Data Structures for Fast Access Control in ECM Systems

by

Zhiping Wu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2014

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

While many access control models have been proposed, little work has been done on the efficiency of access control systems. Because the access control sub-system of an Enterprise Content Management (ECM) system may be a bottleneck, we investigate the representation of permissions to improve its efficiency. Observing that there are many browsing-oriented permission request queries, we choose to implement a subject-oriented representation (i.e., maintaining a permission list for each subject). Additionally, we notice that with breadth-first ID numbering we may encounter many contiguous IDs under one object (e.g., folder) .

To optimize the efficiency taking into account the above two characteristics, this thesis presents a space-efficient data structure specifically tailored for representing permission lists in ECM systems. Besides the space efficiency, checking, granting or revocation of a permission is very fast using our data structure. It also supports fast union of two or more permission lists (determining the effective permissions inherited from users' groups). In addition, our data structure is scalable to support any increase in the number of objects and subjects.

We evaluate our representation by comparing it against the bitmap based representation and a hash table based representation while using random ID numbering and breadth-first numbering, respectively. Our experimental tests on both synthetic and real-world data show that the hash table outperforms our representation for regular permission queries (i.e., querying permissions on a single object each time) as well as browsing-oriented queries with random ID numbering. However, our tests also show that 1) our representation supports faster browsing-oriented queries with breadth-first ID numbering applied while consuming only half the space when compared to the hash table based representation, and 2) our representation is much more space and time efficient than the bitmap based representation for our application.

Acknowledgements

First, I would like to thank Frank Tompa for his supervision through the whole process of my work. Additionally, I sincerely acknowledge Andrew Kane for suggesting that we look at breadth-first numbering and Simple-9 encoding and Ken Salem for suggesting that we look at hashing.

This research was supported by the NSERC Business Intelligence Network and by the University of Waterloo. Their support is highly appreciated. We also gratefully acknowledge the access control data and the insights provided by partner corporations outside the BIN network.

Table of Contents

List of Tables

List of Figures

Chapter 1

Overview

1.1 Problem Overview

Enterprise Content Management (ECM) systems have been widely used to manage digital contents by organizations. An important purpose of adopting an ECM system is to improve the control of information; therefore access control is extremely crucial for ECM systems. An ECM system usually has a complicated permission inheritance hierarchy [1] with many more subjects and objects than are found in a traditional operating system. Additionally, in an ECM system there are many browsing-oriented permission request queries [2]. To avoid making an ECM system's access control over-complicated, they have very limited support for negative authorizations.

Efficient access control is critical for a variety of data management applications, including ECM systems. Previous work on access control has been focused on the design of models, and very little work has been done systematically on the implementation side. In ECM systems the efficiency of access control can be a bottleneck for system performance because of the complicated permission inheritance, the large number of subjects and objects, and many browsing-oriented queries. All these motivate us to explore a more efficient approach to implementing access control systems.

[1] A subject may directly belong to a group or indirectly belong to a group (its group is a member of another group). We refer to all groups to which the subject directly or indirectly belongs as the *ancestors* of the subject. A subject can inherit permissions from its all ancestors.

[2] In ECM systems, when a user browses under an object (e.g., folder, mailbox, etc.), only the child objects on which the user holds certain permission are presented to the user; thus the system has to implicitly query the user's permissions on all child objects. We call this query a browsing-oriented query.

A typical access control model contains a subject hierarchy and an object hierarchy that allow permission inheritance. Figure 1.1 shows an example of a generic access control model. In this example, Alice gets all permissions assigned to her ancestors and herself.

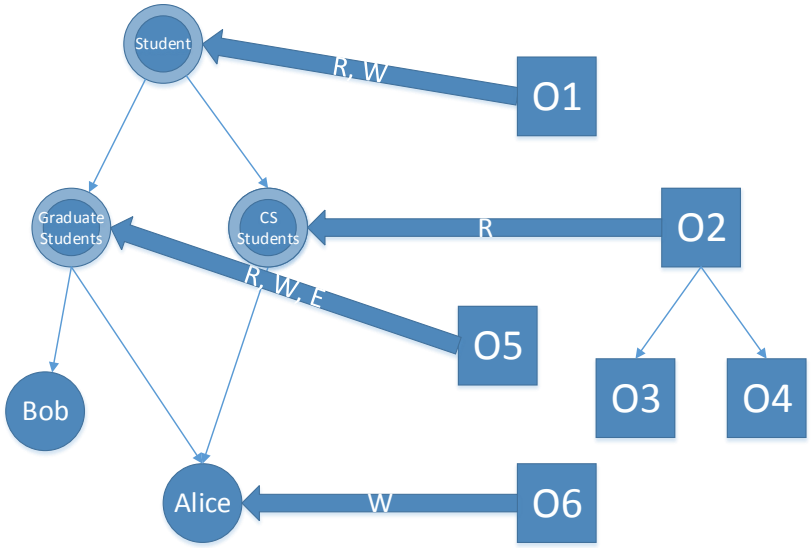

Figure 1.1: An example of a generic model

Traditionally, because an ECM system is installed on top of a relational database management system, it can naturally take advantage of relational tables to store access control information. Up to five tables are necessary, as shown in Figure 1.2. It is also important to create indexes to provide acceptable performance for various types of queries.

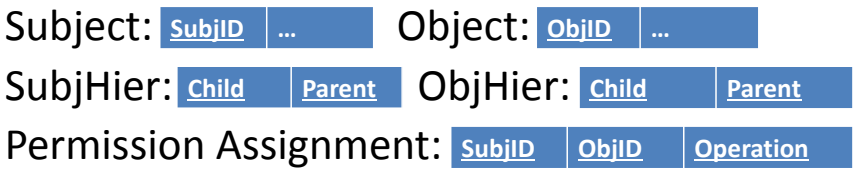

Figure 1.2: Five tables necessary to implement a generic access control model

In ECM systems, however, permission inheritance through the object hierarchy is often eliminated (i.e., in Figure 1.1, Alice does not have permissions on O3 or O4). Instead, the

systems may explicitly copy permissions along the object hierarchy as objects are created. Therefore, only the tables SubjHier and PermissionAssignment are read for access control queries. (Depending on the implementation, we may also have to check the Subject and Object tables if a deleted subject or object is simply marked as deleted without being removed.) For example, given a subject ID, in order to get all effective permissions [3] for the corresponding subject, we have to first find all its ancestors by recursively searching table SubjHier, and we then join the result with table PermissionAssignment. For now, we only consider permission inheritance through the subject hierarchy in our implementation, too.

Some systems may not directly use tables provided by a DBMS but still store access control lists (ACLs) in table-like data structures. Although we can simply put all relational tables (or similar data structures) storing access control data into main memory to speed up the system, we believe that it is necessary to explore a specialized data structure for the representation of ECM systems' access control data.

Our data structure is specifically optimized for ECM systems, taking into account the data and query characteristics. Because there are many browsing-oriented queries in ECM systems, we choose a subject-oriented representation of permissions, which means that each subject has an explicit permission list (a list of all the subject's explicit permissions). Our data structure is very space-efficient, making in-memory computing inexpensive in practice. With breadth-first object ID numbering, a browsing-oriented request may query permissions on several objects with contiguous IDs, and our data structure is optimized for this numbering mechanism. Additionally, it supports fast checking, granting or revoking a permission. Meanwhile, it also supports efficient union of two or more permission lists so that deriving a subject's effective permission list is fast. Another advantage of our data structure is that it is able to scale to support significant increase in the number of subjects and objects.

We then systematically evaluate our data structure against representations based on a hash table and on a bitmap [4] when used to represent access control data. Theoretical analyses and experimental results are presented. We find that a hash table outperforms our data structure for individual permission checking queries (i.e., querying a permission on a single object each time) as well as browsing-oriented queries with random ID numbering since a browsing-oriented query in a system using random ID numbering will have to be processed as multiple individual permission checking queries. However, our tests also

[3] A subject's *explicit permissions* are permissions directly granted to the subject; a subject's *effective permissions* consist of permissions explicitly granted to the subject or to its ancestors.

[4] Instead of a list of object-permissions pairs, a permission list can be viewed as a (usually sparse) bitmap with some bits set to one when the subject has corresponding permissions.

show that 1) our data structure is much more space and time efficient than the bitmap based representation for both individual permission checking queries and browsing-oriented queries using whichever numbering mechanism, and 2) it supports faster browsing-oriented queries with breadth-first ID numbering applied while consuming only half the space as compared to a hash table. To make our tests more realistic, we further test the case when there are contiguous IDs plus a few random IDs under an object. The results illustrate that our data structure can still outperform a hash table for browsing-oriented query after inserting a few objects having non-contiguous IDs.

1.2 The Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 discusses the related work. We first review two approaches to implementing an access control matrix, followed by the discussion of existing work on compression from various communities (including sparse matrix compression, column-oriented database compression and bitmap compression).

In Chapter 3 we first discuss some important characteristics of ECM systems. These characteristics play an important role in the design of our data structure. Then the data structure used to represent subjects' permissions are presented in detail, followed by the description of how we do common access control operations on our data structure. Among these operations, browsing-oriented permission requests have never been seriously discussed in the literature, and our data structure is the only one particularly optimized for this type of queries.

We describe two alternative implementations in Chapter 4. We first describe how we take advantage of a hash table to implement an access control system, including how we do the common operations. Since our data structure can also be interpreted as a compressed bitmap, we also present how a mainstream bitmap compression scheme could be used to implement an access control system.

We compare the space and time efficiency between our data structure and the alternative implementations in Chapter 5. First, how we generate the synthetic data is presented, followed by the description of the real-world dataset. Both synthetic data and real-world data are used for comparison. A detailed performance evaluation and analysis are then presented.

We conclude our work in Chapter 6, in which several directions of future work are also discussed.

Chapter 2

Related Work

This chapter reviews related work, including capabilities and access control lists (ACLs), matrix compression schemes, column-oriented database compression techniques, and bitmap compression schemes.

2.1 Access Control Matrix

First introduced by B.W. Lampson in 1971 [20], an access control matrix is a matrix with each subject represented by a row, and each object represented by a column. A matrix entry M[S,O] is the permissions the subject S has on the object O. An access control matrix represents a static security state; thus all requests in the corresponding state can be answered (assuming that permissions do not propagate). As a static model, we have to also define transition rules (e.g., how entries can be changed) from one matrix (state) to another (state) for a practical system. Table 2.1 shows an example of a simple access control matrix.

Table 2.1: An Example of an Access Control Matrix

	Object 1	Object 2	Object 3
Subject 1	orwx	rw	
Subject 2		r	r
Subject 3	w	rx	

Due to permission inheritance, the effective matrix reflecting all permissions available to each subject may differ from the explicit one [10]. For example, suppose that Subject 3

in Table 2.1 is a group and Subject 2 is a member of the group. If each member of a group can inherit all permissions assigned to the group and Table 2.1 is an explicit matrix, we will have an effective matrix like in Table 2.2. An effective access control matrix is always sufficient to answer any permission requests at the corresponding security state.

Table 2.2: The Effective Matrix of the Explicit Matrix in Table 2.1 (assuming Subject 2 is a Member of Subject 3)

	Object 1	Object 2	Object 3
Subject 1	orwx	rw	
Subject 2	w	rx	r
Subject 3	w	rx	

An access control matrix can be implemented as a set of access control lists (ACLs, column-based), a set of capabilities (row-based), or a combination. For example, an ACL associated with an object contains all (explicit) subjects having permissions on the object and their permissions. In contrast, the capabilities of a subject contain all its permissions on all objects. ACLs are usually managed at a (logically) central point. Capabilities, however, can be passed among subjects. As long as a subject possesses a capability, the subject is believed to have the permission(s) on the object encoded within the capability. Traditionally, most systems are ACL-based, and to the best of our knowledge, all commercial ECM systems are ACL-based.

2.2 Sparse Matrix Compression

An access control matrix is usually sparse. In our application, we require look-up and update of one or more cells to be efficient. Two types of update operations are considered in our context: changing the value in a cell to another value without any pre-knowledge about the value in the cell (whether it is zero or not) or changing a pre-known zero cell to be a non-zero one. The latter type is usually called insertion of a cell. The cost of update is dependent on the cost of look-up and the cost of insertion (a look-up followed by either a change of the value or an insertion). We primarily discuss look-up and insertion in this chapter. Additionally, a cell with the value of zero and an empty cell are not distinguished in the rest of the thesis.

There are many schemes for compressing a sparse matrix. Generally speaking, we can group those schemes into two categories: one category is optimized for space and fast

insertion (often slow look-up); the other category is designed for fast matrix operations, such as multiplication (slow insertion). Dictionary of Keys (DOK), List of Lists (LIL), and Coordinate List (COO) fall into the first category, while Compressed Sparse Row (CSR or CRS) and Compressed Sparse Column (CSC or CCS) fall into the second category [12]. Typically, in the community of scientific computing, schemes in the first category are used to construct a sparse matrix, and then the matrix is transformed into the format of a scheme in the second category for further computation [12]. There also exist schemes that aim to balance the efficiency of insertion and other matrix operations. We review several typical schemes below. There are also many schemes designed for special matrices (e.g. banded matrix, diagonal matrix, and symmetrix matrix) [12]; we, however, are not interested in these special schemes.

2.2.1 List of Lists

List of Lists (LIL) stores one list per row for the non-zero cells, where each entry stores a non-zero cell's column index and value. Four arrays are used to implement the list of lists. The first array, A, contains all non-zero values in the matrix. The second array, C, stores the corresponding column indexes for each element in array A. The third array, NEXT, stores the index of the next element for each element in arrays A and C (-1 if no next element). The last array, R, contains the index of each row's first non-zero element in arrays A and C. For example, a matrix

$$M = \begin{bmatrix} 10 & 20 & 0 & 0 & 0 & 0 \\ 0 & 30 & 0 & 40 & 0 & 0 \\ 0 & 0 & 50 & 60 & 70 & 0 \\ 0 & 0 & 0 & 0 & 0 & 80 \end{bmatrix}$$

may be compressed to be (zero-based index)

$$\begin{array}{lllllllll} A = & [& 30 & 20 & 10 & 70 & 50 & 60 & 40 & 80 &] \\ C = & [& 1 & 1 & 0 & 4 & 2 & 3 & 3 & 5 &] \\ NEXT = & [& 6 & -1 & 1 & -1 & 3 & 4 & -1 & -1 &] \\ R = & [& 2 & 0 & 5 & 7 & & & & &] \end{array}$$

Logically, the above arrays store four lists as the name of the scheme list of lists indicates. Figure 2.1 presents how the non-zeros are stored logically.

This scheme is usually used to construct a small matrix. It supports fast insertion, but the lookup of an entry is slow. For insertion at (RowID, ColumnID), we simply have to add

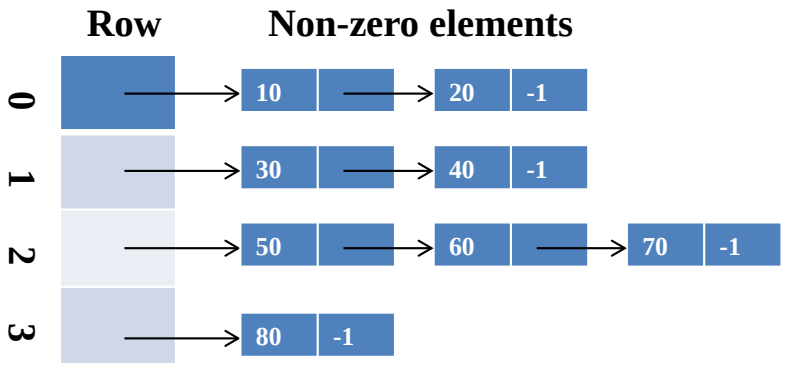

Figure 2.1: A logical example of the list of lists (LIL)

the new element at the front of the list of the row RowID. Specifically, we first append the value of the newly inserted entry at the end of array A and the column number ColumnID at the end of array C; second, the value in array R associated with the row RowID (i.e., R(RowID)) is appended to array NEXT; finally, the value of R(RowID) is changed to refer to the last entry in arrays A, C, and NEXT. For lookup of the value at a specific position, we, however, have to linearly search the non-zeros in the corresponding row.

2.2.2 Coordinate List

The Coordinate List (COO) scheme stores a list of (row, column, value) triples for all non-zero cells. For example, the matrix in Section 2.2.1 may be compressed to the list of triples (zero-based index) in Table 2.4. Alternatively, we may use three arrays, each of which stores the values of a corresponding row in Table 2.4 in the same order.

Theoretically, the triples can be in any order. In practice, however, they are usually stored in insertion order since we simply append a triple to the end of the list whenever we are inserting a value into a cell in the matrix (very efficient insertion). Note that storing tuples in insertion order does not help improve the efficiency of look-up, and a linear search is inevitable anyway in order to look up (or then update) the value of a specific cell. This also indicates that a general update of the value of a cell can be slow since we have to go through the list in order to find the corresponding cell first.

Overall, COO is slow for look-up (and general update). It is, however, very efficient for insertion.

Table 2.3: A Matrix Compressed by COO

row	column	value
0	0	10
0	1	20
1	3	40
1	1	30
2	3	60
3	5	80
2	4	70
2	2	50

2.2.3 Dictionary of Keys

Dictionary of Keys (DOK) encodes non-zero cells as a dictionary (hash table) mapping <row, column> pairs to values (a <row, column> pair is a key). Obviously, DOK consumes more space than COO does (i.e., extra space for buckets and pointers). Various hash tables may be implemented. This scheme supports fast insertion and look-up (O(1)); however, iterating over non-zero values in sorted order is not well supported since the order of the non-zero cells is random after compression.

2.2.4 Compressed Sparse Row

Instead of storing both row and column information for each non-zero cell, Compressed Sparse Row (CSR) further compresses the row information. Thus it is more space efficient than COO.

Let NNZ denote the number of non-zero cells in an m $\times$ n matrix M. COO needs a table containing 3 $\times$ NNZ cells (or 3 arrays, each of which is of length NNZ) to represent M. Using CSR, 3 arrays are necessary. The first one, A, and the second one, C, are both of length NNZ. The array A holds all non-zero elements of M in strict left-to-right top-to-bottom order; the array C keeps the column index for each element in array A. The last array R is of length m + 1, containing the starting pointers to the elements in array A for each row. Therefore, row i contains all elements from A (R(i)) to A (R(i + 1) - 1). For the special case that row i has no non-zero cells, we will have R (i + 1) = R (i). The last element of array R equals to NNZ (zero-based index for array A), which is the ending flag.

Using CSR to compress the matrix in Section 2.2.1, we will get three arrays after

compression (zero-based index)

$$
\begin{array}{lcccccccccc}
A = & [& 10 & 20 & 30 & 40 & 50 & 60 & 70 & 80 &] \\
C = & [& 0 & 1 & 1 & 3 & 2 & 3 & 4 & 5 &] \\
R = & [& 0 & 2 & 4 & 7 & 8 & & & &]
\end{array}
$$

Besides the space-efficiency, look-up is fast using CSR since we can efficiently find elements in a specific row and then use binary search to reach the element in the specific column. The major drawback of this scheme is that insertion is expensive. Suppose a new element needs to be inserted into row i. We have to insert an element in array A and C, respectively (by shifting elements and potentially expanding the arrays). Also, we have to modify the pointers for each row starting from row i + 1. In fact, in scientific computing, a matrix is transformed to CSR format only when it is assumed to be static (no more insertions).

2.2.5 Compressed Sparse Column

Compressed Sparse Column (CSC) is very similar to CSR with the exception that the column information is compressed instead of row information. Therefore, we need an array A to keep all non-zero elements of a matrix with the strict top-to-bottom left-to-right order, an array R to record the row indexes for each element in A, and an array C to keep pointers to element in array A for each column. For example, the matrix in Section 2.2.4 is compressed into the following arrays

$$
\begin{array}{lcccccccccc}
A = & [& 10 & 20 & 30 & 50 & 40 & 60 & 70 & 80 &] \\
R = & [& 0 & 0 & 1 & 2 & 1 & 2 & 2 & 3 &] \\
C = & [& 0 & 1 & 3 & 4 & 6 & 7 & 8 & &]
\end{array}
$$

CSC has the same advantages and drawbacks as CSR does. For an m $\times$ n matrix, whether CSC or CSR is more space-efficient depends on m and n (CSR if m $<$ n; CSC if m $>$ n). Please note that both CSR and CSC may consume more space than COO for matrices containing many rows or columns without any non-zero cells.

2.2.6 MTL4

While most schemes are optimized for either look-up or insertion, Matrix Template Library 4 (MTL4) provides a scheme that balances look-up and insertion by pre-allocating fixed-sized space for each row or column [13]. It then stores each row's or column's non-zero

cells in the pre-allocated space for each row or column (the space has to be big enough for the row or column with the largest number of non-empty entries). For example, we use this scheme to represent the previous matrix in Section 2.2.1. Assuming that we choose a row-based compression and there are at most 4 nonzeros in a row, we will get

$$
\begin{array}{lcccccccccccccccc}
A = & [& 10 & 20 & \varnothing & \varnothing & 30 & 40 & \varnothing & \varnothing & 50 & 60 & 70 & \varnothing & 80 & \varnothing & \varnothing & \varnothing &] \\
C = & [& 0 & 1 & \varnothing & \varnothing & 1 & 3 & \varnothing & \varnothing & 2 & 3 & 4 & \varnothing & 5 & \varnothing & \varnothing & \varnothing &] \\
R = & [& [0,2] & [4,6] & [8,11] & [12,13] & & & & & & & & & & & & &]
\end{array}
$$

The result is similar to CSR; however, there are two differences. First, we have pre-allocated unused entries (represented by phi). Second, the array, R, is now a 2-dimensional array containing the starting and ending pointers for each row's used entries; for example, the first element $[0, 2]$ of R indicates that Row 0 contains the 0th to 1st (2-1) elements in array A; the 2nd and 3rd elements are also reserved for Row 0.

Obviously, with this scheme look-up is as fast as with CSR. For insertion, we (binary) search the elements in the corresponding row, and then insert the new element into the corresponding space; thus no other rows will be affected. The representation, however, relies on there being a small number of non-zero entries in each row or column. Thus this scheme does not work well if some rows/columns have just a few non-zero entries but some others have a relatively large number of non-zero entries (wasting considerable space).

2.3 Column-oriented DB Compression

Similar to a row or a column in an access control matrix, a column in a database table may be sparse (i.e., many entries have the value zero or null). Therefore, work on database compression, particularly on column-oriented database compression, can be relevant to our work. We thus review several techniques widely used in column-oriented database compression (except those bitmap compression techniques that will be discussed in the next section).

There are roughly two categories of compression schemes for column-oriented database compression: heavy-weight compression and light weight compression [1]. Lempel-Ziv encoding is the most common technique in the heavyweight category. It does not have to know the pattern frequencies in advance; instead, it dynamically builds the pattern table while encoding the data. Heavyweight schemes can usually achieve very good compression. However, complete decompression and re-compression are required when the data is updated when using Lempel-Ziv encoding as well as other heavyweight compression schemes.

This makes direct processing of compressed data impossible. Thus these schemes are not suitable for a dynamically-updated database (including our application).

The second category, light compression, however, allows us to process the compressed data directly. Null Suppression, Dictionary Encoding, and Run-length Encoding (RLE) are three commonly-used techniques.

The fundamental idea of Null Suppression is that zeros or empty cells are not stored explicitly (instead, storing just the description of where and how many exist). This technique usually works well when there are many zeros or empty cells. This technique is widely used for not only database compression but also the compression of other data, and it has many variations taking the data characteristics into consideration . Meanwhile it is often integrated with other techniques. For example, all sparse matrix schemes discussed in Section 2.2, in fact, exploit Null Suppression.

Dictionary Encoding uses a shorter pattern to replace a longer pattern. A mapping between actual values and dictionary entries is necessary when using this technique. This technique is particularly effective for a domain in which the number of possible values is very limited but every value is space-consuming. For example, suppose that a company has two campuses and there is an attribute (column) recording each employee's campus address. We may create a dictionary table like below

Table 2.4: A Dictionary Table

Real value	Dictionary code
888 Fake Road, Waterloo, ON Canada	0
999 NotReal Street, Toronto, ON Canada	1

We can then replace every string value in the column with only one bit.

RLE encodes a run (contiguous repetitions) of the same value to a compact representation (e.g., a triple of value, starting position, and run length). We may use less data for each run in some cases (i.e., for a domain with only two values). This technique is particularly useful in a sorted database.

Again, all these techniques have many variations, and careful consideration regarding the data and application characteristics is necessary before choosing or designing a suitable variation. It is also quite usual that more than one technique is used in one database in order to get the optimal outcome.

2.4 Bitmap Compression

In our application, we frequently check, set, and clear a specific bit of a cell's value; therefore, we can also naturally interpret a row or a column of the access control matrix as a bitmap.

We begin by noting immediately that we are only concerned with lossless bitmap compression, and therefore do not consider lossy image compression (such as JPEG) to be relevant. Research on lossless bitmap compression has been introduced by DB researchers designing column-oriented database systems. At the same time, IR researchers have introduced many techniques for compressing ordered lists of integers representing document IDs, with or without storing positions within the corresponding documents. Because there is an equivalent bitmap for any list of document IDs, we use the term bitmap compression to cover both types of work.

Although compressing a bitmap and compressing an ordered list of integers are logically equivalent problems, DB researchers and IR researchers developed quite different schemes because of their different points of departure. Specifically, the input for compression in column-oriented databases is a bitmap, while the compression schemes for information retrieval take a list of integers as input.

Naturally, the fundamental idea among DB researchers is to compress contiguous 1s or 0s into smaller space (e.g. using a byte or word to represent several contiguous 1s or 0s). In general, we can categorize this work into two groups: byte based schemes, which consider a byte as the smallest unit, and word based schemes, which consider a word as the smallest unit. Byte-aligned Bitmap Code (BBC) [5] and PackBits (PAC) [14] fall into the first group, while Hybrid Run-Length encoding (HRL) [25], Word-aligned Bitmap Code (WBC) [30, 29], Pack Word Code (PWC) [30, 29] and Word-Aligned Hybrid run-length code (WAH) [30, 29] fall into the second group. None of these schemes require decompression in advance for bitwise operations; instead, simple interpretation is sufficient. Researchers have shown that word based schemes are usually faster for both compression/decompression and bitwise operations at the cost of a little extra space, since modern CPUs access data by word [30, 29]. Please refer to the paper by Wu et al. [29] for more detailed comparisons among the schemes mentioned above.

In contrast, the core idea of compressing a list of document IDs is to use less space to represent an integer. The compression procedure is usually broken down into two steps, as follows. The first step is to transform the list of document IDs to a list of differences (d-gaps) so that most elements in the list become smaller integers. The second step is to represent each d-gap using one or more bits, bytes, or a fraction of a word. Variable length

sequences of bytes (vbytes) [11] is a standard compression algorithm which contains 7 bits of d-gap and 1 bit indicating whether additional bytes are needed. Simple-9 [3, 4] and its extension Simple-16 [32] encode multiple d-gaps into one word. The four most significant bits of a word are used to indicate the number of d-gaps encoded in the word. Simple-16 has been shown to have faster decompression.

Many other compressions schemes have been proposed with the aim to reduce compression and decompression time. For example, PFOR-Delta [34] encodes d-gaps in batch sizes of some multiple of 32; it is not word-aligned and requires decompression of each batch prior to performing bitwise or lookup operations. Such schemes are not suited to our task.

We review two typical and influential schemes (i.e., WAH and Simple-9) developed by DB researchers and IR researchers, respectively, in detail below.

2.4.1 Word-Aligned Hybrid

Word-Aligned Hybrid run-length coding (WAH) is a very popular bitmap compression scheme. WAH encodes long run of contiguous 0s or 1s using run-length encoding (called a fill), and represents a mixed-value word in its literal version. Therefore, there are two types of words: fill word and literal word. In WAH, each word represent (w-1) $\times$ N literal bits (N is a natural number and N $\geq$ 1), where w is the length of computer word (e.g., 32 or 64). In any word, the most significant bit (MSB) is used as a flag to distinguish a fill word and a literal word (0 for a literal word; 1 for a fill word). For a literal word, the next w-1 bits is simply a copy of the actual value. For a fill word, the second MSB is called the fill bit which represents the value of the contiguous bits. The rest of a fill word encodes the length of the run (number of w-1 bits); for example, in a 32-bit implementation, 62 contiguous 0s may be encoded as 10000000000000000000000000000010. Due to its word-aligned requirements, we may not have a full w-1 bits in the last word we want to encode. Thus there is a special tail word to encode the last few bits of the bitmap (a literal word). There is also an additional word to record how many bits are used in the tail word.

Figure 2.2 shows how a 128-bit bitmap is compressed using a 32-bit WAH [30].

128 bits	1,20*0,3*1,79*0,25*1			
31-bit groups	1,20*0,3*1,7*0	62*0	10*0,21*1	4*1
groups in hex	40000380	00000000 00000000	001FFFFF	0000000F
WAH (hex)	40000380	80000002	001FFFFF	0000000F 00000004

Figure 2.2: An example of WAH encoding [30]

Although WAH requires slightly more space than previous byte aligned schemes, it can better exploit modern CPUs to get better performance of bitwise operations. WAH, however, fails to take into account the efficiency of checking, setting, or clearing given bits in compressed bitmaps.

2.4.2 Simple-9

As a scheme designed to compress an ordered list, Simple-9 first transforms the list of positions to a list of d-gaps. It then encodes as many d-gaps (up to 28) into one 32-bit word as possible (a word-aligned scheme). The four most significant bits of a word are used to indicate the number of d-gaps encoded in the word (called a selector). The remaining 28 data bits encode up to 28 d-gaps, each of which occupies exactly the same number of bits. Table 2.5 shows the 9 possible ways in which a word is partitioned. For some cases, a few bits are wasted.

Table 2.5: Simple-9 encoding options in a 32-bit word

Selector (4 bits)	Number of coded d-gaps	Length of each code (bits)	Number of wasted bits
0000	28	1	0
0001	14	2	0
0010	9	3	1
0011	7	4	0
0100	5	5	3
0101	4	7	0
0110	3	9	1
0111	2	14	0
1000	1	28	0

Interestingly, given a d-gap, instead of encoding its actual value, Simple-9 encodes the actual value minus 1. For example, a list of d-gaps (5, 2, 1) will be encoded as (4, 1, 0). In this case, a bit can be used to encode the value of 1 or 2 (a d-gap is always at least 1).

During the compression, Simple-9 first checks whether the next 28 d-gaps can be encoded into one word; if not, it will check whether the next 14 d-gaps can be encoded into one word; this process will not stop until one of the nine possible ways is found to be appropriate. (Since there are at most 28 bits used to encode a d-gap, any d-gap greater than 2^{28} cannot be encoded.) For example, suppose we have a posting list (4, 11, 12, 13, 16, 21, 22, 29, 30, 42, 65, 66, 76, 94). It will first be converted to a list of d-gaps (4, 7, 1, 1,

3, 5, 1, 7, 1, 12, 23, 1, 10, 18). Then two words will be used to encode these d-gaps. The first word is (0010,011,110,000,000,010,100,000,110,000,∅), where a ∅ means an unused bit, and the second one is (0100,01011,10110,00000,01001,01001,∅∅∅)

Simple-9 has an extension called Simple-16 which has been shown to have faster decompression. We are, however, not interested in decompression efficiency since this is irrelevant to our application.

Chapter 3

Blocked and Ordered Permissions

In this chapter, we first discuss the characteristics of access control sub-systems in ECM systems and explain why we choose a subject-oriented implementation. Then a simple, yet novel data structure to encode subjects' permission lists is presented, followed by the description of how to carry out necessary operations in our system.

3.1 Characteristics of ECM System's Access Control and Subject-oriented Representation

We first assign sequential numerical IDs to all subjects (users, roles, and groups) and to all objects (files, directories, mail messages, etc.). Observing that the number of permission types (PTypes) in an ECM system is typically pre-defined (e.g., 11 types to represent permissions to read an object's name, read its content, append to its content, alter its content, etc.), we use p = |PTypes| bits to represent a subject's permissions on an object; thus, we can always use p bits to encode a subject's permissions on an object, and if a subject has k permissions on an object, the corresponding k bits are set to 1 and the other p-k bits for that object are set to 0. This results in an access control matrix in which each cell contains p-bit data. The explicit permissions are stored, and permission inheritance will be handled at run time when necessary. Table 3.1 shows an example of an access control matrix in an ECM system.

An access control matrix, traditionally, is implemented by either subjects (rows) or objects (columns). In our context, storing the matrix as a whole is inappropriate for several reasons. First and most importantly, a permission request is against either a subject or

Table 3.1: A simplified access control matrix in a typical ECM system (the first row represents objects IDs; the first column represents subject IDs)

	0	1	2	3	4	5
0	11111111111	00000000000	00000000000	00000000000	10111110001	00000000000
1	00000000000	00000000000	00000000000	00000000000	00000000000	00000000000
2	10000000001	00000000000	00000000000	00000000000	00000000000	00000000000

object (e.g., query a specific user's permissions (under a folder), or query who has what permissions on a specific object). This makes subject- or object-oriented implementation quite natural. Second, there are usually many inactive (logged out) users in an ECM system. For in-memory implementations, keeping the access control data of the inactive users in main memory can be a significant waste of main memory, even with a compact data structure applied.

In ECM systems, when a user accesses an object (e.g., folder, mailbox, etc.), only the child objects on which the user holds certain permission are presented to the user; thus there are many implicit permission requests (browsing-oriented queries), and these requests are all subject-oriented. We therefore believe that a subject-oriented implementation is more efficient for ECM systems and decide to choose subject-oriented representation. Each subject will have a permission list containing all its explicit permissions. Additionally, we notice that if breadth-first numbering is applied to object IDs, most IDs involved in a browsing-oriented query are contiguous; we therefore believe that the performance may benefit from storing objects in order in a permission list.

It is also important to point out that although our subject-oriented implementation is like a capability system [16, 21, 24], our implementation is still ACL based since several major characteristics of a capability system are missing from our system. First of all, a subject in ECM systems is a user or a group (a role can be viewed as a group, too); in a capability system, however, a subject is a process. Second, we strictly distinguish subjects and objects while an 'object' in a true capability system can be a process (subject in an access control matrix) or a piece of data (object in an access control matrix). Third, a permission list does not contain any address reference to the corresponding objects; thus a permission list is not a list of capabilities by definition. In fact, our implementation simply replaces the permission-checking module of an ECM system, and it still takes advantage of existing mechanisms to protect the permission lists (using whatever protects ACLs in existing systems) and access objects after checking permissions. Therefore our system is a subject-oriented ACL implementation.

3.2 Description of Blocked and Ordered Permissions

In a real-world ECM installation, we may input more than 6000 permission lists, each of which contains the corresponding subject's access control information on around 8 million objects. Figure 3.1 illustrates a section of a permission list interpreted as a list of p-bit units. We assume p = $|PTypes|$ = 11 throughout the rest of this thesis and require a mechanism to store a sequence of p-bit units. If all objects were explicitly included in a permission list, each permission list would consume around 80 million bits, which means that the literal version of the whole access control matrix requires more than 60 GB space. Therefore, a more compact data structure used to represent the permission lists is necessary. We present our space-efficient permission representation data structure called Blocked Ordered Permission List (BOP) [1] in this section.

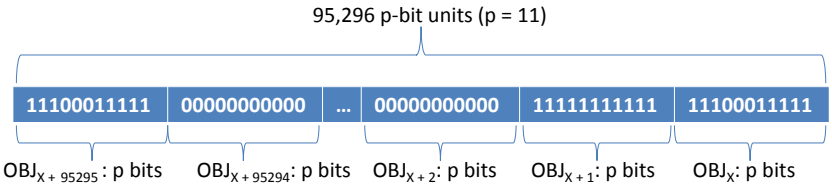

Figure 3.1: An example of a section of a permission list

3.2.1 Blocking

We first divide a permission list into multiple blocks, each of which (logically) contains 95,296 objects with permissions on them (95,296 p-bit units). We have noticed that this choice of the block size results in quite satisfactory performance, although we cannot guarantee that the current setting is the best. Figure 3.1 in Section 3.2, in fact, is an example of a block, in which many units are all-0. We omit blocks in which no bit is set to 1, so the number of stored (physical) blocks may be smaller than the number of logical blocks. Our data structure explicitly records the offset of the first logical unit for each stored block (block offset) in an index so that locating a desired block within a permission list is efficient. Furthermore, we represent each block (as described shortly) using our data structure (which can be considered as compression), with the result that physical blocks have variable size.

[1] It was initially introduced as a bitmap compression scheme [28]. This is very space and time efficient for dynamic bitmaps with many small clusters.

One advantage of blocking is that it can significantly reduce the search space when looking for an object. With the block indexes, we can easily jump into the block that contains the object (in fact, the corresponding permission unit). The search space is then reduced to permission units in a single block instead of all units.

A second advantage is to reduce the space required to represent the position of a unit. With over 8 million objects, we need 24 bits to represent a position of an object; however, with blocking, we need 17 bits to represent a unit's offset to the beginning of a block. The absolute position can be easily computed by adding the recorded unit offset to the block offset.

A third benefit is that we can represent blocks in different formats when appropriate, depending on each block's characteristics (i.e., density of non-zero units).

A final advantage of blocking is to reduce the cost of I/O when pure in-memory implementation is infeasible. Physical blocks are our units of I/O, and the indexes to all physical blocks are kept in main memory. We are thus able to read a specific block from disk instead of a whole permission list. As most operations will check data in only one or two blocks, we can reduce the I/O cost significantly in the case of limited available main memory.

3.2.2 Representation within a Block

A unit with all 0s is not physically stored, and units with one or more 1s (non-empty units) are stored as a list of pairs representing (a) the logical offset of the unit from the first unit in the block and (b) the p bits of the units values. (Because we record the logical offset of each block itself, we can number the logical units within a block starting at 0.) For the access control application, each unit stores all permissions a subject holds on an object. A 32-bit word (permission word) is used to represent each unit of bits; in our implementation, the 11 most significant bits store the unit's literal values and the remaining 21 bits represent the logical offset of the unit from the first unit of the block.

Figure 3.2 shows the 3-word representation using our data structure for the original block in Figure 3.1. It is significant to understand that this approach works only when the permission list is sparse (or if the permission list is very dense, by making the default value 1 instead of 0). In our application, most permission lists are quite sparse, since a subject typically has no permissions on most objects in a large enterprise.

If there are 32,758 (34.375 % of the block size) or more non-empty units in a block, our data structure consumes at least as much as the original version. In this case, we keep the block as a literal bit vector.

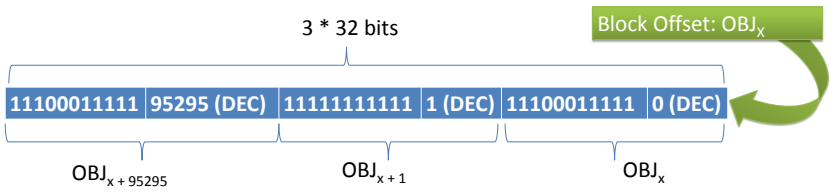

Figure 3.2: An example of a block of a permission list represented by BOP

3.2.3 Unused Bits

With the proposed choices for block size, BOP includes 4 unused bits in every word encoding a non-empty unit. Although this wastes some space, there are at least two benefits resulting from leaving these bits unused. Most importantly, using a full word makes our scheme word-aligned, which makes the implementation easier and the processing faster [6]. Second, up to four additional permission types can be easily supported (making p = 15, without making any changes to the encoding or the algorithms). For example, if a new permission type is introduced by the ECM system, we simply have to let the system know that one additional bit following the current copy of the permission unit is now used in any permission word, and the existing encoding of permissions immediately represents that new permission as being denied to all subjects with no changes whatsoever to the BOP blocks. Specifically, if the number of permission types increases from 11 to 12, BOP will then use the 12 most significant bits to represent the permission unit in a 32-bit permission word (we still just need 17 bits to encode any unit offset, and the number of unused bits in a permission word decreases to 3 from 4).

3.2.4 Scalability

We may consider scalability in three dimensions, including the increase in the number of subjects, objects and permission types. Our data structure can support all three dimensions quite well. When a new subject is added into the system, we simply have to create an empty permission list for it. Permissions can be granted afterwards. When a new object is created, the logical length of every permission list grows; however, no physical operation is necessary until permissions on the newly created object are granted to a subject (appending one and only one permission word to the end of the corresponding permission list). We have discussed the increase in the number of permission types in Section 3.2.3. Increasing the number from 11 to up to 15 is almost free. It is more difficult to support more permission types (having to use longer words or choose smaller block size), but practically it is quite

unlikely that there will be significant increase in the number of permission types in an ECM system after it is deployed.

3.3 Operations on the Permission Lists

3.3.1 Individual Permission Checking

Checking whether or not a user has a specific permission on an object is a common operation in any access control systems. With a subject-oriented representation, there are two approaches. The first one is to 1) compute the effective permission list of the corresponding user (the union of the explicit permission lists of the user and all its ancestors) and then 2) check the value at the corresponding position in the effective permission list. The second one, however, avoids the computation of the effective permission list. Instead, the corresponding positions in the explicit permission lists of the user and all its ancestors are checked individually. As long as one of the permission lists is found to include the permission requested, the user will be authorized the permission. Noticing that one union operation is far more expensive than one checking of a specific position in a permission list, we choose the second approach. We explain how we check a specific position on a permission list below.

Given the object number and number of the specific permission type, the first step is to find the corresponding block. The block index provides starting addresses for all logical blocks. If the index to the desired block points to NULL, the permission list contains no permissions at all on all objects in the block, including the object we are checking for, and thus the return value is false. This step requires O(1) time. If the block physically exists, we (binary) search for the corresponding unit within the block, returning false if it is not physically stored (indicating the permission list contains not a single permission on the requested object). Otherwise, we return the corresponding bit value within the unit. The total time is then O(1) to find the block, O(log N) to find the unit using binary search (where N is the number of physical units in the block), and O(1) to find the bit within a stored unit.

3.3.2 Permission Granting

Granting a permission to a subject is another common operation in an access control system. This requires setting a bit to 1 at a corresponding position in our representation.

To complete this operation, we first locate the unit containing the bit as above. If it physically exists, indicating the permission list contains at least one permission on the object so that at least 1 bit has been set to 1 in the unit, we simply store 1 as the value for the bit we wish to set; otherwise, we (first create a new physical block if necessary, and then) insert an element into the sorted array holding permission words of the corresponding block. Noticing that we will have to expand and then relocate an array when we are inserting an element into an array without any spare space, we implement a buddy system [18] to reduce the frequency of expansion and relocation of arrays (always doubling the array when it is full). Another benefit gained from the buddy system is that it is much easier to back up a block to a disk, although we mainly consider in-memory computing. If backup does not have to be considered, and the system has low-frequent permission granting operations, expanding an array by 1/2 or even 1/3 each time may work well, too. When the number of physical units reaches a threshold (32,758 in our current setting), we transform the block to its literal version and store the 95,296×11 explicit bits instead.

3.3.3 Permission Revocation

Revoking a permission on an object from a subject corresponds to clearing a specific bit in a specific unit. This operation is similar to setting a bit, except that new blocks and new units need not be created. On the other hand, a corresponding unit, and even a complete block, may become all-0 after clearing a bit. We can reclaim the space immediately or set the bit to 0 and collect empty units periodically offline. Using the latter approach, the total online time to revoke a permission is O(log N), where N is the number of physical units in the block.

3.3.4 Union of Permission Lists

To find the effective permissions for a subject, we need to find the union of permission lists for that subject and all his/her ancestor groups. The union of two permission lists requires the union of two lists of blocks, and since our scheme is block-aligned (i.e., the start of every block corresponds to a logical unit numbered by a multiple of 95,296), the operation is essentially a series of block merges.

When two blocks from the two permission lists have the same offset, they cover the same logical range. If both blocks are in their literal version, we merely *or* the bits; if only one block is in its literal version, we iteratively *or* in the bits from all units located within the BOP block. Suppose both blocks are in BOP format; we then need to merge the units

(permission words) in the two blocks into one list, possibly transforming the result to its literal version if the number of units reaches the threshold. If corresponding units appear in both blocks, we merge them by computing the *or* of the p most significant bits of the two words and copying the remainder (offset) bits. We expect most participating blocks are in BOP format because most permission lists are sparse. The total time is O(N×B), where N is the number of units in a block and B is the number of blocks used for a permission list.

3.3.5 Intersection of Permission Lists

Intersection of permission lists is not really necessary for our system; however, we still implement it in case we have queries like 'tell me the common permissions of permission list A and permission list B' in the future. Intersection is similar to union, but rather that creating additional blocks, the merged permission list might have fewer non-empty units and fewer blocks than either input. Because the result must be a subset of both permission lists, an all-0 block or unit may result from intersecting any block or unit with one that is all-0, and all-0 blocks or units may also result from intersecting blocks or units that are neither all-0. Furthermore, the result of intersecting a BOP block and a literal block is either all-0 or a BOP block: for each bit set to 1 in the literal block, we clear the corresponding bit in the BOP block.

3.3.6 Browsing-Oriented Query

Defined in Section 1.1, access control requests of this type have never been carefully considered or optimized. They are, however, probably the most common queries in an ECM system's access control sub-system. A user issues a browsing-oriented query implicitly whenever he/she opens a compound object (i.e., click to enter a folder). For example, given the object hierarchy shown in Figure 3.3 (please recall that permissions are not propagated through the object hierarchy), even if a user exactly knows where the particular object he/she is interested in (for example, O9) is located, he/she will probably go into folder O1 (resulting in a browsing-oriented query), then into folder O2, then continue to O5, and finally explicitly request to access to O9. In this example, we have 3 browsing-oriented queries while there is only one query to check an individual permission. This means that there might be even more browsing-oriented queries than individual permission checking queries. Therefore, we design our data structure to optimize for browsing-oriented queries.

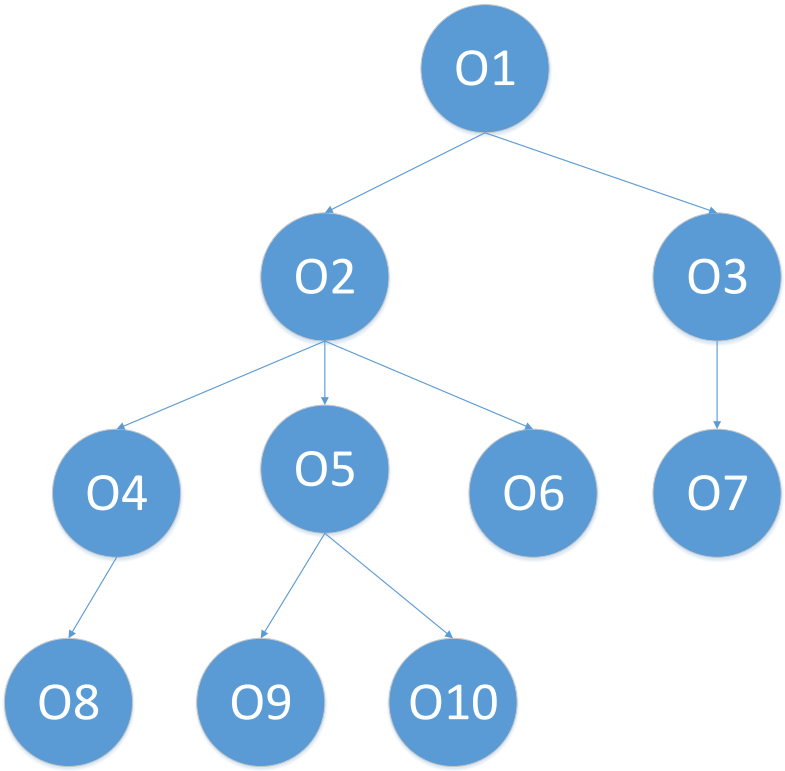

Figure 3.3: A simplified object hierarchy in an ECM system

If the objects are numbered randomly (or in creation order), a browsing-oriented query will have to be treated as a collection of multiple individual permission checking queries, and there is quite limited optimization we can do. However, motivated by document reordering in IR systems (the documents with the same indexed content will then have close or even contiguous IDs) [8, 31], we notice that the efficiency of answering a browsing-oriented query can also benefit from having a list of contiguous object IDs to be checked for a query. Specifically, we can apply breadth-first numbering, and then the object IDs under their parent object are contiguous. In this case, when a browsing-oriented query is issued (having to check for a single list of contiguous objects), we first calculate whether all objects we are checking for are in the same block. If all objects belong to the same block (which is the majority case), we search for the permission word of the first object we need to check using binary search. Either the specific one (we find the one we are looking for) or the next physical permission word (we do not find the permission word of the first object) is returned. This takes O(log N) where N is the number of physical permission

units. Then we check the following physical permission words starting from the returned one, until we reach a permission word whose associated object ID is bigger than the ID of the last one in our checking list. This costs O(M) time, where M is the number of objects under a same object (folder).

If the objects are in two blocks, we check the ending physical permission words of the first block backward until we reach a physical unit whose associated object ID is smaller than the first object ID we are checking for, and we check the starting permission words forward until we reach a physical permission word whose associated object ID is bigger than the last object ID we are checking for. The frequency of this case is not expected to be high since the logical block size is much bigger than the number of objects under a folder.

Since new objects are created over time, we may have to insert a new object under a folder, which will damage the contiguity of the IDs under the folder. A feasible approach is to periodically re-number the IDs. However, at an arbitrary time, we may find that there are several objects with contiguous IDs while one or more objects have arbitrary IDs. This requires us to modify the above algorithm. The above algorithm can answer a browsing-oriented query with a single list of contiguous object IDs as input; with random IDs, a browsing-oriented query may encounter multiple lists of contiguous IDs. Thus we apply the above algorithm for every list encountered by a browsing-oriented query, and then union the returned results. Fortunately, we can safely assume that most IDs are contiguous.

Chapter 4

Alternative Implementations

We present two alternative approaches to implementing an access control system in this chapter.

4.1 Implementation Based on Hash Table

We may implement an access control matrix using a stored set of subject-object-permission triplets (see Section 2.2.3); however, we believe that implementing a subject-oriented system is more reasonable based on the query characteristics (recall Section 3.1). Therefore, instead of storing an access control matrix using a single hash table, we create a hash table for each permission list.

4.1.1 Hash Table

In our implementation, the object IDs are the keys and the 11-bit permission units are the values. We only have to consider non-zero units. Since which keys will appear in a permission list is unpredictable, we cannot find a perfect hash function.

We choose *separate chaining with linked lists* as our collision resolution strategy (see Figure 4.1). However, we do not wish to use 64 bits for each pointer that connects two entries. Considering we can fit a key and a value into 34 bits, each pointer would consume twice the space of a key and a value. We therefore implement a hash table with short pointers in order to improve the space efficiency.

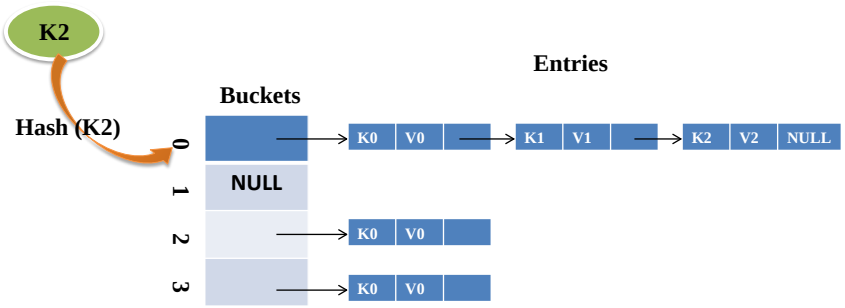

Figure 4.1: Logical hash table example

We use two arrays (see Figure 4.2) in our implementation. The first array, Bucket Array, stores the pointers to the entries in the second array (Entry Array). Specifically, each pointer points to the first entry of the linked list whose keys are hashed to the bucket. The Entry Array stores all entries of a hash table. Each entry, in our implementation, contains KEY, VALUE, and a pointer NEXT pointing to the next entry of the linked list of its associated bucket. We use 26 bits for a key, 11 bits for a value, and 27 bits for a NEXT pointer. It is, in fact, possible to use fewer bits for a key and NEXT pointer since we will have about 8 million (about 2^{23}) objects in total. However, we can make it word-aligned by using a few more bits since each entry now consumes a 64-bit word. We use 32 bits for every pointer in the Bucket Array so that they are also word-aligned.

Let N be the number of allocated entries and K be the number of buckets. N/K is then the *Load Factor*. We pick LoadFactor = 2 to balance the speed and space.

In fact, the values of N and K are determined by the number of entries needed to be stored in a hash table (represented by M). N is a power of 2. N needs to be equal to or bigger than M, but it is strictly smaller than 2×M. For example, if we have M = 12, we then know that N is 16 and K is 8 (16/2).

4.1.2 Hash Function

We start by noticing that we do not want an expensive-to-compute hash function in our application; for example, hash functions used in cryptography such as MD5 or SHA1 are inappropriate.

Our hash function is simple (thus inexpensive to compute for any key), yet effective in randomizing the hash values of the keys [19]. Given a key, we first multiply it with a big

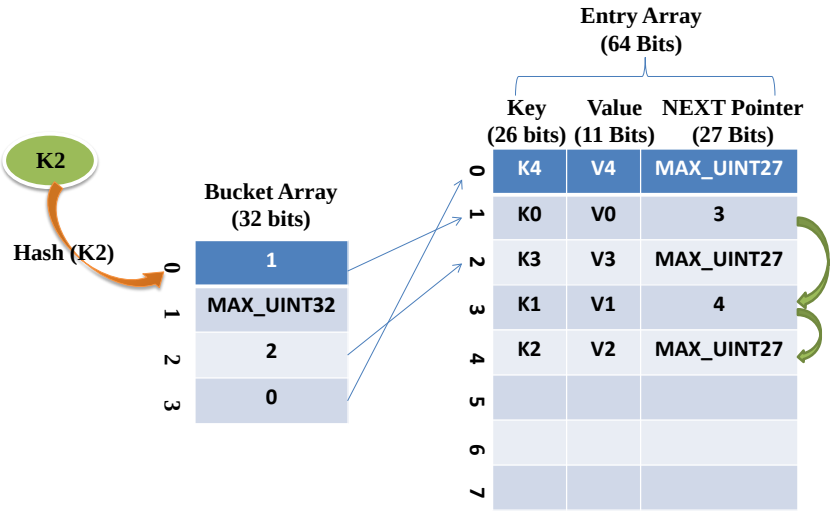

Figure 4.2: Implemented hash table example

prime. The prime has to be big enough so that the product will consume all 64 bits of a word. We fetch the middle 24 bits of the product by getting rid of the 12 most significant bits and 28 least significant bits (the middle 24 bits will now be 24 least significant bits). Finally, we have the previous result *mod* the current number of buckets. The hash function we use is formalized in Equation 4.1.

$$Hash(Key) = ((Key \times BigPrime) << 12 >> 40)\%K, \text{ where K is the number of buckets} \tag{4.1}$$

4.1.3 Access Control Operations

This implementation is capable of doing all necessary operations in our access control system.

Individual Permission Checking

To check an individual permission on a permission list, we first compute the hash value of the key (object ID), and then search the corresponding linked list for the entry. We check the corresponding bit if the entry exists.

Permission Granting

We first search for the entry with the same key. If we find such an entry, we set the corresponding bit to one in the entry; otherwise, a new entry will be added to the linked list. The object ID is the key of the entry, and the corresponding bit is set to one reflecting the permission has been granted. Physically, to add the entry to our hash table, we add it to the end of the Entry Array, modify the NEXT pointer of previous entry of its linked list to the new entry, and set the NEXT point of the new entry to be MAX_UINT27.

Hash Table Extension

We may have to expand the hash table when we add a new entry to a full hash table (N = P). Again, we choose a buddy system for table extension (doubling both the Bucket Array and Entry Array). All entries will be moved to the new hash table, and the new entry will be added to the new hash table.

Permission Revocation

This is similar to permission granting, except that 1) we will never have to add new entries, and 2) we may get an entry having a value of 0. We choose not to reclaim the space occupied by such entries in real time.

Permission List Union

To union two hash tables, we first make a copy of the bigger hash table (with more used entries), and then check for every entry in the smaller (with fewer used entries) hash table whether a corresponding entry exists in the bigger hash table. If a corresponding one exists, we simply union the 11-bit value without making any change to the entry's key or NEXT pointer in the bigger hash table; otherwise we add the entry to the bigger hash table as a new entry. This operation may trigger at most 1 hash table extension.

Permission List Intersection

Similarly, we make a copy of the smaller hash table, and then check for every entry in the smaller hash table whether a corresponding entry exists in the bigger hash table. We remove the entry if no corresponding one is found; otherwise, we intersect the 11-bit values. This operation would not cause any hash table extension.

Browsing-oriented Query

Unfortunately, we have to break down a browsing-oriented query into several individual permission checking queries since two contiguous keys might be hashed to any two buckets.

4.2 Implementation Based on WAH

A permission list in its literal version is a bitmap. Intuitively, we can compress these permission bitmaps in order to get a space-efficient implementation, and then carry out the necessary access control operations on the compressed bitmaps. We implemented a mainstream bitmap compression scheme called WAH (see Section 2.4.1 for the description of WAH) for our access control system.

4.2.1 Access Control Operations

WAH is not optimized for dynamic bitmaps. Therefore, operations like setting/clearing a bit in a compressed bitmap (referring to granting/revoking a permission) have never been implemented and discussed. In this case, we implemented all necessary operations to support an access control system. We implemented a 64-bit version of WAH; thus the number of bits contained in a WAH word is a multiple of 63 (64 - 1 since the most significant bit is a flag denoting whether it is a fill word or a literal word).

Individual Permission Checking

This operation is to check a specific bit on a bitmap. Since each WAH word contains various number of bits, we have to linearly search the WAH words one by one from the beginning of the compressed bitmap in order to find the WAH word containing the specific bit. If the WAH word we find is a 1-fill, the value of the bit we are checking is 1; if it is a 0-fill, the value of the bit is 0; if it is a literal word, we use bit manipulation techniques to check the specific bit.

Permission Granting

This operation corresponds to setting a bit to 1 on a bitmap. The first step is to find the WAH word containing the bit whose value we want to set to 1. If a 1-fill word is returned,

nothing needs to be continued since the bit has been set to 1. If we get a literal word, we set the corresponding bit to be 1 in the word. Setting a bit to 1 in a literal word may form many contiguous 1s; for example, there could be many contiguous 1s before and after a 0, and we are changing the 0 to 1. Potentially, this literal word can be combined with its neighbor(s); we, however, still keep it literal. It is more complicated in case we find a 0-fill. A 0-fill will have to be split into up to three words (two literal words, a literal word followed by a 0-fill word, a 0-fill word followed by a literal word, or a 0-fill word followed by a literal word followed by another 0-fill word), indicating that we will have to update one WAH word and insert another one or two words into the compressed bitmap (which is an array holding all WAH words).

Permission Revocation

Revoking a permission from a permission list is equivalent to clearing a bit in a bitmap. Also, we have to first find the WAH word containing the specific bit, and we may get a 0-fill word, a literal word, or a 1-fill. Finding a 0-fill indicates that the bit we want to clear has been set to 0. We clear the corresponding literal bit if a literal word is found, and we keep it literal without checking whether it is combinable with its neighbor(s). To clear a bit in a 1-fill word, we have break it down to up to 3 words (two literal words, a literal word followed by a 1-fill word, a 1-fill word followed by a literal word, or a 1-fill word followed by a literal word followed by another 1-fill word). New WAH word(s) will have to be inserted for this case; however, the number of 1-fill words is quite limited in sparse bitmaps (close to zero in our application).

Permission List Union

This operation is essentially the merge of words from two compressed permission lists. We run linearly through two lists. When two physical words contain non-equal logical bits, we break the longer one into two parts. The first part will union with the shorter word from the other list (making them logically aligned), and the rest will union with the next word from the other list. When reaching the end of any list, we directly copy the remaining words from the unfinished list.

Permission List Intersection

This operation is very similar to union with the exception that we can simply abandon the remaining words from the logically longer permission list when reaching the end of the

logically shorter one.

Chapter 5

Performance Evaluation

In this chapter, we compare a hash table, WAH, and BOP for access control system implementation. We first describe the data we use for testing, including the synthetic data and the real-world data. We then present and discuss the evaluation results for space consumption and execution speed.

All experiments have been performed on an AMD FX-6100 six-core 3.3Ghz Processor (8M L3, 6M L2 and 64K + 64K L1 caches) with 32GB of memory, running 64-bit Windows 7 Professional with single thread execution of code compiled using *gcc*. All experiments are conducted on permission lists stored in main memory, and therefore no I/O is involved.

5.1 Experimental Data

5.1.1 Synthetic Data

Each permission list we generate contains 100M literal bits (9,090,909 logical objects). All of them are sparse so that no blocks are stored in literal form. We first generate a permission list in its literal version (a bitmap), and we then transform it to the BOP format and to the hash table format, respectively. A permission list is generated by the following steps:

1. Pick a random starting position s, where s is a multiple of 11;
2. Randomly set 60 % of the bits within the range specified by [s, s + 10];

3. Repeat 1 and 2 until 60,000 bits have been set.

4. Convert the literal bitmap to a BOP list and a hash table

It is also important to mention that we allocate four 32-bit words for a BOP block instead of only one when a new non-zero unit is inserted into an empty BOP block. This will consume a little extra space, but it can reduce the frequency of array extension.

Since a permission list is equivalent to a bitmap, we do not distinguish a permission list and a bitmap in the rest of this thesis.

5.1.2 Real-world Data

The real-world dataset is provided by a mainstream ECM system vendor. The dataset contains around 8 million objects and 6,000 active subjects (users and groups).

Among the 6,000 subjects, we have nearly 900 groups and 5,100 individual users. A user can be a member of one or more groups; a group can also be a member of one or more groups. Every subject other than the root is a direct member of the root subject.

In an ECM system, a query is always issued by an individual user. Since when a query is issued, we have to check the permission lists of the issuer (a user) and its ancestors, it is important to understand how many permission lists we have to check for a query in practice. The numbers of ancestors vary between 2 and 110 for the individual users. Figure 5.1 shows more details about the percentages of individual users w.r.t the number of their ancestors. On average, a user has 8.8 ancestors; thus we will have to check 9.8 permission lists for a query on average.

Because subjects can belong to more than one group, not all ancestors appear along one chain. Thus, we measure the maximum distance from each individual user (sink node) to the root subject. This helps us understand how many recursive steps would be needed to compute the transitive closure. Please notice that counting minimum distance is meaningless in our application since it is always 1 for any individual user. In addition, there are no cycles in the directed graph representing the subject hierarchy. Figure 5.2 illustrates the maximum distances for the subjects. On average, the maximum distance between an individual user and the root subject is 5.62.

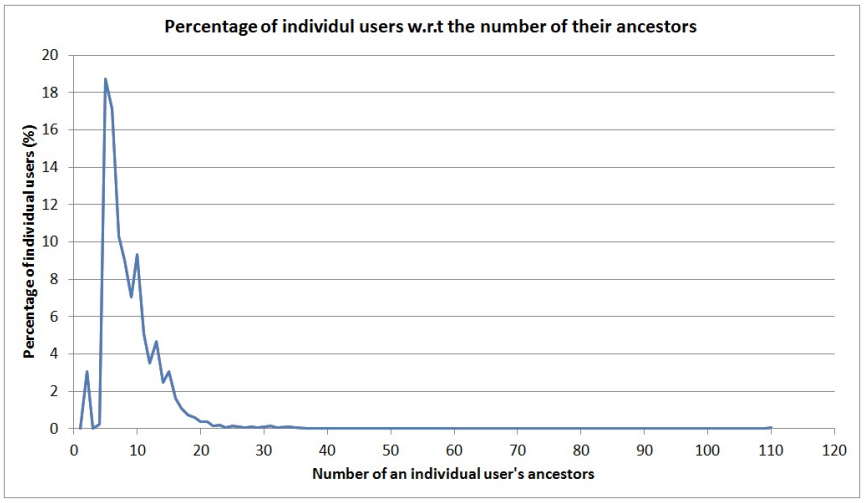

Figure 5.1: Percentage of individual users w.r.t the number of their ancestors

5.2 Performance Evaluation

5.2.1 Space Complexity

Before reporting the sizes of the data structures witnessed in the experiments, we examine the expected space requirements in general. It is straightforward to theoretically compare the space complexities between BOP and our hash table. Since we implement a buddy system for both BOP and our hash table, both data structures contain some unused space. For simplicity, we do not consider the allocated unused space now. BOP consumes 32 bits for each non-zero permission unit; assuming that there are M non-zero units, we need $32 \times M$ bits in total. Certainly, there are some pointers and other meta data, but this needs very little space. In contrast, our hash table needs 64 bits for an entry (a non-zero unit); additionally, each element in the Bucket Array consumes 32 bits, and the number of elements in the Bucket array is over 1/2 of M. Therefore, we need at least $64 \times M + 32 \times M/2 = 80 \times M$ bits in total (meta data is ignored as well). This indicates that our hash table consumes approximately 2.5 times the space compared to BOP.

Moreover, BOP can also be considered as a bitmap compression scheme. Compression ratio (Equation 5.1) is often cited by researchers in the DB community to compare the space requirements for compression schemes. In the context of IR, however, researchers often evaluate the space complexity by considering the average number of bits necessary to encode a d-gap, which measures the expected number of bits required to represent each 1

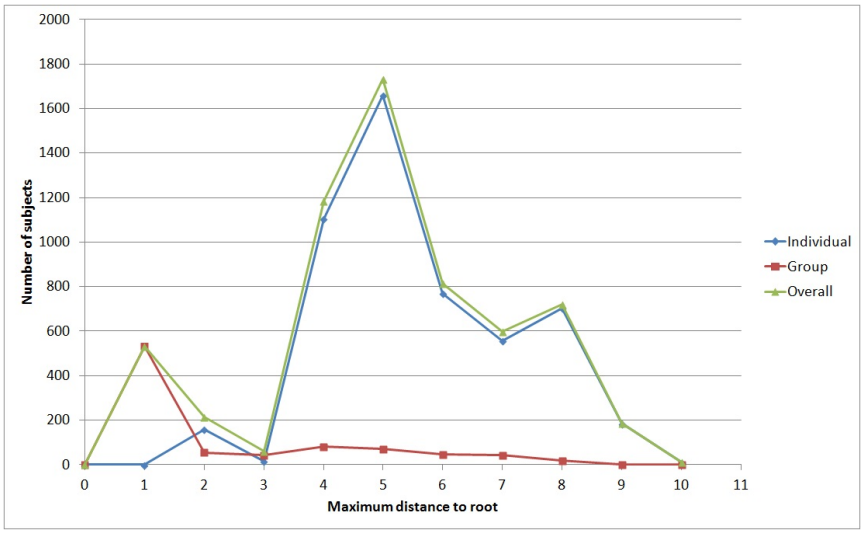

Figure 5.2: Maximum distance from a subject to the root

in the equivalent bitmap. In keeping with both traditions, we compare BOP against WAH using the traditional compression ratio but use the latter method to compare BOP against the vbyte and Simple-9 schemes developed by IR researchers.

A theoretical comparison among BOP, vbyte, and Simple-9 is presented in Table 5.1. BOP requires 32 bits for each non-empty unit, and there are on average 5.96 ones in each such unit. Therefore BOP uses on average 5.37 bits to represent each bit set to 1, whereas vbyte require at least 8 bits for each bit having the value 1. Thus in the prototypical application, BOP outperforms the best cases possible for vbyte.

We also have a relatively accurate estimate for Simple-9. Considering a sparse bitmap, on average, there is one d-gap with a relatively large value (on average around 8,000 which is the avarage distance between two neighboring permission units), and there are 4.96 d-gaps with small values in a non-empty unit. To encode a value around 8,000 we need 13 bits, and Table 2.5 shows that when using Simple-9, such a value can only be represented in a word that encodes two d-gaps. Therefore, to encode a typical non-empty unit requires two words: one to encode the position of the first two bits in the unit and the other to store the remaining ones. Therefore, on average, Simple-9 uses 64 bits (two 32-bit words) to represent 5.96 ones (in a single non-empty unit) in our prototypical application, which means each one (d-gap) requires 10.73 bits.

$$\text{Compression Ratio} = \frac{|\text{decompressed data}|}{|\text{compressed data}|} \tag{5.1}$$

Table 5.1: Comparison of sizes between BOP and the competitive IR compression schemes (measured in expected number of bits per 1 (d-gap))

	BOP	vbyte	Simple-9
Best Case	2.91	8	1.14
Worst Case	32	N/A	32
Sample Case	5.37		approximately 10.73

We then directly applied BOP, our hash table, and WAH to a synthetic permission list (bitmap) and the permission lists in an actual ECM access control system. Table 5.2 shows the results. We implemented a buddy system for BOP and our hash table; for WAH, we used a *vector* provided by C++ (extending the size by half every time). Therefore all implementations included some pre-allocated unused space. There are a few dense blocks in a small percentage of the permission lists in the real-world ECM system, and they are stored in literal version even when using BOP implementation.

Table 5.2: Comparison of sizes among BOP, our hash table, and WAH (the synthetic data includes a single bitmap which is 11.92 MB before compression; the real-world data contains nearly 6,000 bitmaps, and the total size before compression is 56.27 GB)

	BOP	Hash Table	WAH
Synthetic	59.8 KB	163.8 KB	213.4 KB
Compression ratio (synthetic)	204.1	74.5	57.2
Real-word	0.36 GB	0.92 GB	1.1 GB
Compression ratio (real-world)	156.3	61.2	51.2

The results show that all three implementations are much more space-efficient than storing literal bits, among which BOP is the best. Additionally, the experimental results for BOP and our hash table are very close to the theoretical analysis (theoretically, our hash table consumes 2.5 times more space than BOP).

5.2.2 Execution Speed

Individual Permission Checking

We first tested this operation on a synthetic permission list generated using the approach described in Section 5.1.1. Specifically, we randomly picked 50,000 to 500,000 individual permissions (a permission refers to a specific bit in a bitmap, and it refers to an object-permission pair for a hash table) and checked their values on a permission list represented/compressed by BOP, our hash table or WAH. The results are presented in Figure 5.3.

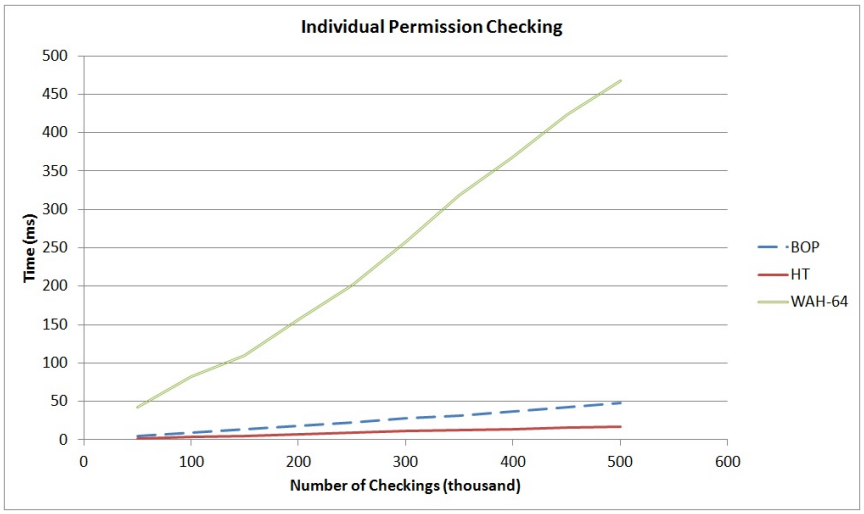

Figure 5.3: Experimental results for individual permission checking on the synthetic data

As expected, the time for all data structures increases linearly when we increase the number of checking times. Hashing is about 2.6 to 2.7 times faster than BOP. That is because the time complexity of searching for an object using our hash table is O(1) while BOP requires O(1) for locating the block plus O(log N) for finding the object, where N is the number of non-zero units in the block holding the desired object (recall Section 3.3.1).

Additionally, BOP is approximately 10 times faster than WAH when applied to a synthetic bitmap. Instead of taking advantage of hashing, or the block index and binary search for efficient search, we have to interpret the WAH words one by one from the beginning of the compressed bitmap until we find the WAH word containing the bit we are searching for. Obviously, this linear search with WAH is far more expensive than the search with our hash table or BOP.

BOP, our hash table, and WAH were then applied to our real-world data. We tested this operation 50,000 times, for each of which we randomly chose a bitmap and checked the value of a random bit. Table 5.3 shows the experimental results. We notice that the performance advantage of BOP over WAH is significantly amplified for the real-world data. There are two reasons. First, BOP is more space efficient, which indicates less cache loading time. This is particularly important when there are 6,000 real-world bitmaps instead of only a single synthetic bitmap. In fact, the space efficiency factor also amplifies our hash table's performance advantage over WAH and narrows the performance difference between BOP and our hash table. Second, although the average density of the real-world bitmaps is close to the density of the synthetic ones, the densities of the real-world bitmaps, in fact, vary in a large range, and this characteristics amplifies the performance advantage of binary search over linear search. Due to these two reasons, BOP is almost 100 times faster than WAH when applied to our real-world data.

Table 5.3: Time needed to check, grant, or revoke 50,000 random permissions on the real-world dataset (ms).

	BOP	Hash Table	WAH
Checking	18.6	10	1945
Granting	38.3	16.2	2629
Revocation	18.7	9.9	1987

Permission Granting

We generated a permission list and then randomly granted 50,000 to 500,000 individual permissions to the permission list. We might grant a permission that had been granted. This operation is similar to the individual permission checking with the exception that a new entry (for hash table), a new permission word (for BOP), or up to two WAH words may have to be inserted. Figure 5.4 shows the experimental results.

It is more expensive to use BOP than our hash table for this operation, too. Besides more searching time as for individual permission checking, another reason is that BOP may have to insert a new permission word into the array holding the block or a brand new block when necessary (which is more expensive than inserting a new entry into a hash table unless an extension is needed for the hash table). In addition, all physical blocks in a permission list are organized as a linked list, and maintaining such a list costs extra time when we are inserting a new block.

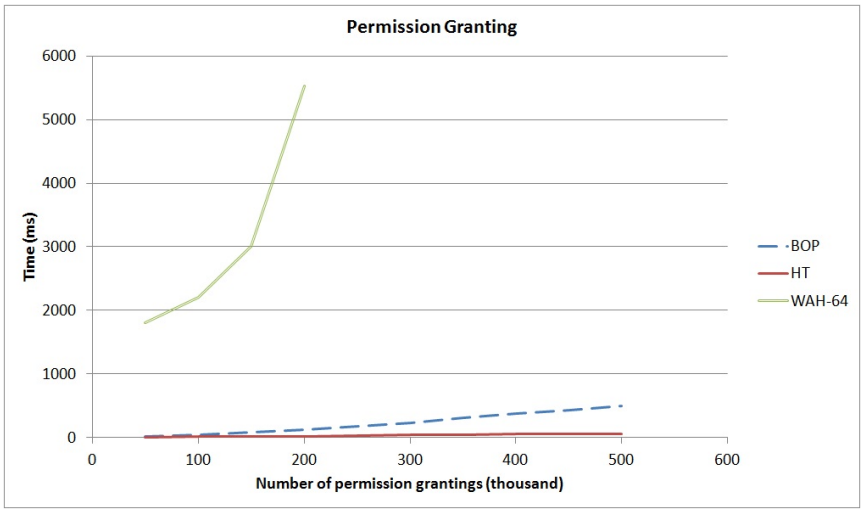

Figure 5.4: Experimental results for permission granting on the synthetic data

Furthermore, the cost for our hash table goes up almost linearly as the increase of the number of permission grantings (but not perfectly linearly due to the hash table extension when necessary). However, the cost for BOP obviously increases super-linearly, which is due to 1) the arrays holding the permission words are becoming larger and larger, and 2) inserting an element into a larger array is more expensive than inserting an element into a smaller array (more expensive to shift elements and more expensive to relocate the array when needed).

The cost for using WAH soars as the number of permission grantings increases. This is due to the quick increase in size of the compressed bitmap after permission grantings, and therefore both search and new WAH word insertion becomes more and more expensive.

Again, we tested this operation on the real-world data, and Table 5.3 includes the results of the experiments. Similar to individual permission checking, the performance advantages of BOP and our hash table over WAH are amplified even with a small number of grantings when applied to the real-world data, and the performance difference between BOP and the hash table is relatively small.

Permission Revocation

Figure 5.5 presents the results for permission revocation on a synthetic permission list using our hash table, BOP, and WAH, respectively. Table 5.3 includes the results for the

tests on the real-world data. The operation is similar to the above two operations; thus it is not surprising that BOP is a little more expensive than our hash table, and both BOP and our hash table significantly outperform WAH.

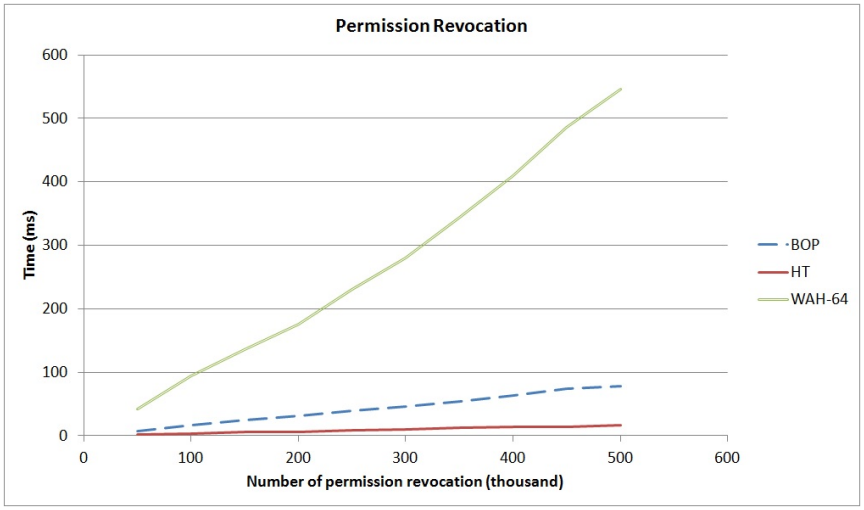

Figure 5.5: Experimental results for permission revocation on synthetic data

It is worth noting that after revoking a permission from a permission list when using BOP, we may get a permission word containing an all-0 unit; we may even get an empty block. We implemented a garbage collection mechanism to reclaim such space. Specifically, if we encounter an empty block after revocation, we remove the block completely. If we get a permission word with an all-0 unit, we remove this word and shift the remaining elements in the array containing all permission words of the specific block (thus all unused words will always be at the tail of the array). However, we do not shrink the array size during this operation (therefore no array relocation). This explains why permission revocation is more expensive than individual permission checking when using BOP (the difference can be very small since we do not encounter many all-0 units in our tests when we revoke just one permission each time).

We did not recycle garbage for our hash table since it is expensive to do it (we will have to update all pointers pointing to all entries shifted) (please recall our hash table implementation presented in Figure 4.2).

For WAH, a word may become combinable with its neighbors. For example, suppose that we have a 0-fill followed by a literal word in which only one bit is set to 1. After we revoke the permission, we get a literal word in which every bit is 0; thus this literal word

can combine with the 0-fill, forming a new 0-fill containing more 0s. We, however, did not do this in our implementation since it requires array shift (and potentially array shrink). This is the reason why revoking a permission is only a little more expensive than checking a permission using WAH.

Permission List Union

We had two categories of experiments for this operation. In the first category, we generated 100 permission lists using the approach described in Section 5.1.1. Then we performed 100 to 500 union operations, for each of which we randomly chose two permission lists. Figure 5.6 shows the test results. We find that the results for all data structures go up linearly but BOP is more efficient than the other two (approximately 5.2 times and 6 times faster, respectively). Unlike the above three operations, because for every entry in the smaller hash table, we have to one by one either insert it into the bigger one or merge it with an entry in the bigger one (searching followed by insertion or merge), our hash table barely outperforms WAH.

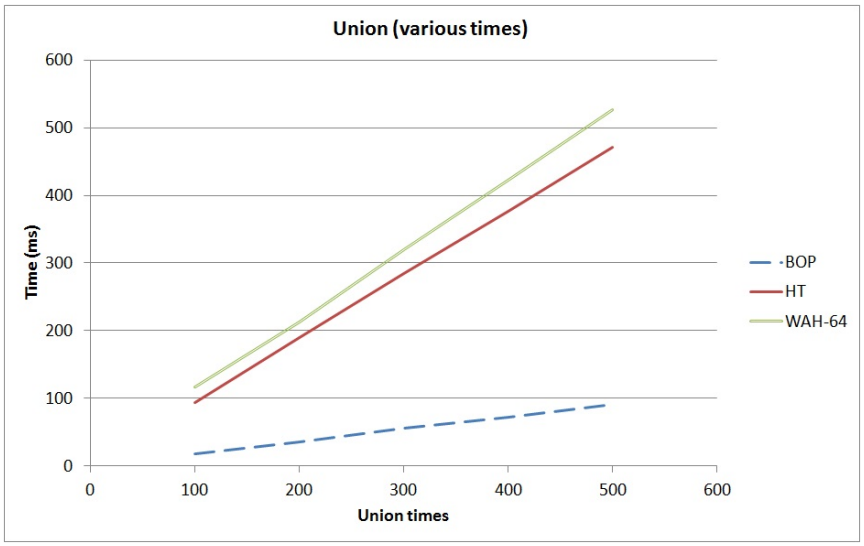

Figure 5.6: Union between two permission lists (various times)

In the second category of the tests, we varied the logical length of the permission lists, but we fixed the density of the units/bits ($1.1 \times 10^{-3}/6 \times 10^{-4}$). This is done by increasing the logical length and the total number of bits to be set proportionally. The results are presented in Figure 5.7

It is interesting to notice that the cost for our hash table increases faster for some settings than for others. In fact, this is not determined by the logical length but the total number of used entries. For each union, we first make a copy of the bigger hash table. As we insert the entries from the smaller hash table to the copy of the bigger hash table, an extension is necessary sometimes but not always. For example, if we have two hash tables, each of which has 9 entries, the chance to require an extension is relatively small since the pre-allocated hash table is big enough as long as we have two entry merges (16 entries after two merges); the situation may not change when the number of each hash table's entries goes up to 10 (i.e., three merges); however, when the number increases to 15, it is very likely that we need an extension, which causes the fluctuation of the slope of line for our hash table. For BOP, however, the increase of the logical length (permission words) simply increases the number of the blocks we need to merge without making the blocks larger or denser.

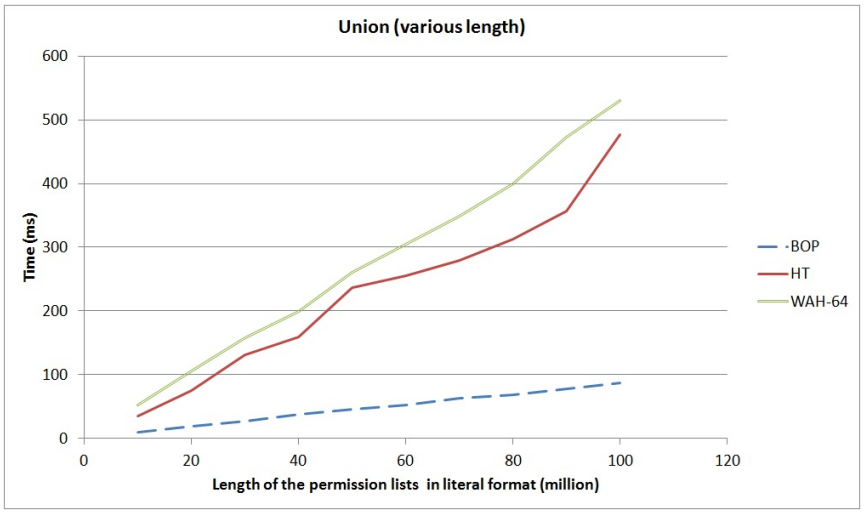

Figure 5.7: 500 unions between two permission lists (various length; fixed density)

When testing union on our real-world data, each test is to pick two random permission lists and union them. We counted the necessary time for 500 unions. Table 5.4 presents the results.

Permission List Intersection

Figure 5.8 and Figure 5.9 present the results for intersection on the synthetic permission lists. The results for real-world data is presented in Table 5.4.

Table 5.4: Time needed for 500 unions or intersections using BOP, our hash table, and WAH on the real-world data (ms)

	BOP	Hash Table	WAH
Union	72	190	368
Intersection	24	33	308

The results for intersection are very close to those for union with two exceptions. First, intersection is cheaper using whichever data structure (the result of an intersection will never contain more entries/permission words than the smaller participant). Second, the line for our hash table goes up more smoothly and the performance difference between our hash table and WAH is widened since this operation requires no hash table extension at all.

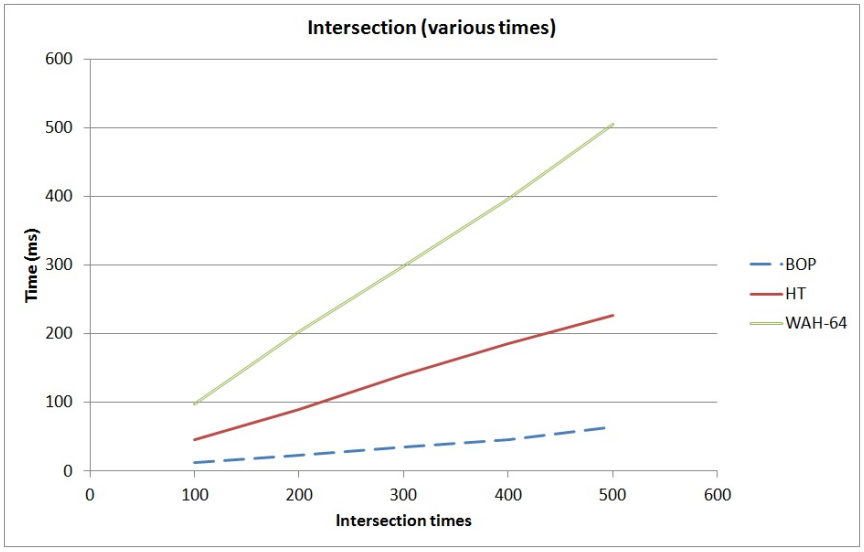

Figure 5.8: Intersection between two permission lists (various times)

Browsing-oriented Query

For BOP and WAH, the first step is always to search for the first object given a list of contiguous objects. Since WAH is far much slower for this step than BOP (in fact, we also notice that the cost of checking for a list of contiguous objects is mainly from checking for

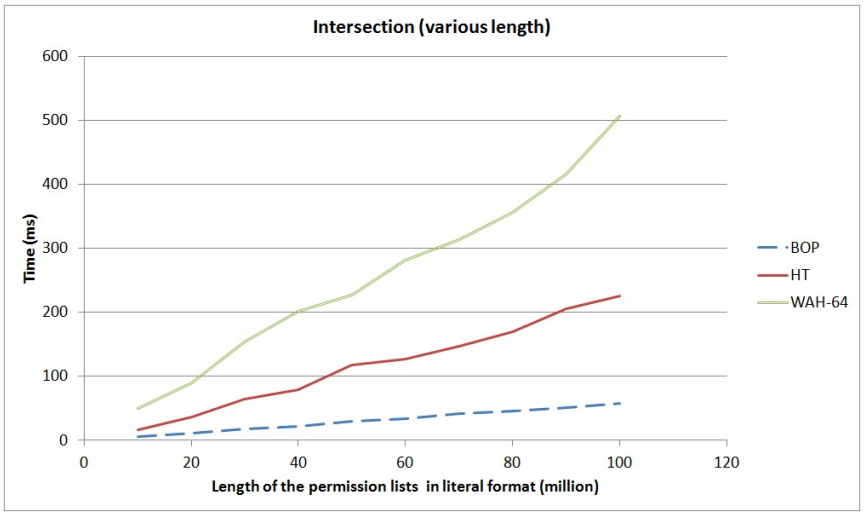

Figure 5.9: 500 intersections between two permission lists (various length; fixed density)

the first one), it is quite obvious that checking for a list of contiguous objects is also slower using WAH. We focused on BOP and our hash table for this test.

Although checking an individual permission is more efficient using our hash table, a major drawback of a hash table is that it is poor for iterating over non-zero values in sorted order. Assuming that breadth-first numbering is used for objects, we will have to check multiple objects with contiguous IDs for a browsing-oriented query. In addition, new objects may be created under an object (e.g., a folder), so there may exist a small number of objects with random IDs under one object. We simulated this in our experiments by including R ($0 \leq R \leq 3$) random IDs in each browsing-oriented query. We tested the performances when there were various contiguous IDs to be checked in a query (again, with R random IDs). Figure 5.10 shows the results.

All BOP lines go up very slowly, and the cost almost doubles when the number of random IDs doubles, indicating that the major cost is to find the first object we need to check when using BOP. In contrast, the cost for our hash table increases more sharply as the number of objects to be checked in a query increases. The number of random IDs will not affect the hash table performance since a hash table has to treat every object as a random one anyway. We therefore did not include any random IDs for the hash table tests.

Additionally, as expected, when the percentage of random IDs is small in a query, BOP is more efficient than a hash table. For example, the line BOP(3) indicates that when the

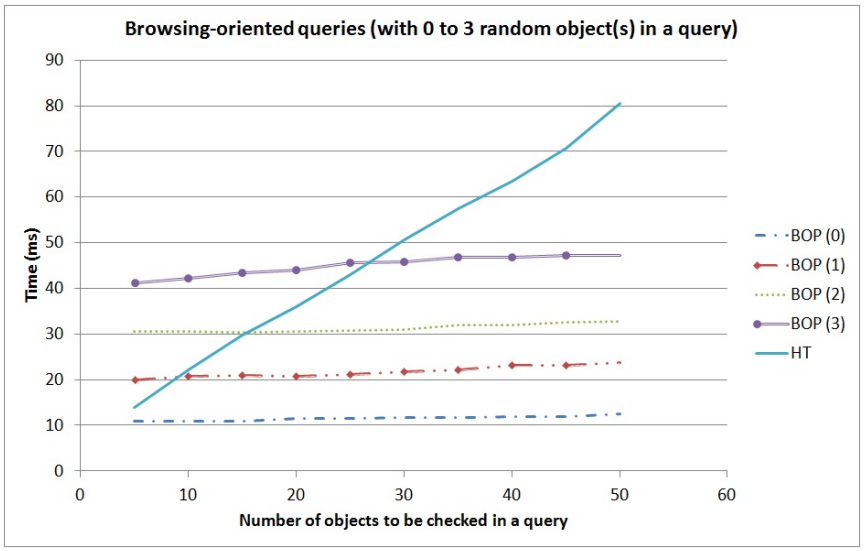

Figure 5.10: Experiment results for browsing-oriented queries on the synthetic data using BOP and our hash table, respectively. A line labeled BOP (R) is for the tests with various contiguous IDs and R random IDs in each test while using BOP.

percentage of random IDs is lower than 11% (3/26), BOP is more efficient. Note that we did not use percentage as a parameter but used a fixed number of random IDs for a BOP line, since both the number of IDs and the number of random IDs are too small to make meaningful claims when using percentages.

In order to test this operation on the real-world data, we applied breadth-first numbering to the objects. We then changed a few percentages of IDs to be random (2% to 10%) Then we tested 50,000 random browsing-oriented queries. The results are shown in Figure 5.11. The results show again that BOP is more efficient than our hash table when the percentage of random IDs is low. It is worth noting that most browsing-oriented queries encountered one list of contiguous objects when a small percentage of IDs are set to random (which is unlike the tests on the synthetic data).

In conclusion, BOP is faster if most IDs we have to check in a browsing-oriented query are contiguous by storing the permission words in order, and we can benefit from BOP if breadth-first numbering is used for objects in practice.

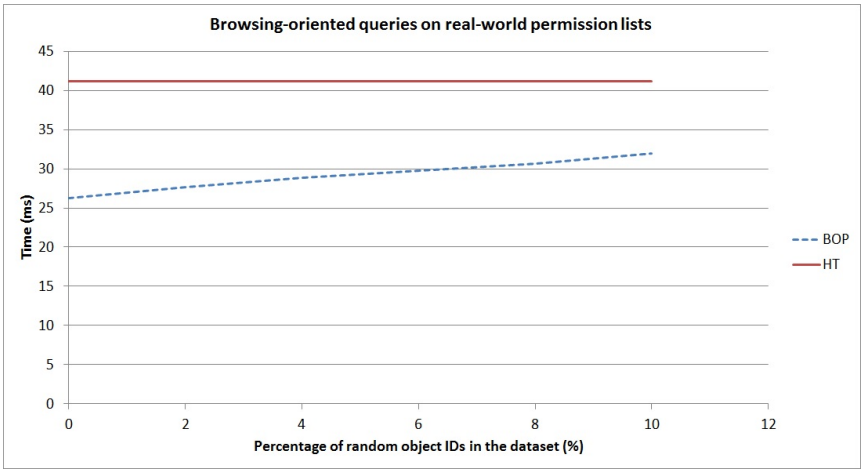

Figure 5.11: Experiment results for browsing-oriented queries on the real-world data using BOP and our hash table, respectively.

Comprehensive Workload

We designed four prototypical workloads in this section. In each one, 1) we have multiple types of queries, 2) we take the subject hierarchy into consideration (affecting individual permission checking and browsing-oriented queries since we have to check the permission lists of the issuer's ancestors, too), and 3) there includes 10% random IDs in the dataset. Four query sets were generated. Each query set includes 100,000 queries consisting of browsing-oriented queries, individual permission checking queries, permission granting queries, and permission revocation queries (see Table 5.5 for details).

Table 5.5: Query sets for comprehensive tests

	Browsing	Individual checking	Granting	Revocation
QS1	0.45	0.45	0.05	0.05
QS2	0.65	0.25	0.05	0.05
QS3	0.35	0.35	0.15	0.15
QS4	0.50	0.20	0.15	0.15

Table 5.6 presents the results of answering these query sets. We find that BOP outperforms our hash table for all query sets. Among the four query sets, QS1 and QS2 assume a less dynamic system, and they are more expensive to run because of the subject hierarchy (a browsing-oriented query or a individual permission query has to check, on average, 9.7

permission lists while we have to work on only a single permission list for a granting or revocation query). Additionally, BOP and our hash table produced very close execution time for QS3 which contains the lowest percentage of browsing-oriented queries. That proves that the performance advantage of BOP is from quicker execution of browsing-oriented queries, and it also indicates that a hash table will outperform BOP in a system with few browsing-oriented queries.

Table 5.6: Execution time for running comprehensive workloads (ms)

	BOP	Hash Table
QS1	1696	1778
QS2	1801	1927
QS3	1333	1397
QS4	1398	1519

Chapter 6

Conclusion and Future Work

6.1 Summary of the Thesis

Access control is critical for enterprise content management. Previous work was focused on access models with little consideration on the efficiency problem. Our thesis investigates the data structures for fast access control in an ECM system.

In this thesis, we surveyed ECM systems' data and query characteristics. Besides the characteristics that an ECM system usually has a complicated subject hierarchy and has many more subjects and objects, the most important characteristic is that there are many browsing-oriented queries in an ECM system.

We then presented a data structure specifically tailored to ECM systems by optimizing it for browsing-oriented queries. In our design, permissions are organized by subjects, and each subject's permission list is represented by our data structure called BOP. BOP first splits a permission list into multiple blocks with indexes to them, and then uses a 32-bit word to represent a non-zero permission unit (4 bits are unused). Permission units in a block are stored in order. How to carry out the necessary operations in an access control system was presented, too. Two alternative implementations were also discussed (using a hash table or a bitmap compression scheme called WAH).

To compare the three data structures, we generated a few synthetic permission lists; we presented the characteristics of the real-world data we got from a mainstream ECM vendor, too. We first evaluated their space efficiencies. We then carried out each operation on permission lists represented by BOP, our hash table, and WAH, respectively (except that we did not test browsing-oriented queries for WAH), and compared their execution

speed. We find that BOP is the most space efficient data structure for our access control system among the three (consuming only 36.5% and 28.0% of the space needed by our hash table and WAH, respectively, when applied to our real-world data). Additionally, the hash table outperforms the other two while checking an individual permission, granting a permission, and revoking a permission; however, BOP is the most efficient data structure for browsing-oriented queries when breadth-first numbering is applied to object IDs (even when there is a small percentage of random IDs). Through the comprehensive tests that contain all types of operations and take the subject hierarchy into consideration, we found that BOP resulted in the lowest overall cost (the performance advantage decreased as the percentage of the browsing-oriented queries went down).

In conclusion, BOP is scalable, space-efficient, and fast when applied to an ECM system, particularly when the percentage of browsing-oriented queries are relatively high.

6.2 Future Work

6.2.1 Object ID Renumbering

In our test, we, in fact, have renumbered the object IDs for tests on browsing-oriented queries in order to get contiguous IDs under an object. As the system runs, new objects may be created under an object, and an existing object may be moved to another folder. These operations increase the number of lists of contiguous IDs encountered by a browsing-oriented query, thus making BOP less efficient. Therefore, we plan to evaluate the cost of renumbering systematically so that we can decide how often we need to renumber object IDs.

6.2.2 Materialization of Effective Permission Lists

Another potential direction is to materialize the individual users' effective permission lists so that we have to check just one effective permission list when answering a browsing-oriented query or an individual permission checking query.

Essentially, this is similar to the materialized view problem widely discussed in the database community [22, 15, 33, 9]. There are two major problems regarding materialized views: selection and maintenance. With regard to view selection, we propose to materialize every active (logged in) individual user's effective permission list. To compute a user's effective permissions, we simply have to union the permission lists of his/her ancestors

and himself/herself. In the previous chapters of this thesis, we assume that the list of any user's ancestors is pre-stored; however, an access control system is dynamic so that a user may get a new ancestor or lose an ancestor during run time. Intuitively, we can maintain a transitive closure matrix for the subject hierarchy. Thus an efficient transitive closure maintenance algorithm is necessary.

If effective permission lists are materialized, they need to be updated when changes are made to the explicit permission lists (e.g., permission grant/ revocation, group membership update, etc.). The straightforward way is to do a complete refresh for each potentially affected user's materialized permission list (to re-union the explicit permission lists of the user and all its groups). However, this can be expensive, particularly if explicit permission lists and the subject hierarchy are relatively frequently updated. Therefore, we plan to develop algorithms for incrementally updating materialized effective permission lists.

Many incremental maintenance techniques that reduce the maintenance cost have been presented in the database community [27, 7, 33, 9]. The fundamental idea is to calculate what needs to be updated in the materialized view, given only the view definition, the current contents of the materialized view, and the update operation, and directly make changes to the materialized view. However, the view maintenance algorithms strongly depend on the operators used to define the views. Therefore, although the fundamental idea of incremental maintenance is crucial in our research, we have to develop our own application-specific algorithms.

The update operations we are considering include permission granting, permission revocation, and membership update. Basically, there are two steps to implement incremental update: find potentially-affected users, and determine which specific piece of data needs to be updated and how it needs to be updated in each potentially-affected materialized permission list. When an update operation is applied to an individual user, there is always only one potentially-affected user, and no special consideration is needed for the design of the incremental update algorithm. Additionally, we may conclude for a potentially-affected materialized permission list that, in fact, no update is required.

Among these update operations, permission granting is the easiest one with regard to incremental update of materialized permission lists. For example, if a new permission is granted to G3 (see Figure 6.1), we simply need to grant the same permission (set the corresponding bit to 1 in the corresponding object-permissions pair) to the individual members of G3 (U1, U2, and U3 in our case).

We will next consider the operation that a permission is revoked from a group. Potentially, every member in the group is affected. However, simply clearing the corresponding permission in the effective permission lists of potentially-affected users is inappropriate

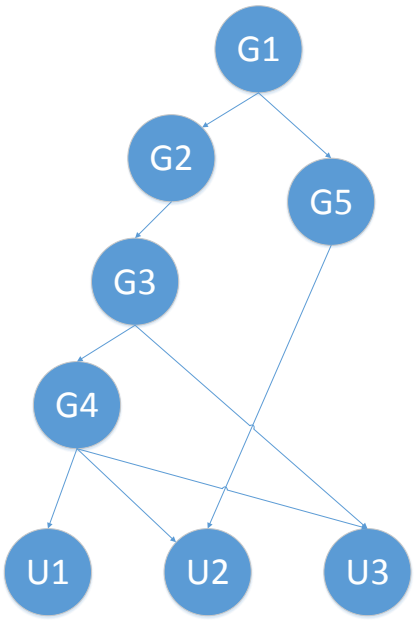

Figure 6.1: An example of a subject hierarchy

since a member may inherit the same permission from another group. We are going to develop an algorithm that can identify the set of materialized permission lists that do need an update. The algorithm may work like this: we first find the set of potentially-affected users (all active users who are members of the group from which the permission is removed); we then check for every one in the set whether the specific permission is explicitly granted to any of its other ancestors so that the user can inherit it. Given the fact that checking a specific permission (or even several permissions) is much faster than the union of two permission lists, we believe that this algorithm may outperform complete refresh.

We are also going to work on incremental update algorithms for membership update, including assigning a group to another group and removing a group from another group. For the former one, suppose G2 is a new member of G1. A potential algorithm is like this: find all members of G2; for each member of G2, we union its materialized effective permission list with the explicit permission lists of G1 and all its ancestors. This algorithm may, however, not be the optimal one. For example, if a potentially-affected user was a member of G1 before the operation, in fact, its materialized permission list does not need to be updated. We will improve our algorithm by taking these cases into consideration. For example, we can take advantage of the transitive closure matrix to reduce the set of potentially-affected users. Suppose SetDesG2 is the set of G2's descendants (which can

be derived from the transitive closure matrix very quickly), SetDesG1 is the set of G1's descendants, and SetActive is the set of active individual users. We can reduce the set of potentially-affected users from SetDesG2 to SetDesG2 ∩ SetActive - SetDesG1.

Removing a group from another group is the most complicated update operation with regard to materialized permission list maintenance. For example, if we remove a group G2 from another group G1, all permissions assigned to G1 and its ancestors (the set of effective permissions of G1, or SetEPG1) may have to be revoked from all members of G2; however, G2's members may inherit those permissions from other ancestors. We will have to check those many permissions for every member of G2, which may be even more expensive than a complete refresh. However, we can at least reduce the set of potentially-affected users by again using the transitive closure matrix (from SetDesG2 to SetDesG2 ∩ SetActive - SetDesG1). In fact, for incremental update, we may also reduce the number of permissions we need to check for a potentially-affected user. For example, if a direct member of G2 is also a direct member of G0 which is an ancestor of G1, the member will retain all effective permissions of G0 (SetEPG0). Thus we can reduce the set of permissions we have to check from SetEPG1 to SetEPG1 - SetEPG0.

It is also worth noting that a complete refresh does not have to be more expensive than the equivalent incremental update. Therefore, we must compare the performances between complete refresh and incremental update for every operation we consider.

6.2.3 Negative Authorizations

Negative permissions have been discussed in academic papers [6, 2], but they are not widely adopted in industry. In fact, there are also some arguments that using negative permissions is harmful [26]. That is because security administrators are more easily confused with complicated conflict resolution policies within a system; thus they are more error-prone even with a correctly-working conflict resolution algorithm. Recently, some health care systems have proposed to use negative permissions (called exceptions in their context) [17]; however, ECM systems have very limited support for negative permissions (notably SharePoint 2013 supports explicit 'Deny' that can override positive permissions [23]).

With negative permissions, there may be conflicts of permission assignments; thus conflict resolution is inevitable in a system supporting negative permissions. Chinaei et al. present a unified conflict resolution algorithm that can resolve conflicts within a system even it uses different conflict resolution policies under different conditions [10]. More commonly, however, people tend to use a uniform and simple conflict policy across the whole system (often negative permissions take precedence). The major benefit of this approach is that

it can ease the difficulty of conflict resolution. However, it makes the system less flexible; also, if the policy needs to be changed, the whole system has to be replaced.

We do not conclude here how negative permissions should be used; we may, however, investigate how to materialize permission lists when negative authorization is supported in a system (probably with a fixed conflict resolution policy being adopted). We will have to take into account the conflict resolution policy in order to get the effective permission list we want to materialize; we also have to resolve potential conflicts for every update. If these steps are too expensive, partial materialization (e.g., materialize an active user's direct or inherited positive permissions and negative permissions separately) may be adopted; we resolve potential (simplified) conflicts at run time.

References

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 671–682, Chicago, Illinois, USA, 2006. ACM.

[2] Mohammad A. Al-Kahtani and Ravi S. Sandhu. Rule-based RBAC with negative authorization. In *Proceedings of the 20th Annual Computer Security Applications Conference*, pages 405–415, Tucson, AZ, USA, 2004. IEEE Computer Society.

[3] Vo Ngoc Anh and Alistair Moffat. Index compression using fixed binary codewords. In *Proceedings of the Fifteenth Australasian Database Conference*, pages 61–67, Dunedin, New Zealand, 2004. Australian Computer Society.

[4] Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166, 2005.

[5] Gennady Antoshenkov and Mohamed Ziauddin. Query processing and optimization in Oracle Rdb. *VLDB J.*, 5(4):229–237, 1996.

[6] Elisa Bertino, Fabio Origgi, and Pierangela Samarati. An extended authorization model for object databases. *Journal of Computer Security*, 3(2/3):169–206, 1995.

[7] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 61–71, Washington, D.C., USA, 1986. ACM Press.

[8] Daniel K. Blandford and Guy E. Blelloch. Index compression through document reordering. In *Proceedings of 2002 Data Compression Conference*, pages 342–351, Snowbird, UT, USA, 2002. IEEE Computer Society.

[9] Songting Chen, Xin Zhang, and Elke A. Rundensteiner. A compensation-based approach for view maintenance in distributed environments. *IEEE Trans. Knowl. Data Eng.*, 18(8):1068–1081, 2006.

[10] Amir H. Chinaei, Hamid R. Chinaei, and Frank Wm. Tompa. A unified conflict resolution algorithm. In *Proceeding of the 4th VLDB Secure Data Management Workshop*, pages 1–17, Vienna, Austria, 2007. Springer.

[11] W.B. Croft, D. Metzler, and T. Strohman. *Search Engines: Information Retrieval in Practice*. Addison Wesley Publishing Company Incorporated, 2010.

[12] G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 2012.

[13] Peter Gottschling and Dag Lindbo. Generic compressed sparse matrix insertion: Algorithms and implementations in MTL4 and FEniCS. In *Proceedings of the 8th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, pages 2:1–2:8, Genova, Italy, 2009. ACM.

[14] Apple Computer Inc. Understanding packbits. http://devword.apple.com/technotes/tn/tn1023.html, 1996.

[15] Pravin P. Karde and Vilas M. Thakare. Selection and maintenance of materialized view and its application for fast query processing: A survey. *International Journal of Computer Science and Engineering Survey*, 1(2):16–29, 2010.

[16] Paul A. Kargerr. Improving security and performance for capability systems. Technical Report 149, University of Cambridge Computer Laboratory, Cambridge, UK, 1988.

[17] Atif Khan and Ian McKillop. Privacy-centric access control for distributed heterogeneous medical information systems. In *Proceedings of 2013 IEEE International Conference on Healthcare Informatics*, pages 297–306, Philadelphia, PA, USA, 2013.

[18] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, October 1965.

[19] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Publishing Company, Inc., Reading, MA, USA, 1973.

[20] Butler W. Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems*, pages 437–443, Prinston, NJ, USA, 1971. Prinston University.

[21] H.M. Levy. *Capability-based Computer Systems*. Digital Press, 1984.

[22] Imene Mami and Zohra Bellahsene. A survey of view selection methods. *SIGMOD Record*, 41(1):20–29, 2012.

[23] Microsoft. Authorization, users, groups, and the object model in SharePoint 2013. http://msdn.microsoft.com/en-us/library/sharepoint/ms414400.aspx, 2013.

[24] Mark Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability myths demolished. Technical report, Combex Inc, 2003.

[25] Mark Nelson and Jean-Loup Gailly. *The Data Compression Book (2nd ed.)*. MIS:Press, New York, NY, USA, 1996.

[26] Diana K. Smetters and Nathan Good. How users use access control. In *Proceedings of the 5th Symposium on Usable Privacy and Security*, pages 15:1–15:12, Mountain View, California, USA, 2009. ACM.

[27] Frank Wm. Tompa and José A. Blakeley. Maintaining materialized views without accessing base data. *Inf. Syst.*, 13(4):393–406, 1988.

[28] Garfield Zhiping Wu and Frank Wm. Tompa. Effective and efficient bitmaps for access control (summary). In *Proceedings of the 2014 Data Compression Conference*, page 433, Snowbird, UT, USA, 2014. IEEE.

[29] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. A performance comparison of bitmap indexes. In *Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management*, pages 559–561, Atlanta, Georgia, USA, 2001. ACM.

[30] Kesheng Wu, Ekow J. Otoo, Arie Shoshani, and Henrik Nordbergi. Notes on design and implementation of compressed bit vectors. Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory, Berkeley, CA, 2001.

[31] Lingpeng Yang, Dong-Hong Ji, and Mun-Kew Leong. Document reranking by term distribution and maximal marginal relevance for chinese information retrieval. *Inf. Process. Manage.*, 43(2):315–326, 2007.

[32] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th International Conference on World Wide Web*, pages 387–396, Beijing, China, 2008. ACM.

[33] Jingren Zhou, Per-Åke Larson, Jonathan Goldstein, and Luping Ding. Dynamic materialized views. In *Proceedings of the 23rd International Conference on Data Engineering*, pages 526–535, Istanbul, Turkey, 2007. IEEE.

[34] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd International Conference on Data Engineering*, pages 59–70, Atlanta, GA, USA, 2006. IEEE Computer Society.