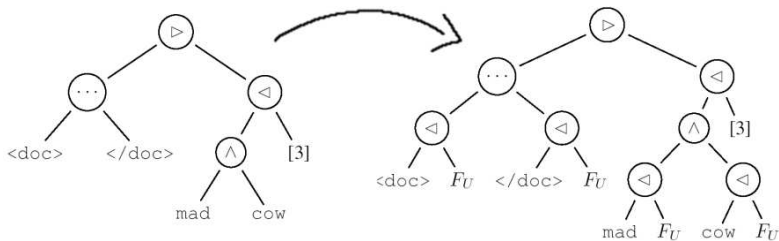
$$10990 \leq |\mathcal{F}_T| \leq 11111.$$

The relative error is  $\varepsilon = 0.5\%$ .

## Implementation II: Query Integration (1)

We have seen how to use GCL to perform relevance ranking. Now: Integrate security restrictions into GCL.

At query time, construct a GC-list  $F_U$  that represents all files searchable by user  $U$ . Then apply restrictions:



## Implementation II: Query Integration (2)

Other components of BM25 ranking function (term weights, average document length, etc.) are modified in a similar way.

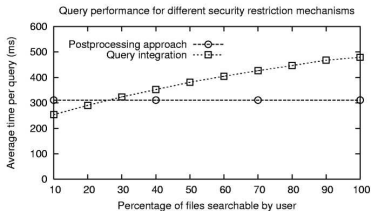
**Unsecure term weights:**

$$w_{T_i} = \log \left( \frac{\#(\langle \text{doc} \rangle \cdots \langle / \text{doc} \rangle)}{\#(\langle \langle \text{doc} \rangle \cdots \langle / \text{doc} \rangle \triangleright T_i)} \right)$$

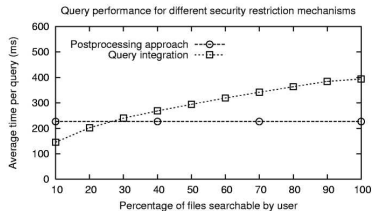
**Secure term weights:**

$$w_{T_i} = \log \left( \frac{\#(\langle \langle \text{doc} \rangle \triangleleft F_U \rangle \cdots \langle / \text{doc} \rangle \triangleleft F_U)}{\#(\langle \langle \langle \text{doc} \rangle \triangleleft F_U \rangle \cdots \langle / \text{doc} \rangle \triangleleft F_U \rangle \triangleright (T_i \triangleleft F_U))} \right)$$

# Experimental Results (1)



(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.

Compared to postprocessing, the unoptimized query integration is between 54-74% slower and 18-36% faster (depending on cache effects and relative number of visible files).

# Query Optimization

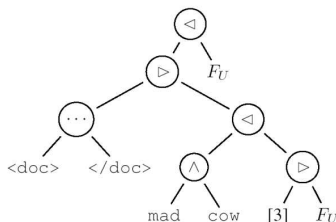
Consider the GCL expression:

$$((\langle \text{doc} \rangle \triangleleft F_U) \cdots (\langle / \text{doc} \rangle \triangleleft F_U)) \triangleleft F_U.$$

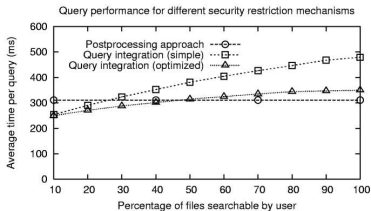
This is equivalent to:

$$(\langle \text{doc} \rangle \cdots \langle / \text{doc} \rangle) \triangleleft F_U.$$

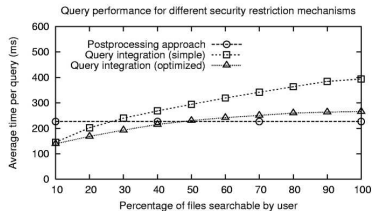
Analogous equivalences exist for the other GCL operators:  $\wedge$ ,  $\vee$ , ...



## Experimental Results (2)



(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.

Compared to postprocessing, the optimized query integration is between 12-17% slower and 20-39% faster (depending on cache effects and relative number of visible files).

# Summary

- We have proposed a file system search security model based on the UNIX security model.
- For one possible implementation (postprocessing), we have shown that an arbitrary user may obtain information about files for which she does not have read permission.
- We have presented a second, secure implementation and pointed out query optimization opportunities.
- Using the optimized implementation of the security model, average time per query is between 61% and 117% of the original, security-unaware implementation.

The End

# Thank You!