Efficient and Secure Search of Enterprise File Systems

Aameek Singh
Georgia Tech
aameek@cc.gatech.edu

Mudhakar Srivatsa
Georgia Tech
mudhakar@cc.gatech.edu

Ling Liu
Georgia Tech
lingliu@cc.gatech.edu

ABSTRACT

With fast paced growth of digital data, keyword based search has become a critical enterprise application. Research has shown that nearly 85% of enterprise data lies in flat filesystems [6] that allow multiple users and user groups with different access privileges. Any search tool for such systems needs to be efficient and yet cognizant of the access control semantics imposed by the underlying filesystem. Current multiuser enterprise search techniques use two disjoint *search* and *access-control* components by creating a single system-wide index and simply filtering search results for access control. This approach is ineffective as the index and query statistics subtly leak private information. The other available approach of using separate indices for each user is undesirable as it not only increases disk consumption due to shared files, but also increases the overheads of updating the indices whenever a file changes.

We propose a distributed approach that couples search and access-control into a unified framework and provides secure multiuser search. Our scheme (logically) divides data into independent access-privileges based chunks, called access-control barrels (ACB). ACBs not only manage security but also improve overall efficiency as they can be indexed and searched in parallel by distributing them to multiple enterprise machines. We describe the architecture of ACBs based search framework and propose two optimization technique that ensure the scalability of our approach. We validate our approach with a detailed evaluation using industry benchmarks and real datasets. Our initial experiments show secure search with 38% improved indexing efficiency and low overheads for ACB processing.

1. INTRODUCTION

In recent years, the total amount of digital data has grown leaps and bounds, doubling almost every eighteen months [7]. As the cost of storage hardware drops, enterprises are storing more data and also keeping it for a longer period of time. It is for both business intelligence as well as regulatory compliance purposes. With this large amount of data available, keyword based search to quickly locate relevant data has become an essential IT capability.

According to many estimates, as much as 85% of enterprise data is stored in unstructured repositories like enterprise and local filesystems [6]. These filesystems have multiple users with different privileges to data and access is controlled using native access control mechanisms like Unix/Windows permissions models. This access control needs to be enforced even while searching through the data. As a simple case in point, a user should not be able to search through data that is not accessible to that user. Also, there are additional subtle requirements that complicate this process and if unhandled, can result in information leaks. For example, looking at the results of a query a user should not be able to extract any information that could not have been inferred by that user by accessing the underlying filesystem. We refer to this principle as *Access Control Aware Search* or ACAS in short. Simply put, ACAS requires that no additional information can be extracted about the filesystem by using the search mechanism. More formally we define it as:

Definition: ***Access Control Aware Search (ACAS)*** *Let $\mathcal{I}_F^U$ be the information that a user U can extract from the filesystem F by accessing it directly (dictated by access rights for U) and let $\mathcal{I}_S^U$ be the information that U can extract by searching on the indices over F over any period of time (based on the search mechanism). The access control aware search (ACAS) property requires that $\mathcal{I}_S^U \subseteq \mathcal{I}_F^U$.*

Surprisingly most enterprise search products in the market, like Google Enterprise [20], IBM OmniFind [10], [5], do not satisfy the ACAS principle. These tools treat search and access-control as two disjoint components and can result in malicious users extracting unauthorized information using the search mechanism. In their approach, a single system wide index is created for all users and it is queried using traditional information retrieval (IR) techniques (the *search* component). Finally the results (the list of files containing query keywords) are filtered based on access privileges for the querying user (*access control*). However, the ordering and relevance score of results, based on Term-Frequency-Inverse-Document-Frequency (TFIDF) measures [21], reveal information that violates the ACAS property. Intuitively, since the index was created based on the lexicon and documents of the complete system, simple post-processing of results would fail to adequately protect system-wide statistics against carefully crafted attacks. We describe this issue and demonstrate an example attack in §2.2.

A technique that satisfies ACAS can be found in common desktop search products like Google Desktop [3] and Yahoo Desktop [4]. For multiple users on the desktop, these tools create distinct indices for each user on the system, with each user index including all files accessible to that user (the *access-control* component) and then querying only that in-

WWW2007, May 8–12, 2007, Banff, Canada.
.

dex for the user (the *search* component). While this satisfies ACAS, it is highly inefficient as it requires every shared file to be indexed multiple times in the indices of each user that can access that file. Since in modern enterprises, a large amount of data is shared by many users, this approach not only causes greater disk consumption (due to increased index size), but the overheads of updating the indices when a file changes also become significant.

In this paper, we propose a distributed enterprise search technique that couples search and access-control into a unified framework to provide secure and efficient search. We use a novel building block called access control barrel (ACB) that ensures access control aware search. An ACB is a set of files that have the same access privileges for users and groups in the system and by dividing filesystem data into independent ACBs, we can ensure that the index for a user is only derived from data accessible to that user in the underlying filesystem, thus satisfying the ACAS requirement.

The ACB-based approach is also space and update efficient as it ensures that each file is included in only a single index. This *minimality* property makes it especially suitable for shared multiuser environments. Further, by dividing data into independent barrels, data indexing can be distributed to multiple machines for parallel processing. This can significantly reduce total indexing time. We also describe two optimization techniques that ensure the scalability of our approach even in complex enterprise environments.

In summary this paper makes the following contributions:

- **Access Control Aware Enterprise Search**: In this paper, we have characterized the access control problem in enterprise search. We also propose a new Access Control Barrel (ACB) concept that prevents information leaks to unauthorized users.
- **Space and Update Efficiency**: By ensuring that a file is included in a single index, our approach provides superior space and update efficiency. Also, our proposed optimizations provide a mechanism to ensure scalability of our approach even in complex settings.
- **Distributed Enterprise Search Architecture**: In contrast to existing centralized approaches, we have developed a distributed architecture that parallelizes indexing and search for better performance and is thus better suited to modern enterprises with numerous underutilized machines.

Next we describe relevant background and discuss current search approaches and their limitations.

2. BACKGROUND AND RELATED WORK

In this section, we briefly describe modern enterprise architecture, that serves as the model environment assumed in this paper. Later, we will discuss related work and limitations of existing enterprise search techniques.

2.1 Modern Enterprise Architecture

Modern enterprises today are data-rich environments with storage capacities running into terabytes and petabytes. A typical enterprise has multiple file servers, most commonly accessed via *nix based Network File System (NFS) or Windows based Common Internet File System (CIFS) protocols. These file servers could be directly attached to storage (Network Attached Storage, NAS architecture as shown in Figure-1) or through a Storage Area Network (SAN). In the context of our work, we only require a common global namespace which is widely supported in both architectures.

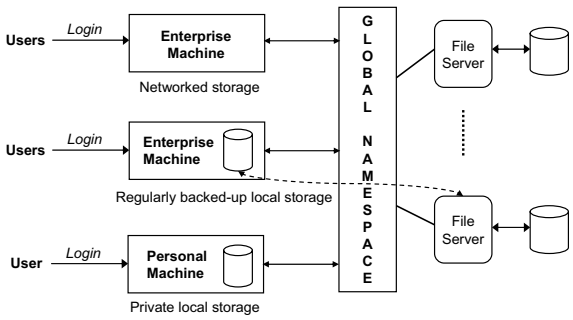

Figure 1: Enterprise Storage Architecture

On the client side, this namespace is accessed by many always-connected (and mostly wired) enterprise machines. These could be user workstations, shared laboratory machines or application servers. Most of these machines have networked storage (for example, NFS mounted). Some of these machines might also have some local storage that is regularly backed up to global file servers. We call such machines *enterprise machines* since they are under administrative control and can be potentially leveraged for indexing and search tasks. Other prominent client category consists of single-user personal machines like laptops. *Laptops* are notorious for hiding data from administrative control with their intermittent connectivity and frequent suspend-resume. We call such clients *personal machines* and consider them to be unavailable for administrative usage.

2.1.1 Access Control

In an enterprise environment, not all data is accessible to all users. Access to data is controlled through the underlying filesystem's access control mechanisms. For example, NFS file servers follow the UNIX permissions model (described below) and Windows-based file servers follow the Windows permissions model. In this work, because of its widespread deployment and easier availability for experimentation, we use the UNIX based NFS architecture (also supported by other *nix flavors like Linux, FreeBSD).

In UNIX access control model [17] each filesystem object (file, directory) has an associated owner who controls the access to that object. This access can be granted to three classes of users: (1) *owner*, (2) *group*, and (3) *others*. The *owner* is the object owner, the *group* is the user group the owner belongs to (for example, a user group for students, faculty) and *others* are all users except the owner and the group. Permissions for multiple users and multiple groups can be set using POSIX Access Control Lists (ACLs) [8]. Further, the granted access can be of three types – Read (r), Write (w) and Execute (x). We refer the reader to [17] for detailed UNIX permissions model discussion. In context of indexing and search, we use the following notion of **searchability** consistent with the *nix `find/slocate` [13] paradigms:

Definition: ***Searchability*** *– A file, F is* ***searchable*** *by user u_i (or group g_m) if there exist* read *and* execute *permissions on the path leading to F and* read *permissions on F, for u_i (or g_m).*

2.2 Limitations of Existing Approaches

In this section, we explore related keyword search solutions and analyze their pros and cons from security and performance perspectives.

Most desktop search products like Google Desktop [3], Yahoo Desktop [4] integrate access control during indexing. Each user has a separate index for accessible files with duplicates for files that are shared with other users. This ensures that each user has an index created from data that was accessible to that user, satisfying the ACAS requirement. In addition, there are no additional costs at query runtime for access control based filtering. However, this causes *additional* disk consumption of $\sum(n_i - 1) * I_i$ where n_i is the number of users accessing file F_i and I_i is size of index for F_i. Additionally, each update to F_i causes updates to n_i indices. In an enterprise, where n_i could be in hundreds or thousands, such costs can be prohibitive. We provide a detailed analysis of this approach in §5.4.

The enterprise search products like Google Enterprise [20], Coveo Enterprise [19] and others [10, 5] integrate access control at query runtime by creating a single system-wide index and filtering results based on access privileges of the querying user. This provides maximum space and update efficiency. However, querying will be more expensive, especially when access permissions for files in the query results are obtained at runtime, which requires disk I/O for inode lookups. Also importantly, these products do not satisfy the ACAS requirement and by carefully crafting queries, a user can obtain information about the underlying filesystem which could not have been inferred otherwise. Below, we describe an example attack that can determine the total number of files containing a particular keyword even when the attacker does not have access to all files containing that keyword. For example, an attacker could monitor the enterprise filesystem to see the number of files containing the word "*bankruptcy*". A sudden increase in the number of such files could alert him/her to sell off company stock, practically amounting to insider trading. This violates the ACAS property, as this information could not have been determined by the attacker through the underlying filesystem directly.

We assume that the relevance score of a result file f_i is computed by the standard TFIDF measure

$$rel(f_i) = \sum_{t_j \in Q} w_{ij} * w_{Qj} \qquad (1)$$

where t_j are the terms in the query Q and w_{ij} is the normalized weight of term t_j in f_i given by

$$w_{ij} = \frac{o_{ij} * log\left(\frac{|N|}{n_{t_j}}\right)}{\sqrt{\sum_{t_k \in f_i}(o_{ik})^2 * \left(log\left(\frac{|N|}{n_{t_k}}\right)\right)^2}} \qquad (2)$$

where o_{ij} is the number of occurrences of term t_j in f_i, $|N|$ is the total number of files in the system and n_{t_j} is the number of files that contain t_j. w_{Qj} is defined similarly.

The attacker, Alice, wishes to know the number of documents that contain the term t_q (e.g. "*bankruptcy*"). The attack works in three steps. First, Alice picks two *unique* terms t_1, t_2 (no file contains these terms) and creates two new files: f_1 containing terms $\{t_1, t_2\}$ and f_2 containing terms $\{t_2, t_q\}$. Note that after creating the files, o_{11}=o_{12}=o_{22}=o_{2q}=1, n_{t_1}=1 and n_{t_2}=2. In the second step, she queries for term t_1 and from (1) and (2) she can calculate $|N|$

$$|N| = 2^{\frac{1}{1-\sqrt{\frac{1}{rel(f_1)^2}-1}}} \qquad (3)$$

In the final step, Alice queries for term t_q and calculates n_{t_q} from (1), (2), (3). This completes the attack.

$$n_{t_q} = 2^{\frac{1}{\sqrt{\frac{1}{rel(f_2)^2}-1}}} * |N|^{\left(1-\frac{1}{\sqrt{\frac{1}{rel(f_2)^2}-1}}\right)}$$

Such attacks are possible on most TFIDF based measures including the popular Okapi BM25 [18]. Additionally, even when relevance scores are not returned as part of the result, good approximations to n_{t_q} can be obtained by exploiting ordering of the results [2]. A recent effort [2] describes an ACAS compliant approach that uses a complex query transformation at runtime for access control. Additionally, it maintains access control lists for all files in the filesystem in-memory which is extremely inefficient for large enterprise environments.

In contrast to these centralized approaches, we use a distributed architecture using a novel access control barrel primitive. Our technique leverages existing work in distributed information retrieval. Kretser et al [12] have compared different approaches of distributed retrieval and concluded that distributed IR systems can be as fast and effective (quality-wise) as the monolithic systems. Xu and Callan [24] and Powell et al [16] suggest that the effectiveness of distributed information retrieval systems can drop by up to 30% when the number of collections exceeds 100. However, in our approach the number of collections (number of barrels per user) was on an average about 5–7 (§5). and thus we expect to obtain high quality results as predicted by [12]. In addition, one can use query expansion and source selection [24, 16] to further enhance the effectiveness of our approach.

We would like to distinguish our work from earlier privacy preserving indexing work. Bawa et al [1] present techniques for constructing a privacy preserving index on documents in a multi-organizational setting. Their goal is to construct a centralized index that can be made public without giving out any private information. Similar to other enterprise search techniques they apply access control at query runtime and incur higher overheads than our proposal. Our approach focuses on integrating access control with search in a single enterprise setting and is more efficient.

3. DISTRIBUTED ENTERPRISE SEARCH

Most of the desktop search products are secure but inefficient for enterprise search, whereas enterprise search products are insecure in terms of ACAS. Bearing these issues in mind, we describe the design and implementation of our distributed approach to enterprise search based on the concept of access control barrels (ACBs), which provides both security and efficiency.

3.1 Design Overview

The main design principle of our approach is to efficiently integrate access control into the indexing phase such that the indices used to respond to a user's query are derived only from the data accessible to that user. This will ensure that we satisfy the ACAS requirement and do not have to do access control based filtering on query results. We accomplish this goal with a pre-processing step that (a) constructs a user

access hierarchy for the users and user groups in the system (§3.1.2) and (b) logically divides data into access-privileges based *access control barrels (ACB)*. We first provide a brief description of ACBs and then describe in detail how they work in conjunction with the user access hierarchy.

3.1.1 Access Control Barrels

An ACB is a set of files that share common searchability access privileges (as defined in §2.1.1). That is, all files contained within an access control barrel can be accessed (and thus searched) by the same set of users and user groups. For example, one barrel could contain files accessible to user *bob* and another for a user group *students*. Intuitively, the idea of barrels is that if we can efficiently create collections of files based on their access privileges, to provide secure search to a user, we can pick the collections that this user has access to and serve the query using only those indices.

This might sound similar to the index-per-user (IPU) desktop search approaches [3, 4, 22] where all files accessible to a user are grouped into a single collection and files accessible to multiple users are duplicated in their collections. ACBs avoid their inefficiencies by following an additional neat property of *minimality*. This property ensures that each file can be uniquely mapped to a single barrel, avoiding duplicating them in multiple collections (we defer the discussion of implementing such minimal ACBs to the next section). Now, files accessible to multiple users are grouped into *shared* collections and search for a user combines the user's private collections with these shared collections using distributed information retrieval [12]. This is efficiently accomplished using the user access hierarchy which is described next. We will also compare the ACB approach with the index-per-user approach in detail in §5.4.

3.1.2 User Access Hierarchy

The user access hierarchy data structure has two main tasks: (1) provide a mechanism to map files to ACBs, and (2) provide techniques to efficiently determine all barrels that contain files searchable by a user. In what follows, we first give a high level description of the data structure and later describe its construction for *nix permissions model.

For most access control models a user is associated with two types of credentials: (i) a unique user identifier (*uid*), and (ii) one or more group identifiers (*gid*) corresponding to the user's group memberships. We represent the set of all such user and group credentials as a directed acyclic graph called Access Credentials Graph or *ACG* in short. For example, there could be a node for credential uid_{bob} or $gid_{students}$. Every node V_i in this graph is associated with a corresponding barrel ACB_i. Our example nodes uid_{bob}, $gid_{students}$ are associated with barrels containing files with searchability privileges to user *bob* and group *students* respectively. Now, mapping files to barrels is equivalent to assigning files to a node in *ACG* (the first task mentioned above).

For a node, V_u (associated with the *uid* credential of a user u), let V_u^* denote the set of all nodes in the *directed* graph *ACG* that are reachable from the vertex V_u. Our construction of the graph *ACG* will ensure that a file F is searchable for a user u if and only if F is assigned to some vertex $v \in V_u^*$. With this property, results for u's query can be computed by combining indices from barrels associated with nodes in V_u^*. The set V_u^* can be easily determined using a simple depth first search on the graph *ACG*. This accomplishes the second task of this data structure.

Next, we explain the process of constructing the graph *ACG* with aforementioned properties from *nix-like user credentials. In a *nix-like access control model a credential C can be expressed in Backus Naur Form (BNF) as:

$$\begin{aligned} C &= root \mid all \mid P \\ P &= uid \mid gid \mid P \wedge P \mid P \vee P \qquad (I) \end{aligned}$$

Note that *root* is a special user with super-user privileges and *all* indicates a credential for all users and groups. We need the $\vee$ operator on the principles to handle POSIX Access Control Lists [8] that allow associating multiple users and groups with a file F. We need the $\wedge$ operator on the principles to handle the implicit conjunction operation that occurs while traversing the directory hierarchy leading to file F (for example, directory `X/Y` where `X` has access only for user group *students*, and `Y` has access only for user group *grad-students*; only users that belong to both groups can access data under `Y`). We define an implication operator $\Rightarrow$ which specifies if one credential can *dominate* another. For example, $\forall u$, $root \Rightarrow u$ says that *root* can access data that any other user can.

Permissions on a file can also be expressed based on a credential defined as above. For example, for a file F that allows access to users x, y and group z, we say that it has a credential $C_F = \{uid_x \vee uid_y \vee gid_z\}$ and is interpreted to say that access is allowed to users who have a credential that dominates this file's credential (user x has access to F since $uid_x \Rightarrow C_F$). Now, if we can create a barrel for each such file credential in the system, we can uniquely map a file to a barrel, achieving ACB minimality. While theoretically the total number of barrels (one for each possible access control setting, thus exponential in number of users and groups) can be very large, practically, this is hardly the case as many files in the system share common file credentials. Our tests put this number at 5–7 and similar observations were made in [11]. Regardless, in §3.2, we will describe two optimization techniques that address this potential scalability issue.

Finally, we construct the graph *ACG* as follows. First, the set of vertices and edges are initialized by adding vertices for all user and group credentials and adding edges for the simple $\Rightarrow$ relationships: $root \Rightarrow u\ \forall u \in U$; $u \Rightarrow g\ \forall g \in G(u)$; for any group g, $g \Rightarrow all$, where $G(u)$ denotes the set of all groups to which the user u belongs. Formally,

$$\begin{aligned} V_{ACG} &= \{V_{root}, V_{all}\} \cup \{V_u \mid \forall u \in U\} \cup \{V_g \mid \forall g \in G\} \\ E_{ACG} &= \{V_{root} \rightarrow V_u \mid \forall u \in U\} \cup \{V_u \rightarrow V_g \mid \forall u \in U, \\ &\quad \forall g \in G(u)\} \cup \{V_g \rightarrow V_{all} \mid \forall g \in G\} \end{aligned}$$

where $\rightarrow$ indicates a directed edge in the graph.

Next, the $\vee$ or $\wedge$ nodes are added when we encounter files with such credentials. This is done during the pre-processing step while assigning files to their appropriate barrels (as described later in §4). For each such file, we insert a new vertex V_C for the file's credential C and adjust the graph to include new edges into and out of V_C as follows:

$$\begin{aligned} Dom(C) &= \{V_{C'} \mid C' \Rightarrow C\} \qquad (II) \\ minDom(C) &= \{V \in Dom(C) \wedge \neg\exists V' \in Dom(C), V \in Dom(V')\} \\ Sub(C) &= \{V_{C'} \mid C \Rightarrow C'\} \\ maxSub(C) &= \{V \in Sub(C) \wedge \neg\exists V' \in Sub(C), V \in Sub(V')\} \\ E_{ACG} &= E_{ACG} \cup \{V \rightarrow V_C \mid \forall V \in minDom(C)\} \\ &\quad \cup \{V_C \rightarrow V \mid \forall V \in maxSub(C)\} \\ E_{ACG} &= E_{ACG} - \{V_1 \rightarrow V_2 \mid \forall\, V_1, V_2, V_1 \in minDom(C) \\ &\quad \wedge V_2 \in maxSub(C) \wedge V_1 \rightarrow C \in E_{ACG} \wedge \\ &\quad C \rightarrow V_2 \in E_{ACG}\} (removing\ redundant\ edges) \end{aligned}$$

This completes the construction of the access credentials graph. Using this, we can now map all files to their appropriate barrels and also identify barrels searchable by a particular user (equivalent to finding V_u^* – a simple depth first search operation on ACG).

3.2 Scalability Optimizations

In this section, we formally state two important properties of our construction – ACB minimality and bounds on the number of barrels and the average number of barrels accessible to a user. Based on these properties, we will describe two optimization techniques that can preserve the scalability of our approach even in certain rare environments.

3.2.1 ACB Minimality

Our ACB construction in §3.1.2 satisfies the following minimality property (Refer to Appendix-A for proof).

Claim. *It is impossible to reduce the number of ACBs without either duplicating files in barrel indices or violating the ACAS property.*

3.2.2 Bounds on the Number of Barrels

Let $B(f)$ denote the access control expression on file f. For the sake of simplicity let $B(f) = g_{i_1} \wedge g_{i_2} \wedge \cdots \wedge g_{i_k}$ (argument for $\vee$ nodes is similar). Let the number of literals in $B(f)$ be chosen from the range (1, ngf) using any arbitrary distribution, where ngf is number of $\wedge$-groups per file.

Claim. *Number of access control barrels is bounded by* $min(|F|, 2^{|G|})$ *and the number of access control barrels accessible to a user is bounded by* $min(|F(u)|, 2^{|G(u)|})$

where $F(u)$ is the set of files accessible to u and $G(u)$ is u's group membership. The detailed proof is included in Appendix-B. Note that a large number of ACBs incur only a marginally larger user hierarchy graph maintenance cost. Using a simple depth first search (DFS) algorithm we can limit the effort required to search any vertex/edge in this graph to $O(|U| + |G|)$. In addition, one can build a hash table on top of the graph data structure to reduce the search time to $O(1)$. The runtime overhead and the quality of distributed information retrieval algorithms is only determined by the number of barrels accessible per user. Practically, $|G(u)| \ll |G|$ and $ngf \ll |G|$ and thus, the number of ACBs per user is usually small.

However, theoretically the number of barrels could be exponential. In the following section, we develop two optimization algorithms to address this rare problem. We design our optimizations based on the ACB minimality claim. Our first optimization trades off number of barrels with the number of copies of a file index and the second technique trades off number of barrels while allowing controlled violation of the ACAS property. Both techniques provide a control mechanism for administrators to choose appropriate trade-offs.

Our optimizations transform the access control graph (ACG) with the goal of decreasing the number of ACBs. However, such transformations must preserve *searchability*, that is, if a file f is accessible to user u, then in any transformed ACG, the file f belongs to some barrel b such that b is reachable from V_u on the ACG ($b \in V_u^*$). We call this property *reachability* on the ACG.

3.2.3 Optimization I – File Index Duplication

In this section we propose algorithms to reduce the number of ACBs while satisfying the ACAS property at the cost of maintaining duplicate file indices. Let us consider any vertex V_C in the ACG such that the credential $C \neq u$, for any user $u \in U$. One can eliminate the vertex V_C from the ACG (thereby decreasing the total number of barrels by one) by adding all the file indices in V_C to every vertex $v \in minDom(V_C)$, where $minDom$ is as defined in Equation-II. Note that if $C \neq u$, then $minDom(V_C) \neq \Phi$, that is, there exists at least one vertex $v \in minDom(V_C)$. If V_C is reachable from some vertex V_u (for user u and $C \neq u$) then at least one vertex $v \in minDom(V_C)$ is reachable from V_u; thus the above construction satisfies reachability on the ACG and preserves searchability. The construction preserves the ACAS property since the credential $domC$ associated with any vertex $v \in minDom(V_C)$ dominates the credential C ($domC \Rightarrow C$). Hence, any user u that satisfies the credential $domC$ also satisfies the credential C. Clearly, the above construction eliminates the vertex V_C at the cost of retaining $|minDom(V_C)|$ copies of file indices for each file in V_C.

In our implementation we define a tunable parameter $minf$ – the minimum number of files per barrel (Our ACB construction in Section 3.1.1 achieves $minf = 1$). If a larger $minf$ is chosen the number of barrels decreases at the cost of more duplication of files indices. Given the parameter $minf$ we present a greedy algorithm to reduce the number of barrels as follows. (i) Sort the barrels in increasing order on their size (number of files in the barrel) $b_0, b_1, \cdots b_k$. (ii) Pick the smallest i such that $b_i < minf$ and the credential associated with barrel b_i is not equal to u for any user $u \in U$. If there is no such barrel the procedure terminates. (iii) Eliminate the barrel b_i. Note that this may change the size of other barrels; so we resort the barrels according to their size and repeat the procedure.

3.2.4 Opt II – Controlled ACAS Compromise

Our second technique reduces the number of ACBs while maintaining only one copy of each file index at the cost of violating the ACAS property for some files. Let us consider any vertex V_C such that $C \neq all$. One can eliminate the vertex V_C from the ACG by adding all the file indices in V_C to some vertex $v \in maxSub(V_C)$, where $maxSub$ is as defined in Equation-II. Note that if $C \neq all$, then $maxSub(V_C) \neq \Phi$, that is, there exists at least one vertex $v \in maxSub(V_C)$. If V_C is reachable from some vertex V_u (for user u) then all vertices $v \in maxSub(V_C)$ is reachable from V_u; thus our construction satisfies reachability on the ACG and thus preserves searchability. However, there may exist a user u' such that a vertex $v \in maxSub(V_C)$ is reachable from $V_{u'}$ but not the vertex V_C. This is possible because the credential C dominates a credential $subC$ associated with vertex $v \in maxSub(V_C)$. This violates the ACAS property though keeps a single copy of the index. However, unlike the single-index approach that violates the ACAS property for all the files in the file system, our approach allows us to control the number of such violations.

If $|maxSub(V_C)| > 1$, one can pick a random vertex $v \in maxSub(V_C)$ to add files of the eliminated barrel. However, we heuristically pick a vertex v in such that it satisfies the $minf$ requirement while incurring only a small number of access control violations. Our first heuristic picks the vertex v that has the smallest barrel associated with it. This

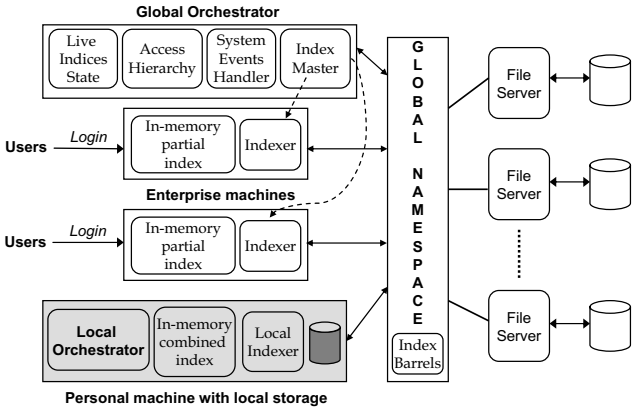

Figure 2: Distributed Indexing and Search

heuristic clearly favors our goal of achieving at least $minf$ files per barrel. Our second heuristic attempts to reduce the number of files violating the ACAS property by picking a vertex v whose credential is the least *popular*. For example, let us suppose that the credential associated with v is a $\wedge$-group $cred = g_{i_1} \wedge g_{i_2} \wedge \cdots \wedge g_{i_k}$. We measure the popularity of the credential $cred$ as $pop(cred) = \prod_{j=1}^{k} pop(g_{i_j})$, where popularity of a group g is determined by the number of members in the group normalized by the total number of user $|U|$ (Similarly for $\vee$-user credentials). Finally, the algorithm for reducing the number of barrels is similar to that in §3.2.3.

These optimization techniques allow administrators to make intelligent choices for their specific environments. For example, files with high update rates may use only the second optimization. This ensures that we have only one copy of the file index and thus keeps the update costs low. Similarly, files with lot of critical information may use only the first optimization technique. This ensures that there are no ACAS violations on the critical file data.

4. SYSTEM ARCHITECTURE

So far, we have introduced the concept of access control barrels (ACBs) and the user access hierarchy as a tool to (a) efficiently map files to ACBs and (b) determine accessible ACBs for a querying user. In this section, we will explain the overall architecture of our indexing and search system and how it fits into the enterprise infrastructure.

Figure-2 shows the architecture. One enterprise machine is chosen as a global orchestrator and is responsible for managing the distributed environment. We will explain its various components in the next subsections. All other participating enterprise machines run a thin client version of the system and are responsible for barrel indexing and query processing for local users. Finally, the personal machines (outside admin control) can integrate their local desktop search with enterprise search by running a local orchestrator agent, an added advantage of our approach.

4.1 Pre-processing: Creating ACBs

As part of the pre-processing step, we first create the basic ACG from the user and user groups in the system as described earlier. For *nix systems user and group information is obtained from */etc/passwd* and */etc/group*. Next, we initiate a filesystem traversal for all data that needs to be indexed. This is required for mapping files to ACBs (represented by a vertex in ACG). During the traversal, we associate each file with a vertex in the directed ACG graph based on its searchability privileges. This mapping is done by finding the vertex that has the same credential as the file (e.g. V_{bob} for credential uid_{bob}). If the file has a $\vee/\wedge$ credential, a new node is added to the access hierarchy and the file is mapped to that node. At the end of this filesystem traversal, we have all barrels in the system and the list of files that are contained in each such barrel. These lists are written to per-barrel files, that are *securely* stored in the enterprise global namespace with access privileges only to the superuser. This stored file is the embodiment of our abstract ACB concept. This completes the pre-processing step and is usually performed by a single enterprise machine – the *global orchestrator*, which stores the user access hierarchy.

4.2 Indexing

After creating ACBs, the next step is to index documents for each barrel. These ACBs can be indexed independently unlike the single index approach where the computation of TFIDF statistics requires centralized indexing of data. The *index master* in the global orchestrator distributes this barrel indexing task to participating enterprise machines. As the barrels are stored in a global namespace and accessible to all enterprise machines, the orchestrator only needs to pass the barrel IDs to these machines. The orchestrator can easily optimize available resources by doing an intelligent distribution of barrels to machines (ensuring no single machine is overly loaded). As we show later in §5, this indexing task distribution provides excellent savings.

On receiving commands from the index master, the *indexer* component of enterprise machine agents retrieve the barrels from the global namespace and start indexing documents. An index is typically comprised of: (a) vocabulary for words that appear in the documents and (b) a words to filename mapping along with their TFIDF statistics used later for ranking. Once indexed, these indices are stored back into the global namespace. Our access privileges based design of barrels provides a natural way of storing indices securely in this namespace. The index files are stored with the same privilege as the files contained in that barrel (all files in a barrel have the same privileges). This allows only the users that had access to files of a barrel (and thus can search through that barrel) to obtain these indices and provides a natural security mechanism for storing these indices using the underlying filesystem access control.

4.3 Search

In our approach, querying and search can also be handled in a distributed fashion. When a user logs into an enterprise machine, the agent on that machine retrieves the indices that are accessible to that user, from the global namespace and caches them in memory. Now whenever a user queries these indices, search can be handled completely in a local environment, saving on (a) query response time and (b) resource requirements of a centralized search server. Note that all available enterprise search products today [20, 10, 19, 2] have to use a highly capable search server (or a cluster) in order to deal with enterprise environments and querying always involves a network hop. In contrast, by integrating access control in a distributed fashion, we can reduce such

requirements. However, our approach has an overhead of combining multiple barrel indices using distributed information retrieval techniques. But unlike the centralized index approaches, we do not have to perform any access control on query results. Our experiments in §5 shows how these two factors tend to balance out.

4.4 Handling Updates

In an enterprise environment, there will be regular updates to data files and access privileges to data and the system needs to handle them appropriately. This task is handled by the global orchestrator which subscribes to all filesystem event notifications using available tools like inotify [14]. Once an event is received the orchestrator might need to make various kinds of changes. Change of file content is handled at a per-barrel basis by requiring that barrel indices to be appropriately modified (it usually does not require indexing the entire barrel again). This event is common to all enterprise search techniques and the ACB based approach does not incur additional overheads. In case of events when access permissions are modified that impact searchability, a document might need to be removed from one barrel and added to another (most indexers can handle this in an incremental manner as well), making it a low-cost event. Another filesystem event is the case of user/group membership modification, in which case the access hierarchy needs to be adjusted. A user/group addition is handled by adding a new node and corresponding edges (as done during initial *ACG* construction). Group membership modification is handled by changing the edges in the directed graph. Finally, a user/group deletion is handled by removing the appropriate node and all edges coming into or out of that node. All these operations are on in-memory *ACG* graph and are efficient. We evalute this in comparison to the index-per-user approach in §5.4.

Note that an update could occur in a barrel that is in-memory at one of the enterprise machines. In order to handle such scenarios, the orchestrator keeps state information of all such in-memory live indices and notifies the appropriate enterprise machine to flush the cached barrel index in that case and reload it after it has been suitably updated.

5. EVALUATION

In this section, we present a detailed evaluation of our approach. First, we compare our approach with existing single index enterprise search approaches. §5.2 describes the indexing experiments including barrels pre-processing and §5.3 describes the querying and search related experiments. We compare our approach analytically with the other ACAS-compliant, but inefficient index-per-user approach in §5.4. §5.5 shows the effectiveness of our optimization algorithms. All experiments were done on a Pentium-III Linux machine with 512 MB RAM and storage mounted via NFS. All numbers below have been averaged over multiple runs.

5.1 Comparison with Single-Index Approaches

In this section, we compare our ACB based approach with the current single-index enterprise search techniques. Recall that the single-index approaches do not satisfy the ACAS requirement and can leak private information to unauthorized users. Through these experiments, we aim to compare any performance overheads incurred by our approach. First, we describe the datasets used in our experiments.

5.1.1 Datasets

The first data set, called T14m, is a publicly available cleaned subcollection [9] of TREC Enterprise track (TREC 14) [23]. TREC 14 is a newly formed track specifically on enterprise search and includes data from the World Wide Web Consortium (W3C) enterprise filesystems. The T14m dataset characteristics are shown in Table-1. It includes emails (*lists*), web pages (*www*), wiki web pages (*esw*) and people pages (*people*). This dataset does not include any access control information.

Scope	Docs	Size	Avg. Doc Size
lists	173,146	485 MB	2.9 KB
www	45,975	1001 MB	23.8 KB
esw	19,605	80 MB	4.2 KB
people	1,016	3 MB	3.1 KB
Total	239,742	1569 MB	6.9 KB

Table 1: T14m: Cleaned TREC 14 subcollections

We also collected statistics from a real multiuser *nix enterprise installation, whose characteristics are shown in Table-2. We collected a subset of the entire directory structure with actual access control settings for 339,466 files arranged in 23,741 directories and replicated the structure in our test environment. The T14m data was used as content for the files (duplicating documents to fill all 339,466 files).

Number of users	926		
Number of user groups	1203		
Number of files	339,466		
Number of dirs	23,741		
Max depth of dir structure	23		
Size of data	2.05 GB		
Number of barrels	2132		
Barrels per user	Max	Avg	Median
	25	6.31	4.26
	21*	5.78*	3.96*

Table 2: Real enterprise dataset; Barrels/user was also computed at a second enterprise (shown by *)

5.2 Indexing Experiments

Indexing is perhaps the most important component of our approach. It includes a pre-processing step that creates the user access hierarchy and the access control barrels followed by actual content indexing of the files contained in ACBs.

5.2.1 Pre-processing

As pre-processing performance is entirely dependent on the enterprise infrastructure (users/groups and directory structure), we use the real enterprise dataset for these experiments. Table-3 shows the evaluation of our implementation.

Task	Performance
Access hierarchy creation	38.7 sec
Barrel creation	263.1 sec
# Files `stat`'ed	202,446 (60%)
# Dirs `stat`'ed	14,059 (59%)

Table 3: Pre-processing performance

Creating access hierarchy for 926 users and 1203 groups took 38.7 seconds, which is a small fraction of the total indexing time. It took 263.1 sec to traverse the filesystem

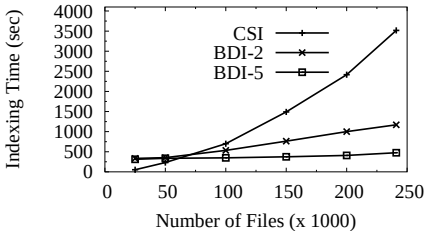

Figure 3: Indexing T14m dataset

and create all ACBs. Additionally only 60% of the filesystem tree needed to be traversed to create all barrels as in many cases, a higher level directory was mapped to a restrictive credential (e.g. only uid_{bob} can access) in which case its contents are automatically added to that barrel without deeper traversal. Overall, these costs are only 10% of the distributed indexing approach (Table-4).

5.2.2 Content Indexing

For indexing of documents, we used the *arrow* indexing and search component of the Bow Toolkit [15]. We modified the ranking algorithm of the toolkit to the distributed IR algorithm of [12]. For our experiments, we considered two architectures: (1) *Centralized Single Index (CSI)* - a centralized single index for the entire dataset analogous to the available enterprise search products (that are *not* ACAS compliant), and (2) *Barrel-based Distributed Indexing (BDI)* - our barrels based distributed indexing approach. *BDI-m* denotes the case when m machines are used to index barrels.

Figure-3 shows the time to index different number of documents of the T14m dataset ranging from 25K to 240K. For BDI architectures, the documents were equally divided between the participating machines for indexing. From the graph, CSI outperforms the BDI approaches when the number of documents is small. This occurs due to the preprocessing costs that the BDI approaches incur. However, as the number of documents increases, BDI quickly outperforms the CSI approach. The distribution of data into ACBs allows us to exploit available enterprise machines for faster indexing (an 85% improvement for 240K files).

The results for the T14m dataset above are a little optimistic as it considers a uniform size and distribution of barrels. However, in reality there could be a few barrels that are significantly larger than the others and dominate the indexing times. To evaluate this, we performed indexing for our real enterprise dataset. As shown in Table-4, the barrel for the `all` node (files that can be read by all users) was significantly larger and took longer than all other barrels combined (and thus total time does not vary with number of machines). However, it was still 38% more efficient than the CSI approach. In general, distribution is most helpful when there are many such large barrels and we expect that to be true in an enterprise-scale environment.

Type	#Max-Docs	Time (s)	Savings
CSI	339,466	4640	–
BDI	189,546	2902	38%

Table 4: Indexing times for real directory structure; #Max-Docs are documents in the largest barrel

5.3 Search Experiments

Recall that searching in our approach requires combining multiple barrels. However, given the small number of barrels per user, overheads should not drastically deteriorate query performance. Secondly, since our approach does not require access control filtering at runtime, there would be savings in query performance as compared to the CSI approach.

For the querying experiments we used 150 queries obtained from TREC 14 Email search. The queries had an average of 5.35 terms per query. The results for CSI and n-BDI (where n is the number of barrels combined) are reported in Table-5.

Type	Index size	Loading time	Avg. time / query
CSI	230 MB	2.5 s	131.12 ms
2-BDI	258 MB	3.37 s	112.89 ms
5-BDI	269 MB	5.68 s	130.68 ms
10-BDI	280 MB	6.90 s	149.90 ms

Table 5: Search performance for T14m *lists*; Loading time is the time to load all indices in memory

First notice that the BDI approaches have slightly larger indices. This is due to the fact that they have to store many words multiple times in different barrel vocabularies. Next, the time to load indices into memory also increases with the number of barrels as there are more file I/Os to gather the index data. However, this is only a one-time cost and once indices are cached, queries proceed normally. Finally, the average query time for BDI approaches is comparable to CSI with 2-BDI and 5-BDI even outperforming it by saving on the privileges check required at runtime in the CSI approach.

We also compared the ranking of the BDI approaches to CSI ranking. For this we evaluated the percentage of top-10 results of the CSI approach that occurred in top-100 of the distributed approach and their average ranks. As shown in Table-6, for our average case of 5 barrels per user, nearly 70% of top-10 results occurred in top-100 of the BDI approach with an average rank of 14. We believe that ranking can be further improved using more sophisticated distributed ranking measures.

Type	10-in-100	Avg. Rank
2-BDI	75%	13
5-BDI	68%	14
10-BDI	61%	15

Table 6: Ranking comparison for TREC 14 *lists*. 10-in-100 is the % age of CSI top-10 results in top-100 of x-BDI and avg-rank is the average rank of CSI top-10 results in x-BDI top-100

5.4 Comparison with Index Per User Approach

For a wider range of experiments and to better analyze the impact of large number of files, we use synthetic data to compare with the other ACAS-compliant index-per-user (IPU) approach. The key parameters for this analysis are summarized in Figure 5.3. For notational purposes, we use Zipf(a, b) to denote a Zipf distribution ($\gamma = 1$) that is truncated to the range (a, b). We choose ngu (the number of groups that a user is a member of) using a Zipf distribution on the range $(2, 10)$ and choose ngf (the number of $\wedge$-groups

Notation	Description	Default
$\|F\|$	Number of Files	10^7
$\|U\|$	Number of Users	10^3
$\|G\|$	Number of User Groups	10^3
pop	Group Popularity	Zipf(1, $\|G\|$)
ngu	Number of Groups per User	Zipf(2, 10)
ngf	Number of $\wedge$-groups per File	Zipf(2, 4)
nuf	Number of $\vee$-users per File	Zipf(2, 4)

Figure 4: Parameters for IPU-ACB Comparison

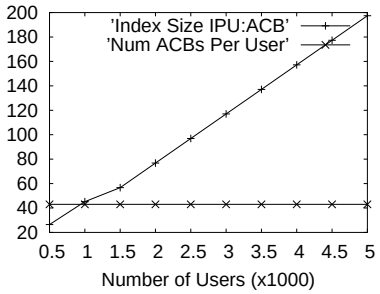

Figure 5: Index Size with # Users

per file) using Zipf(2, 4). These numbers are based on typical enterprise infrastructure as observed earlier.

5.4.1 Index Size Comparison

In this section, we measure the number of ACBs and the number of ACBs per user and compare the size of indices maintained by the IPU and ACB approach. Figures 5 shows the scalability of our approach in index size with increasing number of users $|U|$. In the IPU approach, the index size grows with the number of users, because many users share a file. On the other hand, the ACB approach maintains exactly one copy of each file index and the total number of barrels (and thus the average number of barrels per user) is *independent* of $|U|$. Figure 6 shows the scalability of our approach with ngu the number of group memberships per user. As ngu increases, so does the number of users that share a file and the index size in the IPU approach. In the ACB approach, as ngu increases it results in a smaller in the number of ACBs per user.

5.4.2 Comparison with Access Control Updates

In this section we study the effect of dynamic access control policies on the IPU and the ACB approach using two types of access control updates: adding a new group to a user and adding a new $\wedge$-group to a file. As described in Section 4.4 the ACB approach requires only simple ACG manipulation to reflect these updates and incurs nearly zero cost. In the IPU approach the addition of a new group to a user u's membership requires several file indices to be added to the user u's search index. One would have to inspect all the files in the filesystem to determine such an update thus impeding the scalability of the IPU approach. Figure 7 shows the number of file indices (y-axis) that need to be added when the user u is already a member of $|G(u)|$ groups (x-axis)

Similarly, when a file f's access control expression is changed from $B(f) = g_{i_1} \wedge g_{i_2} \wedge \cdots \wedge g_{i_k}$ to $B(f) \wedge g$, the file f's index has to be removed from several user search indices. One would have to inspect all the users in the filesystem to determine such an update. Figure 8 shows the number of user search indices (y-axis) that needs to be updated versus k (the number of literals in $B(f)$).

5.5 Optimization Effectiveness

In this section, we show the effectiveness of our optimization techniques in decreasing the number of barrels. Figure 9 shows the effectives of our first optimization technique that preserves the ACAS property while maintaining multiple copies of each file index. We plot the tuneable parameter $minf$, the minimum number of files per barrel, on the x-axis. As described in §3.2 our index size increases with $minf$ and slowly reaches the index size of the IPU approach as $minf \rightarrow |F|$. This shows the flexibility of our technique in reducing the number of barrels while incurring significantly lower costs than the IPU approach.

Figures 10 and 11 show the effectiveness of our second optimization technique that maintains exactly one copy of every file index while violating the ACAS property for some files. We have evaluated the effectiveness of our algorithm using three heuristics: `random` chooses a vertex v to add files from the eliminated barrel at random from $maxSub(V_C)$, `size` picks the vertex $v \in maxSub(V_C)$ that has the least number of files, and `pop` picks the vertex $v \in maxSub(V_C)$ that causes the least number of ACAS violations. Figures 10 and 11 show that popularity based approach performs best in terms of both minimizing the number of violations and the number of ACBs. As described in §3.2 the number of violations increases with $minf$ and finally equals that of the single-index approach. This shows the flexibility of our technique in reducing the number of barrels while violating the ACAS property for far fewer files than the single-index approach.

6. CONCLUSIONS AND FUTURE WORK

In this work, we presented an efficient and secure approach to enterprise search. We demonstrated the inadequacy of existing solutions at ensuring access control aware search and developed distributed techniques that elegantly capture access control semantics of enterprise repositories, using *access control barrel (ACB)* and *user access hierarchy* concepts. The distributed and parallel nature of our solution helps improve indexing efficiency and reduces resource requirements for search servers. We also described two optimizations that improve the scalability of our approach even in complex settings. Our experimental evaluation on synthetic and real datasets shows improved indexing efficiency and minimal overheads for ACB processing.

In future, we intend to integrate data-specific indexing and search mechanisms with our approach. This becomes necessary when users want to search (and rank) on higher level metadata concepts along with full content search.

7. REFERENCES

[1] M. Bawa, R. Bayardo, and R. Agarwal. Privacy preserving indexing of documents on the network. In *VLDB*, 2003.

[2] S. Büttcher and C. Clarke. A security model for full-text file system search in multi-user environments.

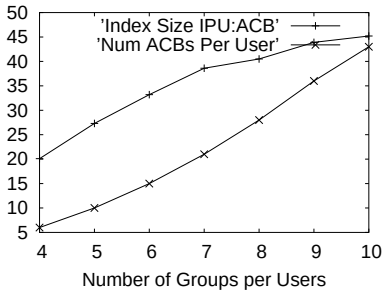

Figure 6: # Groups per User

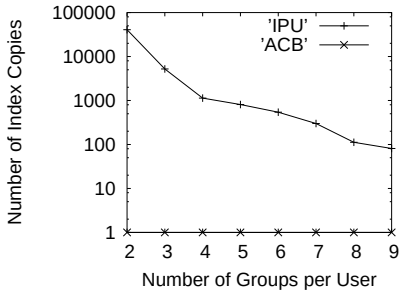

Figure 7: Updating Membership

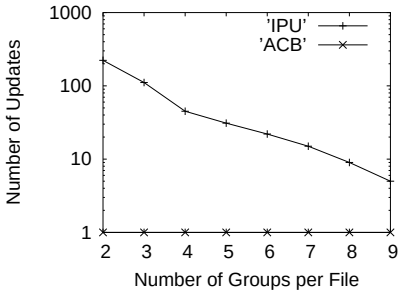

Figure 8: Updating Access Control

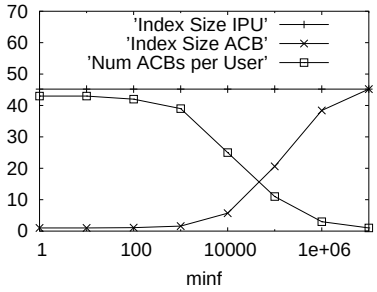

Figure 9: Optimization I

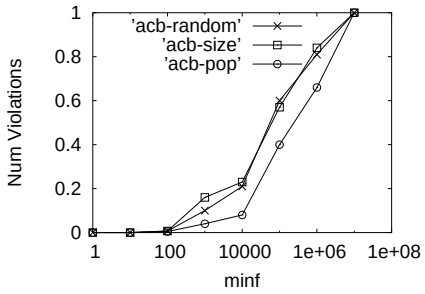

Figure 10: Optimization II: Index Size

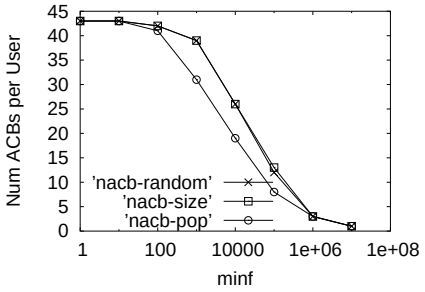

Figure 11: Optimization II: # ACBs

In *USENIX FAST*, 2005.

[3] Google Desktop. *http://desktop.google.com.*

[4] Yahoo Desktop. *http://desktop.yahoo.com.*

[5] Windows Desktop Search for Enterprise. *http://www.microsoft.com/windows/desktopsearch.*

[6] Butler Group. Unlocking value from text-based information. *Review Journal Article*, March 2003.

[7] Gartner Group. *http://www.gartner.com.*

[8] A. Grunbacher and A. Nuremberg. POSIX Access Control Lists on Linux. *http://www.suse.de/%7Eagruen/acl/linux-acls/online.*

[9] Daqing He. Cleaned W3C Subcollections. *http://www.sis.pitt.edu/%7Edaqing/w3c-cleaned.html.*

[10] IBM WebSphere Information Integrator. *http://www-306.ibm.com/software/data/integration/db2ii.*

[11] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *USENIX FAST*, 2003.

[12] O. Kretser, A. Moffat, T. Shimmin, and J. Zobel. Methodologies for distributed information retrieval. In *ICDCS*, 1998.

[13] Linux Manual Pages. *man command-name.*

[14] R. Love and J. McCutchan. inotify linux file system monitor.

[15] A. McCallum. Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering. *http://www.cs.cmu.edu/%7Emccallum/bow.*

[16] A. Powell, J.French, J.Callan, M.Connell, and C.Viles. The impact of database selection on distributed searching. *SIGIR*, 2000.

[17] D. Ritchie and K. Thompson. The UNIX Time Sharing System. *Communications of the ACM*, 17(7), 1974.

[18] S. Robertson, S. Walker, and M. Beaulieu. Okapi at trec-7: Automatic ad hoc, filtering, vlc and interactive. In *TREC*, 1998.

[19] Coveo Enterprise Search. *http://www.coveo.com.*

[20] Google Enterprise Search. *http://www.google.com/enterprise.*

[21] Wikipedia Tf idf. *http://en.wikipedia.org/wiki/Tf-idf.*

[22] MSN Toolbar. *http://toolbar.msn.com.*

[23] TREC Enterprise Track. *http://www.ins.cwi.nl/projects/trec-ent.*

[24] J. Xu and J. Callan. Effective retrieval with distributed collections. In *SIGIR*, 1998.

APPENDIX

A. ACB Minimality

Claim. It is impossible to reduce the number of ACBs without either duplicating files in barrel indices or violating the ACAS property.

Proof. We prove this claim using a simple contradiction argument. Let ACB' denote a set of access control barrels such that the number of barrels in ACB' is smaller than ACB and it contains exactly one copy of each file in a barrel index and it respects the ACAS property. Since ACB' has smaller number of barrels, there exists a barrel $b' \in ACB'$ such that it has two files f_1, $f_2 \in b'$, where f_1 and f_2 belong to two distinct barrels b_1 and b_2 in ACB (by pigeon hole principle). Since f_1 and f_2 are in different barrels in ACB, the files must have different access control expressions. Hence, there may exist a user u such that u can access only file f_1 but not file f_2.

Now, for the file f_1 to be searchable by user u, there has

to be some barrel bar in ACB' such that $f_1 \in bar$ and the barrel bar is reachable from the vertex V_u on the ACG. Since there is only one copy of the file index f_1 in ACB', the barrel b' must be reachable from V_u on the ACG for ACB'. Since the barrel b' is reachable from V_u all the files in the barrel b' must be accessible to the user u. Clearly, allowing the user u to search file f_2 violates the ACAS property. Thus, in order to reduce the number of ACBs, we have to allow either duplication of files indices in barrel or compromising the ACAS property.

B. Bounds on the Number of Barrels

We note that the maximum number of ACBs is bounded by the minimum of the number of files ($|F|$) and the number of *possible* access control expressions. Note that we create a new ACB only if there is at least one file that belongs to that barrel. In the worst case, when every file belongs to a different barrel the number of ACBs is $|F|$. The number of possible access control expressions for files is given by: $\sum_{i=1}^{ngf} {}^{|G|}C_i$, where ${}^{y}C_x$ denotes the number of ways of choosing x balls from y non-identical balls. For example, if $G = \{g_1, g_2\}$ the set of possible access control expressions are $\{g_1, g_2, g_1 \wedge g_2\}$. If ngf is $O(1)$, then the number of ACBs is only polynomial in $|G|$. Hence, the worst case bound for the number of ACBs is $min(|F|, 2^{|G|})$.

Similarly, the maximum number of ACBs accessible to a user u is bounded by the number of files accessible to u and the number of possible access control expressions that u's group membership satisfies. The number of possible access control expressions satisfied by u's group membership is bounded by: $\sum_{i=1}^{ngf} {}^{|G(u)|}C_i$ (note that ${}^{y}C_x = 0$ if $y < x$), where $G(u)$ denotes the group membership for user u. Hence, the worst case bound for the number of ACBs per user is $min(|F(u)|, 2^{|G(u)|})$.