12

A Two-tier Coarse Indexing Scheme for MLS Database Systems

Sushil Jajodia[1], *Ravi Mukkamala*[2] *and Indrajit Ray*[3]
[1]*George Mason University*
Center for Secure Information Systems and Department of Information and Software Systems Engineering, Fairfax, VA 22030-4444, USA. jajodia@gmu.edu
[2] *Old Dominion University*
Department of Computer Science, Norfolk, VA 23529-0162. mukka@cs.odu.edu
[3]*University of Michigan-Dearborn*
Computer and Information Science Department, Dearborn, MI 48128-1491. indrajit@umdsun2.umd.umich.edu

Abstract

In this paper, we propose a two-tier indexing scheme for multilevel secure database systems, primarily with the intent of improving query response time and reducing the storage required for indexing. At the bottom tier, our scheme requires separate single-level indices, one for each partition of the multilevel relation; at the top tier, our scheme requires a *coarse* multilevel index consisting of only those key values from the single-level indices that are necessary to direct a query to the appropriate single-level index. Our scheme seems suitable for both single-level and range queries. We prove this claim for B^+ trees by providing a detailed performance analysis for this index structure. We also give the algorithms for inserting and deleting key values, as well as for searching for a key value, in the proposed index structure.

Keywords

Database management, Indexing, B^+ Trees, Multilevel Security, Multilevel secure database systems, Kernelized architecture.

1 INTRODUCTION

Database systems often index a relation to make access to the relation's tuples faster than is possible by a sequential scan of the entire relation. An index is created on some *indexing field* (typically the primary key) of the relation; the index file stores each value of the indexing field together with a pointer to the block in the physical

Database Security XI T.Y. Lin and Shelly Qian (Eds)

storage that contains the record with that field value. The index file being much smaller than the relation, can be searched much faster than the latter.

This efficiency is not always achieved in trusted database management systems (DBMSs) (e.g., Informix OnLine/Secure (Informix Software 1993), Trusted Oracle (Oracle Corporation 1996), and Sybase Secure SQL Server (Sybase, Inc. 1993)). This is because they provide only two indexing options: either multiple *single-level* indices, a separate index for data at each security level or a *trusted* multilevel *global* index over all data in the multilevel relation. While the single-level index structure works well for those queries where users specify the security level of the data (we refer to these as *single-level* queries), it performs poorly if the security levels of the data are left unspecified by the users. A common kind of query is the *range query* in which the user gives a desired range of values for the indexing field and wishes to retrieve all those tuples whose values in the indexing field fall within the desired range; obviously, single-level indices perform dismally in the case of range queries (Luef & Pernul 1992). The problem with the multilevel index is that while it is more efficient for answering range queries than the single-level index structure, it is less efficient for single-level queries.

In this paper, we propose a two-tier indexing scheme which retains the advantages of both kinds of index structures. We maintain multiple single level indices, one for each security level and construct a *trusted coarse* multilevel index over these single level indices. The coarse index consists of only those key values from the single-level indices that are necessary to direct a query to the appropriate single-level index. Our scheme seems suitable for both single-level queries as well as range queries. To prove this claim, we choose B^+ tree as the indexing scheme, and provide a detailed performance analysis for this index structure.

The rest of the paper is organized as follows. Section 2 describes the related work. Section 3 contains an example to illustrate the different indexing schemes and their relative merits. Section 4 gives an informal description of the different steps involved in creating and managing our index scheme. (The detailed algorithms have been left out for lack of space.) In subsection 4.4, we describe a variant of our indexing scheme which is cheaper to use in terms of storage, but sometimes is more expensive in terms of querying. In section 5, we give details of our performance model, and summarize the results of our analysis in section 6. Section 7 concludes the paper with a discussion of our future work.

2 RELATED WORK

For obvious reasons, indexing schemes have received wide attention from database researchers (e.g., (see Elmasri & Navathe 1994, Knuth 1973, Korth & Silberschatz 1991)). The notion of B-tree, a variant of B^+ trees, was first introduced by Bayer & McCreight (1972). Algorithms for searching and insertion and deletion in B-trees and B^+ trees can be found in Knuth (1973) while the issue of concurrent operations

on B-trees has been dealt with in Bayer & Schkolnick (1977) and Lehman & Yao (1981).

To the best of our knowledge, the only work that deals with indexing for MLS database is by Luef & Pernul (1992). They maintain multiple single-level indices, one index for each security class. To facilitate range queries, they add *cross links* in the index structure; a cross link is a pointer from a block B_i to another block B_j at a lower security level, signifying that the range of data in B_j contains a subset of the range of data in B_i. Cross links help in the evaluation of a range query by allowing immediate access to the block containing the relevant data in the next index once an initial index has been searched for values in the range. Although the concept of cross links is novel and seems to speed up the evaluation of range queries, the cost of maintaining the cross links appears to be very high. The authors assume that the security levels in the system form a total order; it seems that for any arbitrary partial order of security classes the overhead of maintaining cross links may be prohibitive. In fact, the authors acknowledge that it is impossible to predict whether indexing with cross links performs better or worse than the single-level index structures.

3 MOTIVATION FOR OUR APPROACH

Consider the multilevel index file F shown below:

Key Values: 5(TS), 7(TS), 8(TS), 9.5(TS), 10(S), 12(S), 15(C), 13(C), 7.5(C), 7.3(C), 20(S), 21(S), 22(S), 9(TS), 7.6(C), 11(S), 6(TS), 14(C)

A single-level index structure for this file is shown in figure 1. If a user query does not specify the security level of the data to be retrieved, then this query must be processed at least at those partitions whose levels are dominated by the level of the search request. This is useful if the search yields data at multiple levels. However, if the relevant records are found in only a small number of the partitions or only at a single level then the effort in searching indices at the other partitions is wasted. This is particularly wasteful if there are a large number of security levels and hence a large number of partitions to search. Note that single-level indices have the advantage that they are easier to maintain and can be maintained locally at each partition.

The alternate approach is to maintain a global multilevel index created over all key values. Such an index performs well for range queries, but not for single-level queries because the global index is created over all key values.

We try to combine the advantages of both the indexing schemes with our approach and propose a two-tier index structure. The index structure at the bottom tier consists of a collection of B^+ trees, one for each single-level partition of the multilevel relation. On top of these single-level indices, we maintain a multilevel B^+ tree index consisting of selected key values from each of the single-level indices. Unlike the global indexing scheme, the top tier index is really a coarse index; it contains only those values that are necessary to direct a query to the appropriate single-level index.

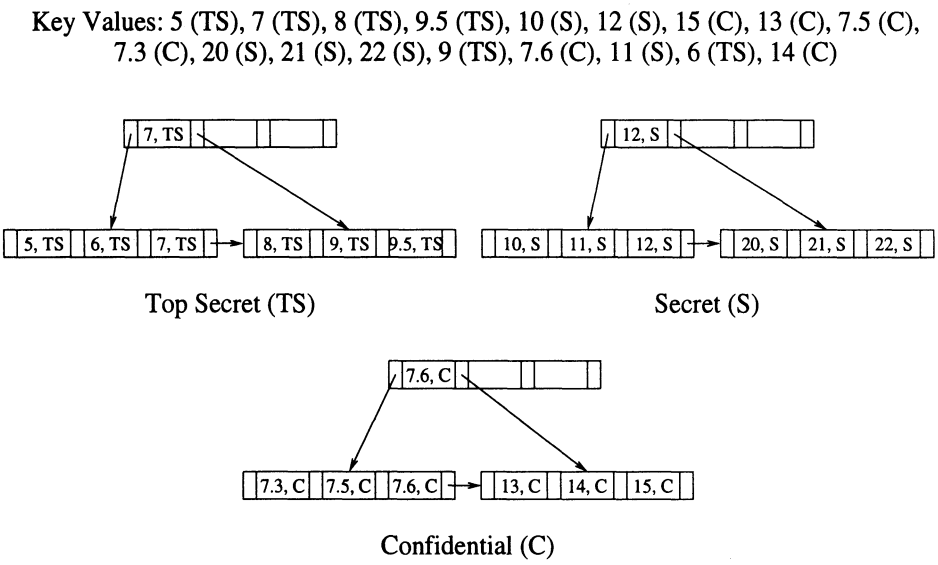


Figure 1 Single-level Index for Each Partition

Since we do not want to repeat the search through a single-level index starting from its root, the pointer from the leaf node of the coarse level index points to the leaf node of the relevant single-level index.

The proposed index structure for the index file F is shown in figure 2. Single-level queries are evaluated by bypassing the coarse index and going directly to the appropriate single-level index. All other queries are evaluated at the coarse level first and then directed to the leaf nodes of only those single-level indices that may contain the desired key. The following section describes the indexing scheme in greater details.

4 THE INDEXING SCHEME

There are several possible variations of our coarse index structure. Each trades off the amount of storage required by the index structure for the cost of a query. We will describe only one of the index structures, which we choose to call *Coarse Index 1*, in detail; this will give an understanding of the key issues involved in the proposed structure. We will briefly describe a variant, called the *Coarse Index 2*, to give some idea about different coarse index structures and then provide a detailed performance analysis of both index structures. A discussion of other variations of our index structures will be the content of a future work.

Below, when we speak of a key, we assume that it also includes the security label associated with the key.

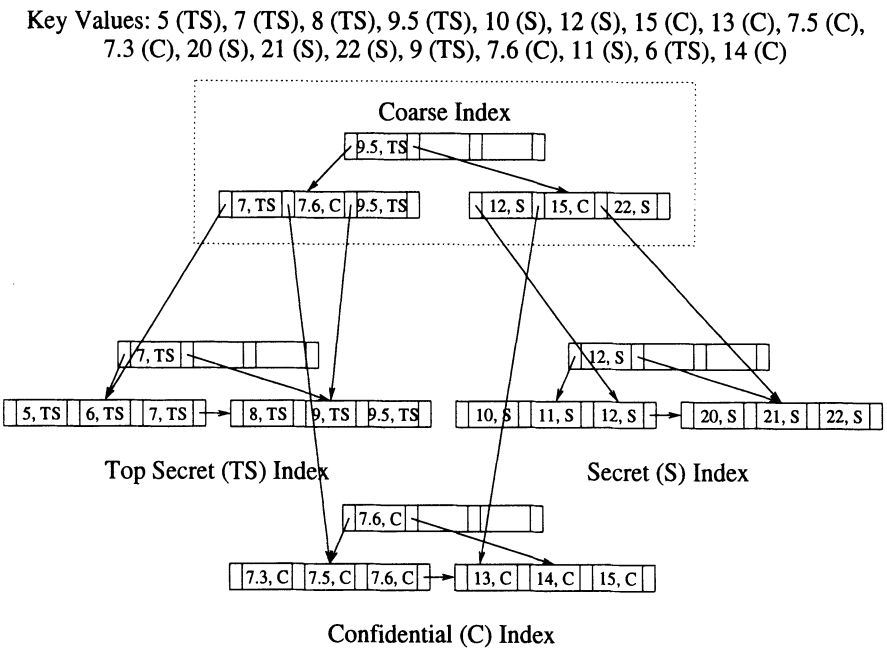


Figure 2 Two-tier Coarse Index for MLS Data

4.1 Inserting a key in the index

A key K_i is inserted in the coarse index as follows:

1. Search the relevant single-level index structure to find out if the key K_i being inserted is already present. If so, return with an insertion violation error.

2. If K_i can be inserted, then insert it at the relevant single-level index.

3. Determine if K_i and/or any other key K_j from this single-level index need to be inserted at the coarse level index:

(a) If, after K_i is inserted at the single- level index, K_i is the largest key in the node into which it was inserted, then K_i needs to be inserted at the coarse level index.

(b) If the node of the single-level index in which K_i is to be inserted has to be split into two nodes to accommodate K_i, then the largest keys K_j and K_k in each of the two nodes need to be inserted at the coarse level index. Note that a situation can arise where one of K_j or K_k was the largest key value in the node before it was split. In that case, this key value is already present in the coarse index and need not be re-inserted. Further, one of K_j and K_k may be the same as K_i; in this case the key will be inserted according to step 3.(a) above.

(c) In all other cases there is no need to insert any value at the coarse index and insertion process is complete.

4. If a key K_l (K_i and/or some other key) from step 3 above is being inserted in

the coarse index, determine if there is a need to get a key from a single level index other than the one in which K_l belongs (This process is necessary because there is intermingling among the keys):

(a) If K_l is to be inserted between keys K_m and K_n in the coarse index, then visit the single-level index pointed to by the pointer P_n (for key K_n), provided this single-level index is different from the one that K_l is in. Note that this pointer points to key values that are less than or equal to K_n but greater than K_m. Thus, there is a possibility that there are keys in the index pointed to by K_n that are less than K_l. In such a case, get the largest of such keys from the single-level index and insert it along with K_l in the coarse level index.

(b) If K_l is to be inserted as the smallest key in the coarse index and if the pointer corresponding to the current smallest value in the coarse index points to a single-level index different from the one in which K_l is, then as in step 4.(a), get the largest key value in this index that is smaller than K_l and insert it in the coarse index along with K_l.

(c) In all other cases, insert K_l only in the coarse index.

5. Once all the relevant keys have been inserted in the coarse index, along with pointers to the corresponding leaf nodes of the single-level indices containing these keys, the insertion process terminates.

4.2 Deleting a key from the index

We now describe how a key is deleted from the index structure. An important feature to note in this deletion process is that if a key K_i is deleted from the coarse index, we may need to insert another key value K_j from the single-level index containg K_i, such that K_j is the largest value smaller than K_i in the single-level index. If K_j occupies the same position in the coarse index that K_i originally did, then we will just replace K_i with K_j. Otherwise, we may have to insert a second key K_k from a different single-level index as was required in the insertion algorithm.

1. If K_i is not present in the single-level index, then return with a deletion violation error.

2. If key K_i can be deleted, then delete it from the single-level index.

3. If K_i is in the coarse index, then delete K_i from the coarse index. Get a replacement for K_i from the single-level index that K_i was in as follows:

(a) If the deletion process results in the merging of two adjacent leaf nodes, then let K_j be the largest key value in the merged node.

(b) If the deletion process does not result in any merging of leaf nodes, then let K_j be the largest key smaller than K_i in the node that contained K_i.

(c) K_j is K_i's replacement in the coarse index provided K_j is not already present in the coarse index.

4. If K_j is not already present in the coarse index, insert K_j following the insertion algorithm described earlier. Note that, as in the insertion algorithm, this step

may require a second key K_k from some single-level index different from the one containing K_j, to be inserted in the coarse index.

5. This completes the key deletion process.

4.3 Searching for a key in the index

The search process is straightforward and works as follows:

1. If the key K_i being searched for is in the coarse index, locate K_i in a leaf node of the coarse index. Follow the pointer P_i corresponding to K_i to a leaf node of some single-level index. Locate K_i in this leaf node and return with success.

2. If K_i is not in the coarse index, locate the smallest key value K_j in the coarse index that is larger than K_i.

(a) Follow the pointer P_j corresponding to K_j to the leaf node of some single-level index containing K_j.

(b) Search for K_i in this leaf node. If found, return search successful; otherwise return search unsuccessful.

4.4 A variation of the coarse index structure

Refer to the key insertion procedure described in section 4.1. Note that whenever a leaf node at a single-level index is split, we insert a key value from each of the resulting two nodes in the coarse index. This results in the coarse index having at least one entry in its leaf nodes for each of the leaf nodes of any single-level index. To find a key at a leaf node of a single-level index after the search has traversed the coarse index structure, it requires a search of only one leaf node of the single-level index.

In this variation, henceforth called Coarse Index 2, we no longer require that for every leaf node of a single-level index there be at least one entry in the coarse level index. Instead, for a chain of leaf nodes N_i, N_j ... N_p in that order, we include some value K_p of N_p at the coarse index and make the pointer of K_p point to leaf node N_i. As a result all key values in N_i, N_j, ... N_p up to the value K_p are pointed at by the pointer of K_p in the coarse index. To find a key value that is less than or equal to K_p in the single-level index, we follow K_p's pointer in the coarse index to leaf node N_i, then sequentially search the chain of leaf nodes till we either come to the desired value or get to K_p.

5 PERFORMANCE ANALYSIS

In order to assess the suitability of the proposed indexing schemes for specific secure database applications, it is important to determine the costs and benefits of the

schemes. We now derive analytical expressions for two chosen performance metrics for the single-level indices, the Global indexing scheme, and the two Coarse Index methods. We model an MLS database system in terms of the following parameters:

o The number of keys (or records) in the system (D)

o The number of distinct security levels (L)

o The distribution of the keys among the different security levels ($R_1 : R_2 : \ldots : R_l$)

o The expected number of keys in a given query range (Q) (Here, we assume that query is a read-only operation which specifies a range of keys to be searched)

o The order of the B^+ trees (P)

o The average fullness factor for the nodes in the B^+ trees (F) (that is, on an average only a fraction F of the entries in any given index tree are filled. Obviously, $1 \geq F \geq 0.5$ by the definition of the B^+ tree)

o The average size of the clusters ($S_1, S_2, \ldots, S_L$), which is determined by the interleaving of keys of different security levels

The model parameters are summarized in Table 1. The following example database illustrates the notation. Since it is also used as a base case in the results section, to describe the effect of different parameters on the performance, we refer to it as the base database. The base database has one million keys ($D = 10^6$), with four distinct security levels ($L = 4$, U $\preceq$ C $\preceq$ S $\preceq$ TS), with the keys being distributed in the ratio of 4:3:2:1 among the four classes (where 4, 3, 2, 1 correspond to levels U, C, S, and TS, respectively). The query under consideration covers 1000 keys ($Q = 1000$). The order of the B^+ tree is 10 ($P = 10$) and each node is assumed to be full ($F = 1.0$) at the time the query arrives. The average cluster sizes of the U, C, S, and TS classes are 100, 75, 50, and 25 keys, respectively. For example, on the average, the number of consecutive keys with security level U is 100. The data for the base case is also included in Table 1.

¿From the basic model, we can derive the following factors which are used in the rest of the analysis. Note that we use upper case letters for the basic model parameters and lower case letters for derived parameters.

o Total number of keys of level i, $k_i = D \cdot \frac{R_i}{\sum_{j=1}^{L} R_j}$

o The average number of clusters of level i, $c_i = k_i / S_i$

o Average number of keys in the query range corresponding to level i, $q_i = Q \cdot \frac{R_i}{\sum_{j=1}^{L} R_j}$

o Average number of clusters in the query range corresponding to level i, $r_i = q_i / S_i$

o The total number of clusters in the query range, $r_\sigma = \sum_{i=1}^{L} r_i$

To compare the performance of different index methods, we chose two metrics: size of the index table and the cost of executing a query. These are defined as follows:

1. *Size of index table:* We measure the size in terms of the number of nodes (often referred to as blocks in literature (Elmasri & Navathe 1994)) of the B^+ tree representing the index structure. Obviously, the smaller the size, the more suited it is for database applications, especially those running on limited memory machines. We denote this metric by α. The metric is computed for the Global index (α_1),

the Coarse Index 1 (α_2), the Coarse Index 2 (α_3), and the four single-level indexes ($\alpha_u, \alpha_c, \alpha_s, \alpha_{ts}$).

2. *Cost of query execution:* During the execution of a range query, an index is searched to access the actual data. Since the number of data blocks to be accessed for executing the query execution is independent of the index method, we have defined the cost of query execution only in terms of the number of nodes (or blocks) of the index structure (B^+ tree) that would be searched (or accessed). This measure is well-suited for comparing the different indexing methods being proposed in this paper. We denote this cost by β. The metric is computed for queries at the four different levels. It is assumed that a query at a given level needs to access data corresponding to its own level as well as those dominated by it. For example, an S level query accesses data related to U, C, and S levels. Accordingly, the cost of a query at S level includes the cost of accessing indexes at U, C, and S levels. We compute this metric for the Global index (β_1), the Coarse Index 1 (β_2), the Coarse Index 2 (β_3), and the four single-level indexes ($\beta_u, \beta_c, \beta_s, \beta_{ts}$).

5.1 Evaluation of the index size metric

In order to estimate the size of a B^+ tree, we first need to determine its height. Given the order (P), the fullness factor (F), and the number of keys K (or data pointers at the leaf level), the height of a tree (h) can be estimated as $h = [\log(K/(F \cdot (P - 1)))/\log(F \cdot P)] + 1$ (Elmasri & Navathe 1994). Here, we assume that while the intermediate nodes of a B^+ tree can hold up to P pointers to lower level nodes, the leaf node only holds up to $(P - 1)$ data pointers as one of the pointers is used for pointing to its right sibling node. Using this equation, the expressions for the single-level indices and the global index can be easily derived by proper substitutions for K. These are summarized in Table 2. The expressions for the coarse indices, however, are more complex. We now derive these expressions.

In the case of Coarse Index 1, the leaf node has one or more pointers to a cluster depending on whether or not a cluster covers one or more nodes at the leaf level of a single index. In other words, if a cluster of keys occupy n (partial or full) nodes at the corresponding index tree's leaf level, then Coarse Index 1 would have n pointers pointing to the n nodes. If n_i denotes the average number of nodes (or blocks) that a cluster of level i occupies in the single-level index at level i, then it can easily be derived as follows where $\oplus$ represents the modulus operator: $n_i = 1 + \lfloor (S_i - 1)/(F \cdot (P - 1)) \rfloor + \frac{\lfloor S_i - 1 \rfloor \oplus \lfloor F \cdot (P-1) \rfloor}{(F \cdot (P-1))}$.

Since there are c_i clusters for level i, Coarse Index 1 would point to $\sum_{i=1}^{L} (c_i \cdot n_i)$ nodes at the single-level indices. Thus, the height of Coarse Index 1 is given by substituting this term for K in Equation (1). Since Coarse Index 2 only has one pointer to each of the clusters, it would point to $\sum_{i=1}^{L} c_i$ nodes at the single indices.

Hence, its height is given by substituting this value for K in Equation (1). These are summarized in Table 2.

Once the height h is known, the size of an index table may be computed using the properties of the B^+ tree as $\alpha = [(F \cdot P)^h - 1]/[F \cdot P - 1]$. By substituting the appropriate term for h in Equation (3), we can compute the size metric for the single-level indices ($\alpha_u, \alpha_c, \alpha_s, \alpha_{ts}$), the Global index ($\alpha_1$), the Coarse Index 1 ($\alpha_2$), and the Coarse Index 2 (α_2). These are summarized in Table 2.

5.2 Evaluation of the cost of query execution

Let us first consider the case of single-level indices. Clearly, given a U level query, we need to access only the index for U level. Once we reach the leaf node of the index that represents the beginning of the query range, data pointers to other keys may be obtained by traversing the leaf nodes horizontally (using the right sibling links) until the range is covered. Accordingly, the cost metric β_u is given by $\beta_u = h_u + \lfloor q_u/[F \cdot (P-1)] \rfloor$. For a query of level C, we need to access both keys of type U and C by repeating search on single-level indices of C and U. Accordingly, $\beta_c = h_u + \lfloor q_u/[F \cdot (P-1)] \rfloor + h_c + \lfloor q_c/[F \cdot (P-1)] \rfloor$. We can derive similar expressions for β_s and β_{ts}. These are summarized in Table 3.

In the case of Global index, since keys of all levels are present at the leaf, all keys in the query range (i.e., Q keys) need to be searched, for any level of query. Hence, $\beta_1 = h_1 + \lfloor Q/[F \cdot (P-1)] \rfloor$.

The same β_1 will be applicable for all four types of queries (U, C, S, TS). The computations of the cost metrics are more complex for Coarse Index 1 and Coarse Index 2. First let us consider the case of Coarse Index 1. For a level j query, we need to find the pointer to the first cluster in the given query range for level j and all levels dominated by j. Hence, we need to search the leaf nodes of the coarse index, starting from the beginning of the query range, going horizontally, until we find the first cluster for each type of the required level. If First(i) represents the average number of nodes to be scanned to find the first cluster of level i, starting from the beginning of query range, then the number of nodes scanned in Coarse Index 1, before all the required first cluster pointers are obtained, is given by $\max_{\forall i, i \leq j} \text{First(i)}$. In addition, since the corresponding single-level indices have also to be scanned horizontally, the cost of execution is given by $\beta_{2,j} = h_2 - 1 + \max_{1 \leq i \leq j} \text{First}(i) + \sum_{1 \leq i \leq j} \left\lceil \frac{q_i}{F \cdot (P-1)} \right\rceil$.

First(i) may be computed by using probabilistic arguments. (We omit the details here.) It is given as follows where t_i is the average size of clusters in the query range excluding cluster i. ($t_i = \sum_{l=1, l \neq i}^{L} (r_l / r_\sigma) \cdot S_l$): $\text{First}(i) = \frac{r_i}{r_\sigma} + \sum_{k=1}^{r_\sigma - 2r_i} \left(\prod_{l=0}^{k-1} \left(1 - \frac{r_i}{r_\sigma - l} \right) \right) \cdot \frac{r_i}{r_\sigma - k} \cdot \left\lceil \frac{k \cdot t_i}{F \cdot (P-1)} \right\rceil$.

The computations for Coarse Index 2 are quite similar except that this index carries only one pointer per cluster. Thus, the expression for $\beta_{3,j}$ is given as fol-

Table 1 Model parameters and Base case

Mnemonic	Description	Base case
D	Total number of keys	10^6
L	Security levels	4 (U,C,S,TS)
R_i	Portion of level i keys	$R_1 = 4, R_2 = 3, R_3 = 2, R_4 = 1$
Q	Average number of keys within a given query range	1000
P	Order of the B^+ tree	10
F	Node fullness factor	1.0
S_i	Average size of level i cluster	$S_1 = 100, S_2 = 75, S_3 = 50, S_4 = 25$

Table 2 Summary of Height and Size computations

Mnemonic	Expression
h_u	$\frac{\log(K_u/(F\cdot(P-1)))}{\log(F\cdot P)} + 1$
h_c	$\frac{\log(K_c/(F\cdot(P-1)))}{\log(F\cdot P)} + 1$
h_s	$\frac{\log(K_s/(F\cdot(P-1)))}{\log(F\cdot P)} + 1$
h_{ts}	$\frac{\log(K_{ts}/(F\cdot(P-1)))}{\log(F\cdot P)} + 1$
h_1	$\frac{\log(D/(F\cdot(P-1)))}{\log(F\cdot P)} + 1$
h_2	$\frac{\log\left(\sum_{i=1}^{L} c_i\cdot n_i/(F\cdot(P-1))\right)}{\log(F\cdot P)} + 1$
h_3	$\frac{\log\left(\sum_{i=1}^{L} c_i/(F\cdot(P-1))\right)}{\log(F\cdot P)} + 1$

Mnemonic	Expression
α_u	$\frac{(F\cdot P)^{h_u}-1}{F\cdot P-1}$
α_c	$\frac{(F\cdot P)^{h_c}-1}{F\cdot P-1}$
α_s	$\frac{(F\cdot P)^{h_s}-1}{F\cdot P-1}$
α_{ts}	$\frac{(F\cdot P)^{h_{ts}}-1}{F\cdot P-1}$
α_1	$\frac{(F\cdot P)^{h_1}-1}{F\cdot P-1}$
α_2	$\frac{(F\cdot P)^{h_2}-1}{F\cdot P-1}$
α_3	$\frac{(F\cdot P)^{h_3}-1}{F\cdot P-1}$

lows. $\beta_{3,j} = h_3 - 1 + \max_{1\le i\le j}\text{First}'(i) + \sum_{1\le i\le j}\left\lceil \frac{q_i}{F\cdot(P-1)}\right\rceil$. where $First'(i) = \frac{r_i}{r_\sigma} + \sum_{k=1}^{r_\sigma - 2r_i}\left(\prod_{l=0}^{k-1}\left(1-\frac{r_i}{r_\sigma - l}\right)\right)\cdot\frac{r_i}{r_\sigma - k}\cdot\left\lceil\frac{k}{F\cdot(P-1)}\right\rceil$.

6 RESULTS

To determine the effect of different system parameters on the two performance metrics (α and β) with different indexing methods, we have evaluated the metrics under different configurations. Our evaluation methodology is as follows.

1. The base case for the evaluation is as described in Table 1.
2. Under each of the configurations, we evaluate the index size (α) for the four

Table 3 Summary of Cost of Query Executions

Mnemonic	Expression
β_u	$h_u + \left\lfloor \frac{q_u}{F \cdot (P-1)} \right\rfloor$
β_c	$h_u + h_c + \left\lfloor \frac{q_u}{F \cdot (P-1)} \right\rfloor + \left\lfloor \frac{q_c}{F \cdot (P-1)} \right\rfloor$
β_s	$h_u + h_c + h_s + \left\lfloor \frac{q_u}{F \cdot (P-1)} \right\rfloor + \left\lfloor \frac{q_c}{F \cdot (P-1)} \right\rfloor + \left\lfloor \frac{q_u}{F \cdot (P-1)} \right\rfloor$
β_{ts}	$h_u + h_c + h_s + h_{ts} + \left\lfloor \frac{q_u}{F \cdot (P-1)} \right\rfloor + \left\lfloor \frac{q_c}{F \cdot (P-1)} \right\rfloor + \left\lfloor \frac{q_u}{F \cdot (P-1)} \right\rfloor + \left\lfloor \frac{q_{ts}}{F \cdot (P-1)} \right\rfloor$
β_1	$h_1 + \left\lfloor \frac{Q}{F \cdot (P-1)} \right\rfloor$
$\beta_{2,j}$	$h_2 - 1 + \max_{1 \le i \le j} \text{First}(i) + \sum_{1 \le i \le j} \left\lceil \frac{q_i}{F \cdot (P-1)} \right\rceil$
$\beta_{3,j}$	$h_3 - 1 + \max_{1 \le i \le j} \text{First}'(i) + \sum_{1 \le i \le j} \left\lceil \frac{q_i}{F \cdot (P-1)} \right\rceil$

single-level indices (U,C,S,TS), the Global index (GI), the Coarse Index 1 (CI1) and Coarse Index 2 (CI2). The cost of query metric β) is evaluated for all four types of queries (U,C,S,TS). For each type, we evaluate the metric when only the single-level index for that type is available, as well as for the Global index, and the two Coarse indices.

3. To evaluate the effect of cluster size on the metrics, we vary the cluster size (S) in the base model. In the current runs, we keep the ratio of the cluster sizes constant at 4:3:2:1 as in the base model but varied the constant factor from 1 through 10^6. So the cluster sizes were varied from the set $< 4,3,2,1 >$ to $< 4 \cdot 10^6, 3 \cdot 10^6, 2 \cdot 10^6, 10^6 >$. Due to space limitations, only a subset of the results are presented in figures 3 and 7.

4. To evaluate the effect of key ratio size on the metrics, we vary the key ratio (R_i's) in the base model. We evaluated the metrics under a different set of ratios including 1:1:1:1, 10:1:1:1, 100:1:1:1, 1000:1:1:1, ..., 1:1:1:1000. Due to space limitations, the results are not plotted, but the summary of the observations is presented below.

5. To evaluate the effect of the tree order (P), we vary the order from 2 through 500. The results are summarized in figures 4 and 8.

6. To evaluate the effect of number of keys (D), we experimented with D values from 100 to 10^8. Portions of the results are summarized in figures 5 and 9.

7. To evaluate the effect of the fullness factor (F), we changed values of F from 0.7 through 1.0. The results are summarized in figures 6 and 10.

8. To determine the query range effect, Q was varied from 1 through 10^6. The observations are summarized below.

Following is a summary of our observations regarding the behavior of different indexing methods under different model parameters.

Index size (α): As mentioned above, index size is a very important metric since it

determines the overall performance of the system. If the index size is small enough that it can fit in the fast memory of the system, then quick turnaround times may be achieved for query executions. Following is a summary of our analysis illustrating the effect of different system parameters on the size of various index methods.

(i) *Effect of Cluster size (S_i):* Since the single-level indices and the Global index have pointers to individual data and not to clusters, this parameter has no effect on their sizes (i.e., αs). On the other hand, it has significant impact on the sizes of Coarse Index 1 and Coarse Index 2 (see figure 3). As the size of the clusters increase, the number of clusters decrease. This means that the number of elements pointed to by Coarse Index 2 also decrease. For example, when the cluster sizes of the four classes were 40, 30, 20, 10, respectively, Coarse Index 2 requires 4938 nodes (or blocks). But when the size of each cluster is ten times (i.e., 400, 300, 200, 100), the size decreases to 494, a tenth of the previous. Thus the Coarse Index 2's size is inversely proportional to the cluster size.

Since Coarse Index 1 points to all nodes that a cluster of data keys occupy at a single-level index, the relationship between its size (α_2) and the cluster size is not so straightforward. In general, increase in cluster size would decrease the index size but not as dramatically as in the case of Coarse Index 2. For example, in our experiments, when the cluster sizes were changed from 40, 30, 20, 10 to 400, 300, 200, 100 respectively, for the four levels, the index size reduced from 20987 nodes to 14444 nodes.

(ii) *Effect of Key ratio (R_i):* Since the sizes of single-level indices directly depend on the number of keys they have to point to, the key ratio has an effect on individual index sizes. However, the sum of their sizes remains essentially unchanged. Thus while the sizes of single-level indices were 30864 each when the ratio was 1:1:1:1, it changed to 49382, 37036, 24961, 12345 respectively, when the ratios were changed to 4:3:2:1. Observe that the difference in the sum of sizes in the two cases is not significant. Obviously, the size of the Global index is unaffected by the ratios since it has keys of all levels. Similarly, this factor does not have significant impact on the sizes of Coarse Index 1 and Coarse Index 2.

(iii) *Effect of Tree order (P):* Since the order determines the maximum number of elements in a node of an index tree, the size of the tree decreases with the increasing order (see figure 4). While the decrease is considerable for smaller values of tree order, the effect is not so dramatic at higher values of tree order. For example, while the size of Coarse Index 2 reduced from 32000 to 7100 when order is increased from 2 to 4, it only reduced from 330 to 160 when the order is doubled from 50 to 100. The performance shows similar trends for all index types.

(iv) *Effect of Number of Keys (D):* The sizes of all indices grow linearly with the number of keys (see figure 5). For example, when the number of keys is increased from 10^4 to 10^6, the size of Coarse Index increased from 20 to 2000. Similarly, the size of Global index grew from 1235 to 123500 for the same changes in the number of keys.

(v) *Effect of Fullness factor (F):* An increase in fullness factor implies that the

same tree size can accommodate more number of keys. However, since B^+tree requires that each node be at least half-full, F's effect on the size is not as significant as other factors. However, it can be noticed that the size decreases, somewhat linearly, with the fullness factor (see figure 6). For example, as the fullness factor is increased from 0.7 to 0.8 and then to 0.9, the size of Coarse Index 2 changed from 2963 to 2540 and then to 2222.

(vi) *Effect of Query Range (Q):* Obviously, there is no effect of query range on the index size.

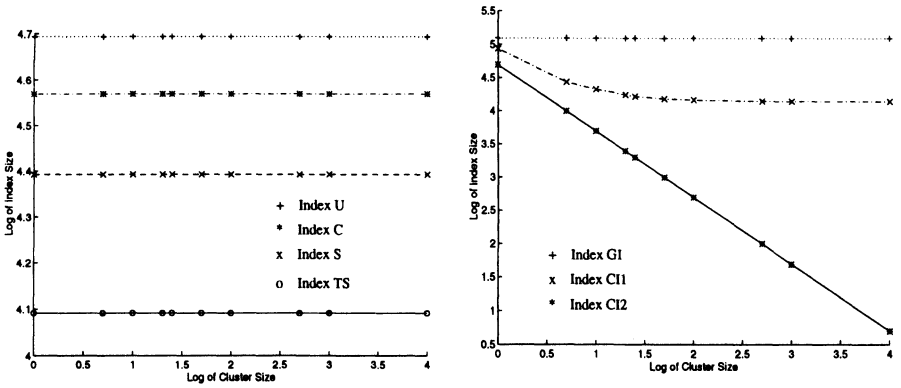
Cost of Query (β)**:** This metric, which indicates the number of node (or block) accesses required to execute a query, influences the execution time of the query. Obviously, for a lower turnaround time, we prefer as few nodes to be accessed as possible. Following is a summary of our analysis illustrating the effect of model parameters on this metric. Due to space limitations we have included the results for query types U and TS in figures 7-10.

(i) *Effect of Cluster Size:* Since the structure of the single-level indices and the Global index are unaffected by clustering, the query cost is also unchanged under these cases (figure 7). In the case of Coarse Index 1, given any query range, the index guarantees a pointer to the single-level index that begins the range. Hence, except in some special cases where the cluster size is not a multiple of the node order (or $F \cdot P$), this metric is not sensitive to the cluster size.

The situation is quite different with Coarse Index 2. Here, since the index guarantees a pointer only to the beginning of a cluster in the single-level index, it is sensitive to the cluster range. For example, if a cluster with a size of 1000 starts from key 100 and ends with key 1099 (here, for simplicity we assumed that all keys from 100 to 1099 exist), then a query requiring the range of 100-5000 is pointed to the same block in the single-level index as the one with a query range of 1098-6000. Hence, the second query has to search several additional nodes at the leaf of a single-level index before it encounters the block with 1098 from where the search starts. Larger the cluster size, larger is such overhead. In the example cases that we studied, for cluster sizes of up to 100, the performance of both index methods was comparable. Beyond cluster size of 300, Coarse Index 2 becomes expensive.

(ii) *Effect of Key Ratio (R_i):* While this has some effect on the cost of single-level index accesses and coarse index accesses, it has no effect on the Global index. The query cost using Coarse Index 1 is affected to a small extent. For example, when the key ratio is changed from 10:1:1:1 to 1000:1:1:1, the query cost for U with Coarse Index 1 increased from 90 to 115. Similarly, for TS query, it increased from 155 to 170. For the same changes, the single-level index access cost increased from 90 to 115. In fact, the increase in the Coarse Index 1 is to be mainly attributed to the increase in single-level index access only. For the same changes, the cost with Coarse Index 2 increased from 95 to 120.

(iii) *Effect of Tree Order (P):* The tree order has similar effects on the query cost as it has on the index size—while the rate of reduction in the cost is exponential at the smaller values of P, it is not so large for larger values of P (see figure 8). For



(a) Graph for U, C, S and TS

(b) Graph for GI, CI1 and CI2

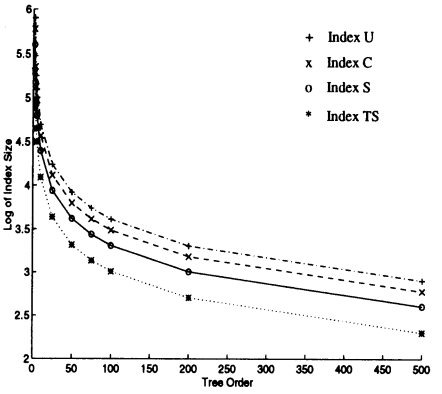
Figure 3 Effect of Cluster Size on Index Size

example, for a U query, while the cost for Coarse Index 2 reduced from 470 to 160 when order is increased from 2 to 4, it only reduced from 13 to 8 when the order is doubled from 50 to 100. The performance shows similar trends for all index types and all query types.

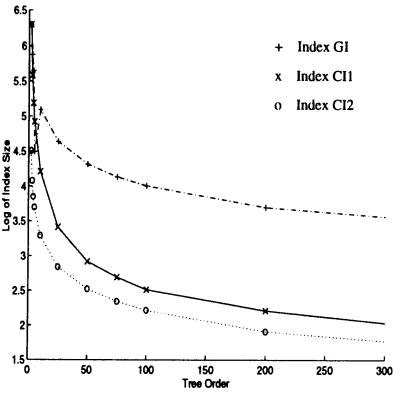
(iv) *Effect of Number of Keys (D):* Since all indices grow logarithmically with the number of keys, the cost also grows only logarithmically with the number of keys (see figure 9). For example, when the number of keys is increased from 10^4 to 10^6, the cost of U query with Coarse Index 2 increased from 53 to 55. Similar changes with Coarse Index 1 resulted in an increase of cost from 57 to 59.

(v) *Effect of Fullness factor (F):* As in the case of the index size, the fullness factor results in increased block accesses while descending the index trees during searching as well as while performing a horizontal search at the leaf nodes (see figure 10). Thus, the query cost increases as the fullness factor decreases. For example, when the Fullness factor is increased from 0.7 to 0.9, the cost of U query with Coarse Index 2 decreased from 77 to 60. Similar changes with Coarse Index 1 resulted in a decrease of cost from 83 to 65.

(vi) *Effect of Query Range (Q):* Since a horizontal scanning of nodes involving all the keys in a given range is required at the leaf level of the indices, this factor has an effect on the query cost. Since each node of the tree holds $F \cdot P$ pointers or 10 in our case, the increase is logarithmic (with base 10). For example, when the query range is increased from 100 to 10000, the cost of U query with Coarse Index 2 increased from 15 to 455. The same change with Coarse Index 1 resulted in an increase in cost from 10 to 460. With Global index, the cost increased from 17 to 1117.

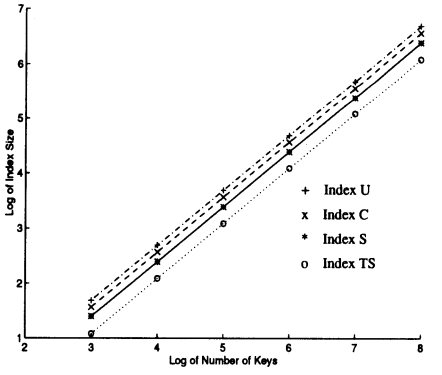


(a) Graph for U, C, S and TS

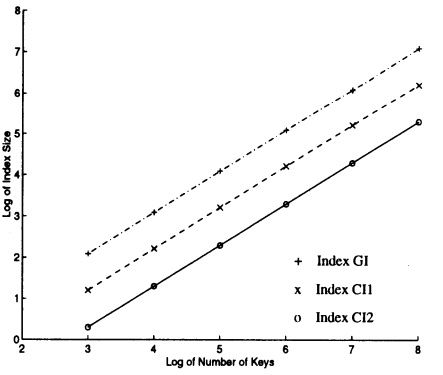


(b) Graph for GI, CI1 and CI2

Figure 4 Effect of Tree Order on Index Size

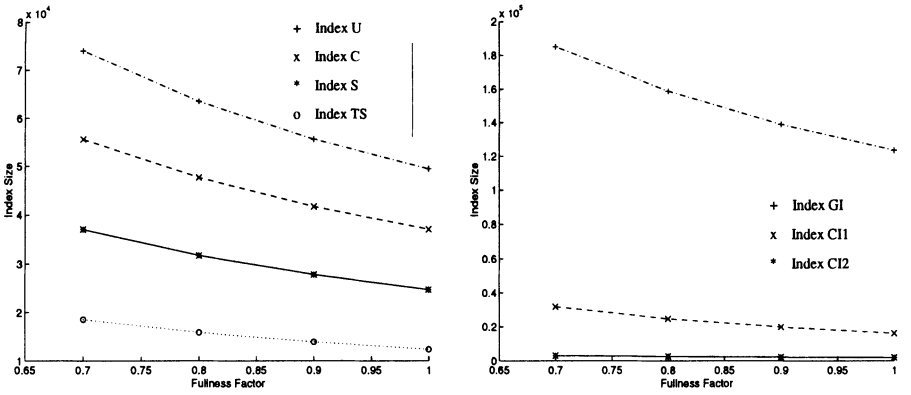


(a) Graph for U, C, S and TS



(b) Graph for GI, CI1 and CI2

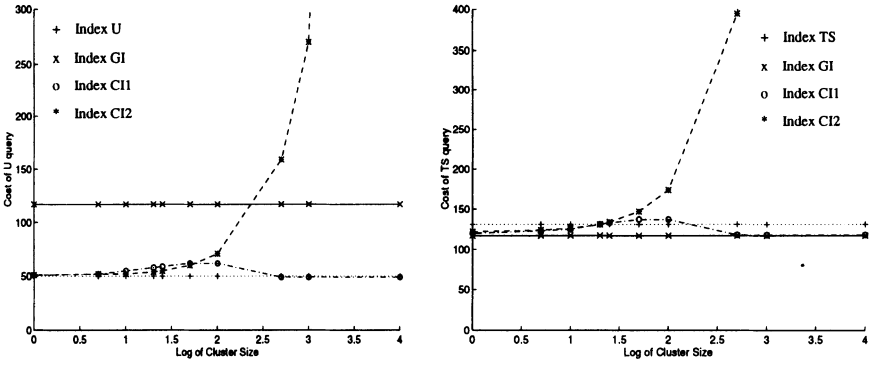
Figure 5 Effect of Number of Keys on Index Size



(a) Graph for U, C, S and TS

(b) Graph for GI, CI1 and CI2

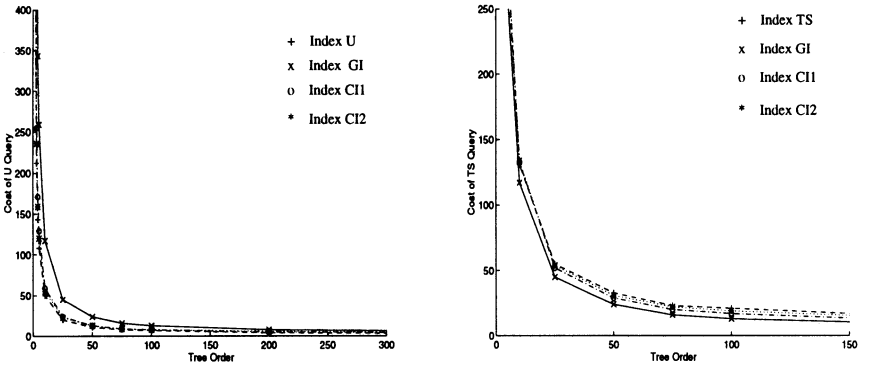
Figure 6 Effect of Fullness Factor on Index Size



(a) Graph for U, GI, CI1 and CI2

(b) Graph for TS, GI, CI1 and CI2

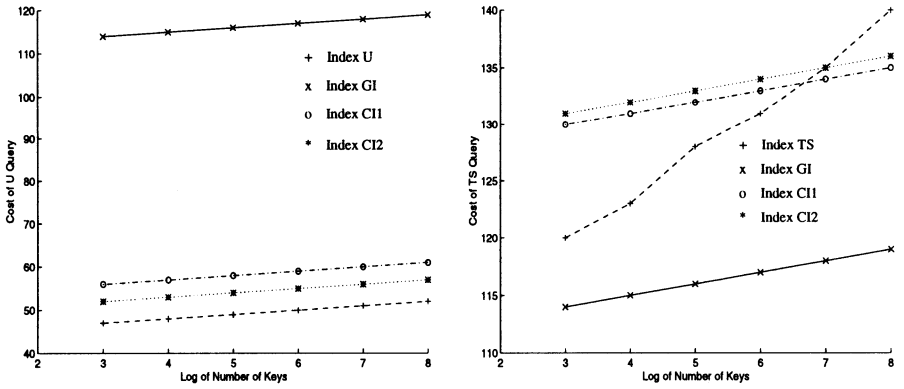
Figure 7 Effect of Cluster Size on Cost of Query



(a) Graph for U, GI, CI1 and CI2

(b) Graph for TS,GI, CI1 and CI2

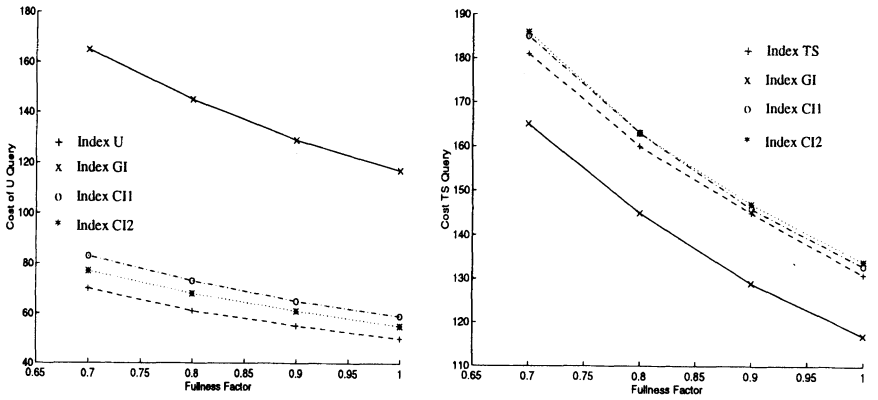
Figure 8 Effect of Tree Order on Cost of Query



(a) Graph for U, GI, CI1 and CI2

(b) Graph for TS, GI, CI1 and CI2

Figure 9 Effect of Number of Keys on Cost of Query



(a) Graph for U, GI, CI1 and CI2

(b) Graph for TS, GI, CI1 and CI2

Figure 10 Effect of Fullness Factor on Cost of Query

7 CONCLUSION

In this paper, we have introduced two types of coarse indexing schemes — Coarse Index 1 and Coarse Index 2 — in the context of MLS databases, primarily with the intent of improving query response time and reducing the size of indices. With the encouraging results from this study, we propose to look at other coarse indexing schemes. Especially, we propose to look at schemes in which a coarse index points to a security level that a cluster belongs to, and at schemes in which the coarse index points to some intermediate node of a single index where a cluster begins. Finally, we have assumed that the coarse index is trusted. It would be interesting to investigate a kernelized implementation of the coarse index.

REFERENCES

Bayer, R. & McCreight, E. (1972), 'Organization and Maintenance of Large Ordered Indexes', *Acta Informatica* **1**, 173–189.

Bayer, R. & Schkolnick, M. (1977), 'Concurrency of Operations on B-Trees', *Acta Informatica* **9**, 1–21.

Elmasri, R. & Navathe, S. B. (1994), *Fundamentals of Database Systems*, second edn, Addison-Wesley, Reading, MA.

Informix Software (1993), *Informix-OnLine/Secure Security Features User's Guide*, Menlo Park, CA.

Knuth, D. (1973), *The Art of Computer Programming, Volume 3: Sorting and Ser-*

aching, Addison-Wesley.

Korth, H. F. & Silberschatz, A. (1991), *Database System Concepts, Second Edition*, McGraw–Hill, New York.

Lehman, P. & Yao, S. (1981), 'Efficient Locking for Concurrent Operations on B-Trees', *ACM Transaction on Database Systems* **6**(4).

Luef, G. & Pernul, G. (1992), Supporting Range Queries in Multilevel-Secure Databases, *in* C. E. Landwehr & S. Jajodia, eds, 'Database Security, V: Status and Prospects', Elsevier Science Publishers B.V. (North-Holland), pp. 117–130.

Oracle Corporation (1996), *Trusted Oracle7 Server Administrator's Guide, Release 7.2*, Redwood City, CA.

Sybase, Inc. (1993), *Sybase Secure SQL Server Security Administration Guide*, Emeryville, CA.

BIOGRAPHY

Sushil Jajodia is Director of Center for Secure Information Systems and Professor of Information and Software Systems Engineering at the George Mason University, Fairfax, Virginia. His research interests include information security, temporal databases, and replicated databases.

Ravi Mukkamala is an Associate Professor in the Department of Computer Science at the Old Dominion University. His research interests include distributed database systems, data security, high-speed data communications, and performance analysis.

Indrajit Ray is an Assistant Professor in the Computer and Information Science Department at the University of Michigan-Dearborn. His research interests included distributed systems, advanced transaction models and architectures, database management systems and computer and information systems security.