ELSEVIER

Data & Knowledge Engineering 33 (2000) 199–217

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

Index set: A practical indexing scheme for object database systems

Jung-Ho Ahn *, Ha-Joo Song, Hyoung-Joo Kim

Department of Computer Engineering, Seoul National University, Shilim-Dong Gwanak-Gu, Seoul 151-742, South Korea

Received 24 June 1999; received in revised form 8 November 1999; accepted 14 February 2000

Abstract

Efficient indexing in a class hierarchy is essential for the achievement of high performance in query evaluation for object database management systems. In this paper, we present a practical indexing scheme, *index set*, which provides good index configuration for any real database environment. The proposed scheme considers the distribution of key values, as well as query patterns such as query weight on each class. The index set can easily be applied to any database system, since it uses the well-known B^+-tree structure. We develop a cost model and, through experiments, demonstrate the performance of the proposed scheme over various class hierarchies.

Keywords: Object database systems; Index; Class hierarchy; B^+-tree

1. Introduction

1.1. Problems of indexing in object database systems

The commercial success of a data model depends on how well the underlying system is able to support it. It is also well known that indexing is the key to achieving high performance in query evaluation. For instance, the value of the relational data model would be diminished without the efficiency of B^+-tree index structures [4] used to evaluate declarative queries.

However, while they provide optimal performance for queries in one-dimensional space, single-attribute B^+-trees are not suitable for an object-oriented data model. That is, B^+-trees cannot efficiently support queries on class hierarchies since their answer must be restricted to target classes as well as to search predicates.

In order to support queries on class hierarchies, several index structures, including the H-tree [13] and the HcC-tree [15], have been proposed. However, these new structures are seldom used, since they are complex and require new concurrency mechanisms. A practical alternative was

* Corresponding author. Tel.: +82-2-871-6945; fax: +82-2-888-0269; web: http://oopsla.snu.ac.kr.
E-mail addresses: jhahn@oopsla.snu.ac.kr (J.-H. Ahn), hjsong@oopsla.snu.ac.kr (H.-J. Song), hjk@oopsla.snu.ac.kr (H.-J. Kim).

0169-023X/00/$ - see front matter
PII: S0169-023X(00)00006-9

proposed, namely, an indexing scheme based on class-division [14], in which the basic concept is time-space tradeoff. Although indexing by class-division has the advantage of ease in applicability, it takes little notice of the number of instances, and none at all of the distribution of key values. Thus, this approach offers no guarantee of performance enhancement for all cases.

As a result, we set out to develop a new indexing scheme, *index set*, that overcomes the above problems and gives maximal performance gain with minimal space and update overhead.

1.2. Related work

There have been many studies concerning indexing for object databases [2,3,8,10,12–15]. The simplest approach for class-hierarchy indexing is the class-hierarchy index [12]. This method maintains only one index on an attribute of all classes in the class hierarchy. This approach is based on the fact that one index may in general be more efficient in evaluating a query whose access range spans most of the classes in the class hierarchy, than single-class indexes on each class. The class-hierarchy index has the advantage that it is easy to implement since the structure is based on the B^+-tree. However, it may show critical degradation of performance for a query against a leaf class, since it has to read many unnecessary index pages.

Some studies have attempted to solve this problem by introducing links or chains in the B^+-tree structure. Examples are the H-tree [13], the HcC-tree [15], and the CG-tree [10]. However, as mentioned above, these new structures are difficult to use and they do not guarantee sufficient enhancement of performance to admit a new data structure. Therefore, only the class-hierarchy index based on the B^+-tree structure has been used in real object database management systems [14].

Ramaswamy and Kanellakis [14] proposed a practical indexing scheme by class division, which is a variation of the class-hierarchy index. It divides a class hierarchy into several class divisions by their division algorithm and heuristics, and builds indexes on each class division. Here, a class can be included in several divisions simultaneously. That is, the class-division scheme enhances the performance through the replication of indexes. However, the class-division algorithm is not applicable to class hierarchies with multiple inheritance. In addition, like all other indexing schemes, it is not designed to consider the number of instances or the range of key values.

Many indexing techniques for queries on composite hierarchies have also been studied. Bertino and Kim [3] proposed the nested index, path index, and multi index for queries on composite objects, and compared their performance. Ishikawa et al. [8] suggested an indexing scheme using the signature file technique. Other types of indexing for the composite class hierarchy such as access support relation, the object skeleton and the hierarchical join index can be found in [7], [9] and [17], respectively.

Gudes [5] and Bertino and Foscoli [1,2] proposed an index structure that combined the class-hierarchy index and the nested index.

1.3. Paper organization

The rest of the paper is organized as follows: In Section 2, we introduce our new indexing scheme, *index set* for indexing on class hierarchies. Section 3 describes the cost model used in this study and Section 4 presents the results of the performance evaluation. Finally, conclusions from our study and areas for future research are given in Section 5.

2. The index set

2.1. Indexing in a class hierarchy

A query on a class hierarchy may be formulated as a problem of external searching in two-dimensional spaces: one dimension is an attribute against which a predicate is compared, and the other dimension is the classes to which target objects belong.

Example 2.1. A query "Find all students who are 21 years old" on the class hierarchy of Fig. 1 has the search condition that a person should be a `Student` as well as the predicate that the person should be 21 years old. Here, all `TA`s can also be considered `Student`s by the IS-A relationship imposed on the class hierarchy. Therefore, objects of all classes in the hierarchy rooted at `Student` should be searched for this example query. [1]

To enhance the performance of this kind of query on class hierarchies, it is necessary to investigate a new indexing technique that can consider class hierarchies. However, as we have mentioned above, it is difficult to introduce a new data structure. In addition, a general multi-key indexing structure such as the R-tree [6] does not provide satisfactory performance with a query on a class hierarchy, which is a special case of a two-dimensional search [14].

2.2. Our approach

As far as retrieval performance is concerned, building indexes on the full extent of each class would improve the query performance dramatically [11]. The *full extent* of a class is the union of extents of the class or any of its subclasses and the *extent* of a class is the set of instances belonging to the class. For instance, the performance of the query on the class hierarchy in Example 2.1 can be improved significantly by using the set of indexes on the full extent of each of `People`, `Professor`, `Student` and `TA`. Each index is built using the B^+-tree structure, which is the best general purpose index structure for one-dimensional queries.

The above solution naturally requires a high storage overhead in return for improving retrieval performance. That is, in Example 2.1, the instances of `TA` should appear in the four indexes built on the full extents of `Person`, `Faculty`, and `Student`, as well as `TA`. Moreover, if any new TA object is created (or deleted), or the index field is changed, four indexes must be updated instead of one. We should, therefore, investigate how we can maximally improve retrieval performance with minimal space and update overheads. It is a question of a trade-off between space and time.

However, it is unreasonable to apply a single trade-off point to all computing environments, since an optimal point is very dependent on a real environment, such as the number of indexing elements and the distribution of key values. The space and update overhead is proportional to how many times instances are replicated in the set of indexes – in this paper, we will call this *index replication*. In this regard, we consider the degree of index replication as a variable and introduce the *index set* scheme that finds an optimal set of indexes with a given degree of replication.

[1] There are a few alternatives for representing objects in object database systems [11,16]. In this paper, we assume that each class keeps only its direct instances, since this approach is widely accepted.

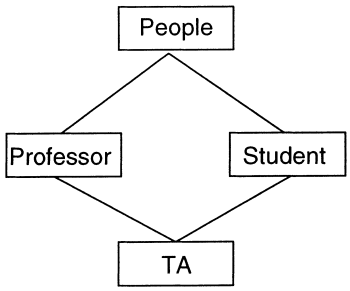


Fig. 1. `People` class hierarchy.

Example 2.2. All possible indexes with classes C_1, C_2 and C_3 are as follows:

$$M = \{\{C_1C_2C_3\}\ \{C_1C_2\}\ \{C_1C_3\}\ \{C_2C_3\}\ \{C_1\}\ \{C_2\}\ \{C_3\}\},$$

where $\{C_i\}$ denotes an index on the extent of class C_i, and $\{C_j \cdots C_k\}$, an index on the extent of classes $C_j \ldots$ and C_k, respectively. The problem here is to find an index set O, a subset of the set M, that minimizes the query costs and satisfies the constraint r on the degree of index replication. The degree of index replication of an index set is r when the extent of a class is indexed by at most r indexes in the set.

It is intractable to traverse all of the search space, since the number of possible index sets with n classes is $2^{2^n-1} - 1$. We therefore devise a greedy algorithm to find a near-optimal index set.

2.3. The greedy algorithm

In this section, we first present a basic greedy algorithm for finding an index set without considering replication. We then extend the basic version so it can handle index replications.

In general, when the query frequency of a class is higher than that of its subclasses or when a class has many more instances than its subclasses, it is beneficial for the retrieval performance to maintain one index for a class and all its subclasses rather than several indexes for each class. On the contrary, it is helpful to build indexes on each class individually for the converse cases [10]. It depends on the query frequency, the number of instances, and the distribution of key values whether we should build indexes on every class or one index on all classes. Based on this insight, our greedy algorithm computes the query cost of every index set and changes the set successively towards the maximal benefit to query performance.

The outline of our greedy algorithm is as follows. First, we build an initial index set that consists of single indexes on each class – not on full extents. Then, we compute the benefit of merging indexes in the index set by considering how the merged index can improve performance when evaluating queries. We use the term 'merging indexes' when we build one index on all classes on which the original indexes are built. For each index merge, we compare the cost of query evaluation and then select the cheapest one. If the selected merge helps, we include it in the new index set instead of the original indexes, repeating this process until no possible index merge provides any more benefit. The resulting index set is the greedy optimal index set.

The detailed algorithm is shown in Algorithm 1.

Algorithm 1. The greedy algorithm
$O_{\text{new}} := \{\{C_1\}\{C_2\}\cdots\{C_n\}\}$
repeat
$O_{\text{old}} := O_{\text{new}}$
for all I, J such that $I, J \in O_{\text{old}}$ and $I \neq J$ **do**
$O := (O_{\text{old}} \cup \{I \cup J\}) - \{IJ\}$
if $\texttt{Retrieval\_Cost}_O < \texttt{Retrieval\_Cost}_{O_{\text{new}}}$ **then**
$O_{\text{new}} := O$
end if
end for
until $O_{\text{old}} = O_{\text{new}}$
O_{new} is the greedy optimal index set

Example 2.3. In this example, we illustrate the execution of our greedy algorithm using the class hierarchy of Fig. 2. The number of instances of each class is also given in parentheses.

Beginning with the initial index set $O = \{\{1\}\ \{2\}\ \{3\}\ \{4\}\}$, we make successive choices of indexes to merge. The `Retrieval_Cost` of each possible cases of merging two indexes in the first iteration is given in the column 'Iteration 1' of Table 1. The `Retrieval_Cost` will be elaborated in Section 3. In the first round, the index set O_4 shows the best performance, so we pick this set as a new index set.

Next, we examine all candidates that can be derived from the previous selection and calculate the costs of each set. The costs are shown in the second column in Table 1. In the second iteration, the winner is the index set O_1.

In our third iteration, there is only one case to evaluate, as shown in column 'Iteration 3'. However, the new index set gives no improvement to the retrieval performance and the greedy selection is therefore $O = \{\{123\}\ \{4\}\}$.

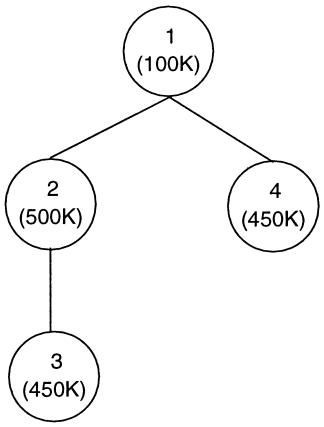


Fig. 2. Example class hierarchy.

Table 1
`Retrieval_Cost` of possible sets at each iteration

	Iteration 1		Iteration 2		Iteration 3	
O	{ {1} {2} {3} {4} }	1.2944	{ {1} {2 3} {4} }	1.2549	{ {1 2 3} {4} }	1.2527
O_1	{ {1 2} {3} {4} }	1.2782	{ {1 2 3} {4} }	1.2527	{ {1 2 3 4} }	1.5985
O_2	{ {1 3} {2} {4} }	1.3291	{ {1 4} {2 3} }	1.2647		
O_3	{ {1 4} {2} {3} }	1.3043	{ {1} {2 3 4} }	1.5786		
O_5	{ {1} {2 4} {3} }	1.4204				
O_4	{ {1} {2 3} {4} }	1.2549				
O_6	{ {1} {2} {3 4} }	1.5375				

The basic algorithm, however, does not allow multiple indexing for a class and this restriction is likely to limit performance improvement. Instead, we can consider replication when we compute the performance enhancement of merging two indexes. Algorithm 2 shows the extended greedy algorithm for handling index replication. In every iteration of the algorithm, we also examine the candidate sets where the original indexes are retained as well as the merged version after merging. This is represented by the innermost loop in the Algorithm 2. Here, we evaluate four candidate configurations: when preserving all the original indexes, when preserving only one of them, and when preserving neither of them.

Algorithm 2. The extended greedy algorithm
$//r_{C_i}$ is the replication count for a class C_i. That is, r_{C_i} is the number of replications of C_i in an index set. r is the replication constraint.

$O_{\text{new}} := \{\{C_1\}\{C_2\}\cdots\{C_n\}\}$
repeat
  $O_{\text{old}} := O_{\text{new}}$
  for all I, J such that $I, J \in O_{\text{old}}$ and $I \neq J$ **do**
    for all K such that $K \in \{\{IJ\}\ \{I\}\ \{J\}\ \emptyset\}$ **do**
      $O := (O_{\text{old}} \cup \{I \cup J\}) - K$
      calculate the replication count r_{C_i} of O for all classes C_i
      $O := (O - \{\{C_i\}\ \mid\ r_{C_i} > r\}) \cup \{\{C_i\}\ \mid\ r_{C_i} < r\}$ where $1 \leqslant i \leqslant n$
      if $\texttt{Retrieval\_Cost}_O < \texttt{Retrieval\_Cost}_{O_{\text{new}}}$ and O still satisfies the constraint r **then**
        $O_{\text{new}} := O$
      end if
    end for
  end for
until $O_{\text{old}} = O_{\text{new}}$
remove useless indexes from O_{new}
resulting O_{new} is the greedy optimal selection

In the extended greedy algorithm, we also add a single-class index on each class providing the replication constraint allows it. This is for cases in which the newly added indexes may improve

query performance directly or indirectly. That is, they may be involved in a new merge that enhances performance.

It should be noted that the extended algorithm may produce an index set that has unnecessary indexes. This is because, as mentioned above, we preserve indexes that were merged whenever possible. We do not remove the original indexes if the replication constraint allows it and performance does not degrade, in view of the possibility that they may provide performance enhancement in later loops. Therefore, we should delete the unnecessary indexes from the resulting index set of the greedy algorithm.

This can be achieved by removing indexes that do not affect query performance.

Example 2.4. In this example, we show the execution of the extended greedy algorithm with a replication constraint of 2 on the class hierarchy of Fig. 2. The execution of our extended algorithm is summarized in Table 2. Only one of the index sets resulting from merging is given if the results are identical.

As in the basic greedy algorithm, we calculate the benefit of each merge, but more candidates are examined by the extended algorithm, as we have described. For example, the index sets

Table 2
`Retrieval_Cost` of possible sets at each iteration

Iteration 1		
O	{ {1} {2} {3} {4} }	1.2944
O_1	{ {1} {2} {3} {4} {1 2} }	1.2599
O_2	{ {1} {2} {3} {4} {1 3} }	1.2646
O_3	{ {1} {2} {3} {4} {1 4} }	1.2646
O_4	{ {1} {2} {3} {4} {2 3} }	1.1196
O_5	{ {1} {2} {3} {4} {2 4} }	1.2162
Iteration 2		
O	{ {1} {2} {3} {4} {2 3} }	1.1196
O_1	{ {1} {3} {4} {2 3} {1 2} }	1.1196
O_2	{ {1} {2} {4} {2 3} {1 3} }	1.1593
O_3	{ {1} {2} {3} {4} {2 3} {1 4} }	1.0898
$O_{4,1}$	{ {1} {4} {2 3} {1 2 3} }	1.2169
$O_{4,2}$	{ {1} {2} {3} {4} {1 2 3} }	1.0954
O_5	{ {1} {3} {4} {2 3} {2 4} }	1.1196
O_6	{ {1} {2} {4} {2 3} {3 4} }	1.2402
$O_{7,1}$	{ {1} {4} {2 3} {2 3 4} }	1.1733
$O_{7,2}$	{ {1} {2} {3} {4} {2 3 4} }	1.1023
	⋮	
Iteration n		
O	{ {1} {2} {3} {4} {1 2 3 4} }	1.0780
O_1	{ {3} {4} {1 2 3 4} {1 2} }	1.0780
O_2	{ {2} {4} {1 2 3 4} {1 3} }	1.1178
O_3	{ {2} {3} {1 2 3 4} {1 4} }	1.1113
O_4	{ {1} {4} {1 2 3 4} {2 3} }	1.1353
O_5	{ {1} {4} {1 2 3 4}{2 4} }	1.2133
O_6	{ {1} {2} {1 2 3 4} {3 4} }	1.3192

$O_{4,1}$ and $O_{4,2}$ result from merging indexes {1} and {2 3}. The index set $O_{4,1}$ is the case in which all of the original indexes ({1} and {2 3}) are preserved and set $O_{4,2}$ is when only {1} is retained. We do not show results with the cases in which only {2 3} is retained or when none of the original indexes are retained, since they are the same as the cases $O_{4,1}$ and $O_{4,2}$, respectively.

After several applications of the greedy algorithm in the same manner, we reach the last iteration, n, where we cannot find any new index set that improves the cost of the previous result. Thus, the index set $O = \{\{1\}\ \{2\}\ \{3\}\ \{4\}\ \{1\ 2\ 3\ 4\}\}$ is the greedy selection. However, we should remove the indexes {1} and {2} from the results, since they do not contribute.

Therefore, the final index set is $O = \{\{3\}\ \{4\}\ \{1\ 2\ 3\ 4\}\}$.

2.4. Query evaluation

A class may be covered by several members of an index set simultaneously. Thus, a query can be evaluated in several ways and a query optimizer should select the indexes that allow a given query to be evaluated with minimal cost. The optimal execution plan can be obtained by the following scheme. First, the indexes that are necessary for the query are selected. Then, all of the combinations of indexes that are not thus far used are traversed and one of them is selected, such that it requires minimal cost for query evaluation. Although the search space increases exponentially with the number of classes, the number of indexes in the given index set is not very large. Also, once we achieve optimal execution plans for each target class, we can use them for all subsequent queries without recreating them. Thus, the method of finding execution plans does not greatly affect the performance of query evaluation.

As an alternative, a greedy method again seems appropriate in finding optimal query execution plans. For instance, we can choose execution plans by choosing indexes that cover the most classes in turn.

3. The cost model

In this section, we present a cost model that evaluates the retrieval and storage costs of an index set. We first describe the basic assumptions on our cost model and preliminary parameters. Then we derive the retrieval cost and the storage cost from them. Finally, we show the results of a few experiments for the verification of our cost model.

Our index set uses the structure of the class-hierarchy index for each index, which is a variation of the B^+-tree structure where class `ID`s are stored with index entries in the leaf nodes. Therefore, we have developed our cost model based on the discussion in [12], which proposed the cost model for the class-hierarchy index.

We make the following assumptions for our cost model:

- All key values have the same (average) length.
- The key values of an attribute are uniformly distributed among the instances of a class.
- Leaf-node records are either all smaller than the size of an index page or all larger.

3.1. Parameters

Database parameters:

- D_{c_j} – number of distinct values in class C_j,

- D_i – number of distinct values in index i,
- N_{c_j} – cardinality of class C_j,
- N_i – sum of the cardinalities of classes in index i,
- N – total number of instances in the database

$$N = \sum_{\text{for all classes}} N_{C_j},$$

- K_i – average number of elements contained in an attribute of index i

$$K_i = N_i / D_i,$$

- NC_i – average number of classes for an index record of index i

$$NC_i = \sum_{\text{for all classes in index } i} \frac{D_{c_j}}{D_i},$$

Index parameters:

- P – size of an index page,
- f – average fanout of an internal node,
- kl – average length of a value for an indexed attribute,
- XL_i – average length of a leaf-node record for index i

$$XL_i = \text{header\_length} + kl + (\text{sizeof}(\text{CLASSID}) + \text{sizeof}(\text{offset}) + \text{sizeof}(\text{number\_of\_OIDs})) \times NC_i + \text{sizeof}(\text{OID}) \times K_i,$$

where *header* consists of record_length, key_length, overflow_page_id and number_of_classes,

- LP_i – number of leaf-node pages for index i (excluding overflow pages),
- OP_i – number of overflow pages for index i

$$\begin{aligned} &\text{if } XL_i \leqslant P \quad LP_i = \lceil (D_i \times XL_i)/P \rceil, \\ &\text{if } XL_i > P \quad LP_i = D_i \\ &\qquad\qquad LP_i + OP_i = D_i \times \lceil XL_i/P \rceil, \end{aligned}$$

- H_i – internal height of index i (excluding the leaf-node level)

$$H_i = \text{the number of terms in } (LP_i + \lceil LP_i/f \rceil + \lceil \lceil LP_i/f \rceil / f \rceil + \cdots + 1).$$

3.2. Storage cost model

The storage cost for index i is given by the following equation:

$$SC_i = \begin{cases} LP_i + (\lceil LP_i/f \rceil + \lceil \lceil LP_i/f \rceil / f \rceil + \cdots + 1) & \text{if } XL_i \leqslant P, \\ LP_i + OP_i + (\lceil LP_i/f \rceil + \lceil \lceil LP_i/f \rceil / f \rceil + \cdots + 1) & \text{if } XL_i > P. \end{cases}$$

Therefore, the total storage cost required for an index set is given by

$$SC = \sum_{\text{for all indexes in the set}} SC_i.$$

3.3. Retrieval cost model

Single key query evaluation: The number of index pages accessed to evaluate a single key query is obviously the height of the index used. Therefore, the retrieval cost for index i is

$$RC_i^{\text{single}} = \begin{cases} H_i + 1 & \text{if } XL_i \leqslant P, \\ H_i + \lceil XL_i/P \rceil & \text{if } XL_i > P, \end{cases}$$

Thus, the average number of pages accessed for an index set for a query q is

$$RC^{\text{single}} = \sum_{\text{for all indexes used by } q} RC_i^{\text{single}}.$$

Range query evaluation: The retrieval cost for a range query is proportional to the range specified for a given query. Thus, we can formulate the number of pages to be fetched for index i as follows:

$$RC_i^{\text{range}} = \begin{cases} H_i + \lceil \text{query\_range} \times LP_i \rceil & \text{if } XL_i \leqslant P, \\ H_i + \lceil \text{query\_range} \times (LP_i + OP_i) \rceil & \text{if } XL_i > P. \end{cases}$$

The total retrieval cost for query q is given by

$$RC^{\text{range}} = \sum_{\text{for all indexes used by } q} RC_i^{\text{range}}.$$

3.4. Average retrieval cost ratio

The performance of an index set is very dependent on the target classes. Therefore, we need to introduce a single metric for performance comparison between index sets. For this consideration, we define average retrieval cost ratio (ARCR), which is the ratio of the average retrieval cost provided by an index set relative to that of the fully-replicated index configuration. The fully-replicated index configuration here is a set of indexes on the full extent of each class.

Retrieval performance is very dependent on query patterns in a real computing environment. This means that we should apply query patterns in real world situations to ARCR. Important considerations in cost evaluation are the ratio of the single-key query and the query ratio of each class.

The retrieval cost ratio R_{C_i} of each class C_i is given by

$$R_{C_i} = \frac{\sum_{\text{for all index } I_j \text{used for a query on } C_i} H_{I_j}}{H_{I_{C_i}^{\text{full}}}} \times \text{single point query ratio} + \frac{\sum_{\text{for all index } I_j \text{used for a query on } C_i} LP_{I_j}}{LP_{I_{C_i}^{\text{full}}}} \times (1 - \text{single point query ratio}),$$

where $I_{C_i}^{\text{full}}$ is the index on the full extent of a class C_i.

By applying the query weight, we formulate the ARCR as follows:

$$\text{ARCR} = \sum_{\forall C_i} R_{C_i} \times \text{query weight on } C_i,$$

ARCR is used as the `Retrieval_Cost` in our algorithms.

3.5. Verification of the cost model

We performed several experiments for the justification of our cost model. In these experiments, we examined the number of index pages accessed for queries on three classes with 100,000 instances each. We ran the queries using two different index configurations, one index on all three classes and three single-class indexes on each class, varying the range of key values, the number of instances, the single key query ratio, and the query range. Table 3 shows one of the results: the performance ratio of two index configurations. The results were almost the same for the other experiments with various configurations. The table also includes the expected ratio that is calculated with our cost model. Here, the error ratio is computed by

$$\frac{\text{experimental result} - \text{expected result}}{\text{expected result}} \times 100.$$

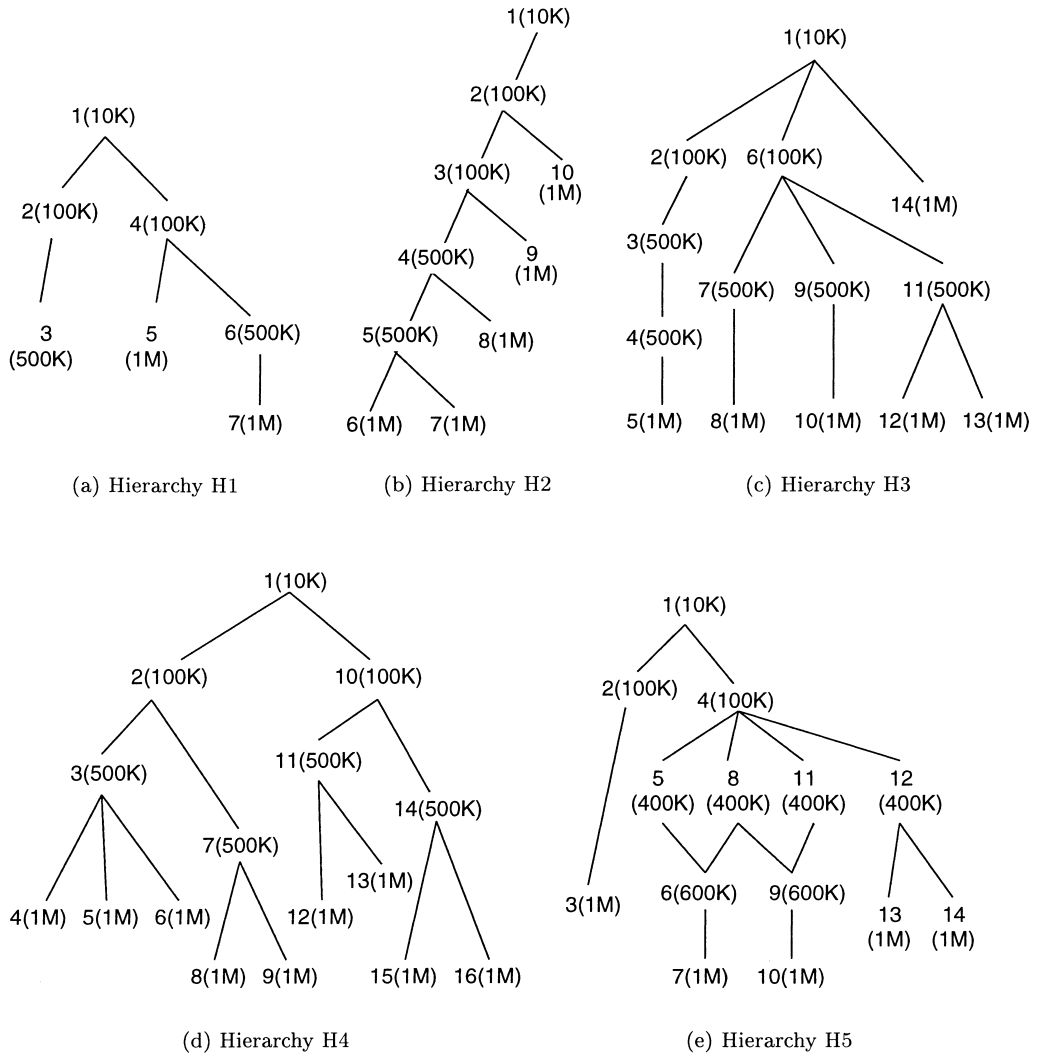
As may be seen in Table 3, the error ratios between the experimental and the expected results are always within 5%. The error ratio increases somewhat in cases where the query range is narrow or the single-key query ratio is low. This is due to the assumption of uniform distribution, which is not met when the size of an answer for a query is small.

4. Performance analysis

We evaluated the performance of our index set on typical class hierarchies depicted in Fig. 3. These include a simple hierarchy of size five (hierarchy `H1`), a skinny hierarchy (hierarchy `H2`), a bushy hierarchy (hierarchy `H4`), and a combined hierarchy (hierarchy `H3`). We also used a 14-class hierarchy with multiple inheritance (hierarchy `H5`). Our performance comparison was conducted for the index set, the class-division scheme and class-hierarchy indexing. We also performed the same experiments for single-class indexing, where an index is maintained on each class, in order to compare the behaviors of class-hierarchy indexing and the single-class indexing schemes.

Table 3
The verification of our cost model

Single-key query ratio (%)	5			10			95		
Query range (%)	1	5	10	1	5	10	1	5	10
Expected result	1.63	1.63	1.63	1.71	1.71	1.71	2.93	2.93	2.93
Experimental result	1.71	1.61	1.61	1.75	1.70	1.68	2.90	2.90	2.90
Error ratio (%)	+4.9	−1.2	−1.2	+2.3	−0.6	−1.8	−1.0	−1.0	−1.0



(a) Hierarchy H1 (b) Hierarchy H2 (c) Hierarchy H3

(d) Hierarchy H4 (e) Hierarchy H5

Fig. 3. Class hierarchies used in the experiments.

For each indexing scheme, we computed the relative ratio of (a) retrieval performance for a query on each class, (b) average retrieval performance, and (c) total storage cost, against an index set with the fully-replicated index configuration using our cost model presented in Section 3. Similarly to the average retrieval cost ratio, the total storage cost ratio is the metric relative to the fully-replicated index configuration. Table 4 lists the parameter settings used in the experiments. [2] In all experiments, the single key query ratio is 10% and queries are distributed uniformly over all

[2] For the single-class index configuration, we removed the fields CLASSID and number_of_classes from leaf-node record structures.

Table 4
Parameters used in the experiment

Parameters	Values
P	4096
f	$255 = (4096/(8+4)) \times (3/4))$
kl	8
sizeof(OID)	8
sizeof(CLASSID)	4
sizeof(Page_id)	4
sizeof(offset)	2
sizeof(number_of_OIDs)	2
sizeof(record_length)	2
sizeof(key_length)	2
sizeof(number_of_classes)	2

classes. The distribution of key values was totally inclusive in the range of 1–500,000 and we give the number of instances of each class in parentheses in Fig. 3.

4.1. Comparison

We first applied our greedy algorithm to each hierarchy, varying the replication constraint. Table 5 shows the resulting configurations of our index sets and the corresponding replication constraints. It also shows the results of the class-division algorithm. IS-*i* denotes an index set with replication *i*. CD, CH and SI represent indexing by class-division [14], class-hierarchy indexing [12], and single-class indexing [12], respectively.

The graphs in Fig. 4 illustrate the performance of each indexing technique. The graphs on the left side show the retrieval performance for queries on each class (including its subclasses). The graphs on the right show the average retrieval cost ratio and total storage cost ratio of each technique.

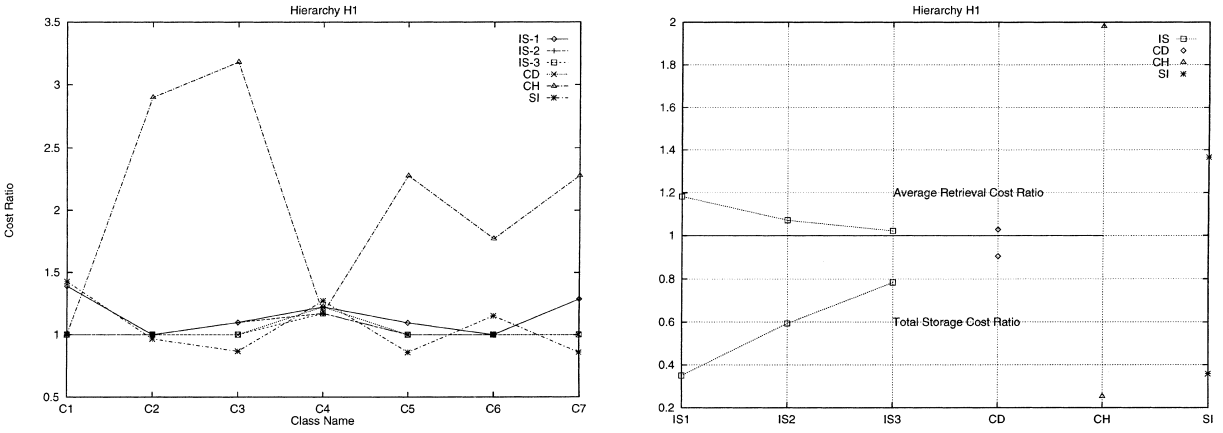
We will comment on several aspects of each technique. First, the performance curves of class-hierarchy indexing and single-class indexing are exactly opposite to each other. Obviously, CH should fetch many more index pages as a query targets fewer classes, although it is the best solution for a query on an entire class hierarchy. Therefore, the performance of CH is extremely bad for leaf classes. On the contrary, SI is efficient providing there are only a few classes in the target class domain. This result reveals that neither of these two indexing schemes is suitable for queries on class hierarchies.

As we expected, indexing by class-division achieves good retrieval performance through the replication of indexes. However, the index configuration of the class-division scheme requires more space and update overhead than our index set scheme. That is, the retrieval performance of class-division is worse than that of an index set with similar or smaller storage cost. For example, in Fig. 4(a), while the average retrieval cost ratio and total storage cost ratio of IS-3 for hierarchy H2 are 1.022 and 0.784, respectively, CD shows the total storage cost ratio 0.951, higher than IS-3, and the average retrieval cost ratio of 1.028 for CD is worse than that for IS-3.

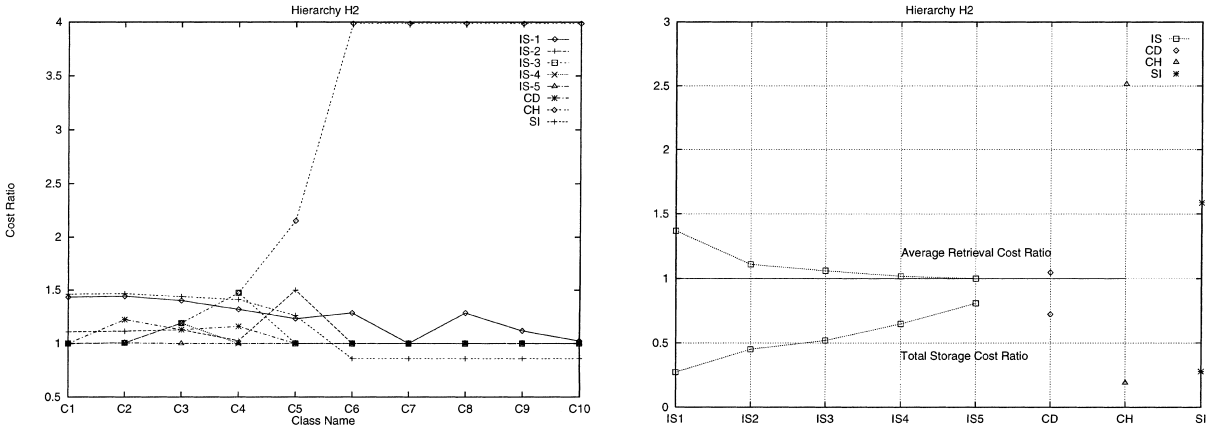
Table 5
Index configurations for the index set and class-division methods

	Index configuration
(a) *Hierarchy* H1	
IS-1	{ { 1 – 3 } { 4 5 } { 6 7 } }
IS-2	{ { 1 – 7 } { 1 – 3 } { 6 7 } { 5 } }
IS-3	{ { 1 – 7 } { 2 3 } { 6 7 } { 3 } { 5 } { 7 } }
CD	{ { 1 – 7 } { 2 3 } { 4 5 } { 6 7 } { 3 } { 5 } { 7 } }
(b) *Hierarchy* H2	
IS-1	{ { 1 10} { 2 3 9 } { 4 8 } { 5 6 } { 7 } }
IS-2	{ { 1 2 9 10} { 3 – 8 } { 6 } { 7 } { 8 } { 9 } { 10 } }
IS-3	{ { 1 – 10} { 5 – 7 } { 6 } { 7 } { 8 } { 9 } { 10 } }
IS-4	{ { 1 – 10} { 4 – 8 } { 5 – 7 } { 6 } { 7 } { 8 } { 9 } { 10 } }
IS-5	{ { 1 – 10} { 3 – 9 } { 4 – 8 } { 5 – 7 } { 6 } { 7 } { 8 } { 9 } { 10 } }
CD	{ { 1 – 10} { 3 4 8 9 } { 5 – 7 } { 2 10 } { 4 8 } { 6 } { 7 } { 8 } { 9 } { 10 } }
(c) *Hierarchy* H3	
IS-1	{ { 1 – 5 } { 6 12 } { 7 8 } { 9 10 } { 11 13 } { 14 } }
IS-2	{ { 1 – 5 } { 6 11 – 13 } { 7 8 } { 9 10 } { 5 } { 8 } { 10 } { 12 } { 13 } { 14 } }
IS-3	{ { 1 – 5 } { 6 – 14 } { 4 5 } { 7 8 } { 9 10 } { 11 12 } { 5 } { 8 } { 10 } { 12 } { 13 } { 14 } }
IS-4	{ { 1 – 5 } { 6 – 14 } { 6 – 13 } { 3 – 5 } { 4 5 } { 7 8 } { 9 10 } { 11 – 13 } { 5 } { 8 } { 10 } { 12 } { 13 } { 14 } }
CD	{ { 1 – 14 } { 2 – 5 } { 6 – 10 } { 11 – 13 } { 4 5 } { 7 8 } { 9 10 } { 3 } { 5 } { 8 } { 10 } { 12 } { 13 } { 14 } }
(d) *Hierarchy* H4	
IS-1	{ { 1 – 4 } { 7 8 } { 10 13 } { 11 12 } { 14 16 } { 5 } { 6 } { 9 } { 15 } }
IS-2	{ { 1 – 6 } { 7 – 9 } { 10 – 13 } { 14 – 16 } { 4 } { 5 } { 6 } { 8 } { 9 } { 12 } { 13 } { 15 } { 16 } }
IS-3	{ { 1 – 9 } { 10 – 16 } { 3 – 6 } { 7 – 9 } { 11 – 13 } { 14 – 16 } { 4 } { 5 } { 6 } { 8 } { 9 } { 12 } { 13 } { 15 } { 16 } }
CD	{ { 1 – 16 } { 5 – 8 } { 3 4 } { 5 6 } { 7 8 } { 11 12 } { 15 16 } { 2 } { 4 } { 5 } { 6 } { 8 } { 9 } { 10 } { 12 } { 13 } { 14 } { 15 } { 16 } }
(e) *Hierarchy* H5	
IS-1	{ { 1 – 3 } { 5 – 7 } { 4 8 – 11 } { 12 14 } { 13 } }
IS-2	{ { 1 – 3 } { 4 12 – 14 } { 5 – 7 } { 8 – 11 } { 3 } { 7 } { 10 } { 13 } { 14 } }
IS-3	{ { 1 – 4 12 – 14 } { 1 – 3 } { 4 12 – 14 } { 5 – 8 } { 9 – 11 } { 6 7 } { 9 10 } { 3 } { 7 } { 10 } { 13 } { 14 } }
IS-4	{ { 1 5 – 11 } { 2 – 4 12 – 14 } { 4 12 – 14 } { 2 3 } { 5 – 7 } { 9 – 11 } { 12 – 14 } { 6 7 } { 9 10 } { 3 } { 7 } { 10 } { 13 } { 14 } }
IS-5	{ { 1 – 14 } { 1 – 3 } { 5 – 11 } { 12 – 14 } { 5 – 7 } { 9 – 11 } { 6 7 } { 9 10 } { 3 } { 7 } { 10 } { 13 } { 14 } }

As for our index set, not surprisingly the retrieval cost decreases as IS uses more replications. However, the performance enhancement is not linear with storage overhead, as is apparent in Fig. 4. Therefore, it is important to select a degree of replication that provides a significant speed up with small overhead. In our experiments, IS with a replication constraint of 2 or 3 offers good performance improvement over a partitioned configuration. However, we experienced only small performance enhancements with more replications.



(a) Hierarchy H1



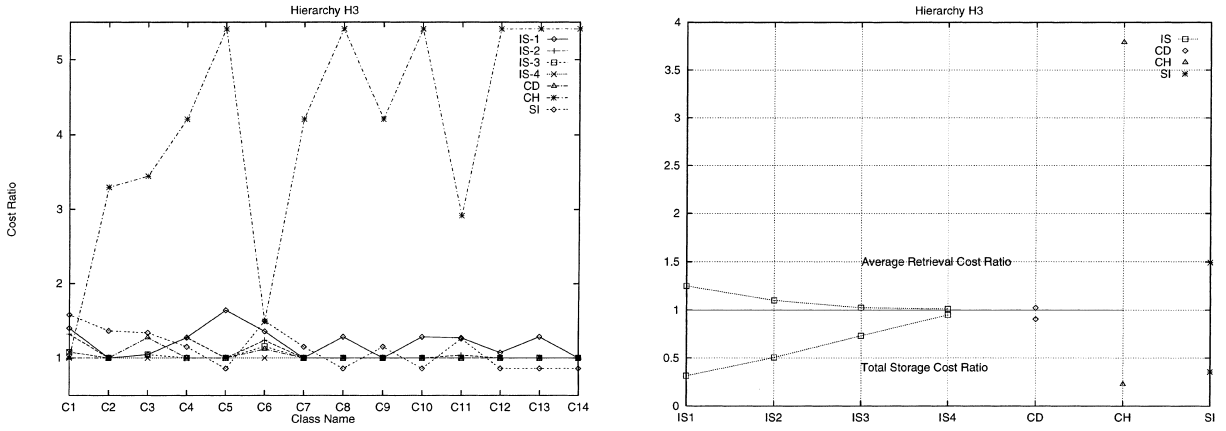
(b) Hierarchy H2

Fig. 4. The performance of indexing techniques.

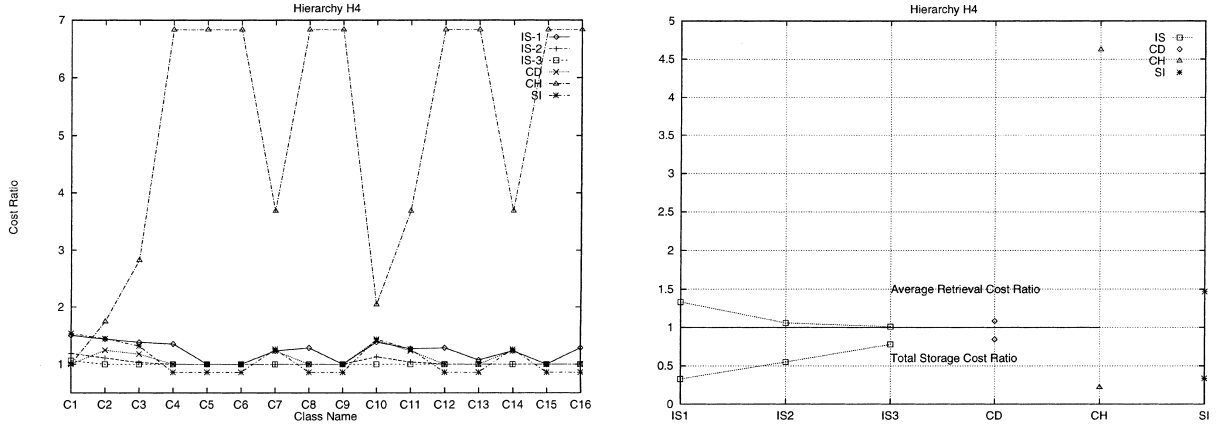
One interesting result is that on the skinny hierarchy H2, IS-2 performance is sharply improved over IS-1. However, the performance is nearly unchanged with more replicates. This is because with replication 2, the index set could provide optimal performance on leaf classes and also reasonably good retrieval performance on super classes in this skinny hierarchy.

Our index set technique also presents good results on the class hierarchy with multiple inheritance. We do not show the result of the class-division scheme for hierarchy H5, since it cannot be used on a hierarchy with multiple inheritance.

Finally, our IS-1 schemes also provide uniformly good performance in comparison with CH and SI schemes over all the example hierarchies. These results indicate that we can obtain performance enhancement simply by using a proper index configuration without any storage and update overhead and our greedy algorithm can find good index sets even for nonreplication cases.



(c) Hierarchy H3

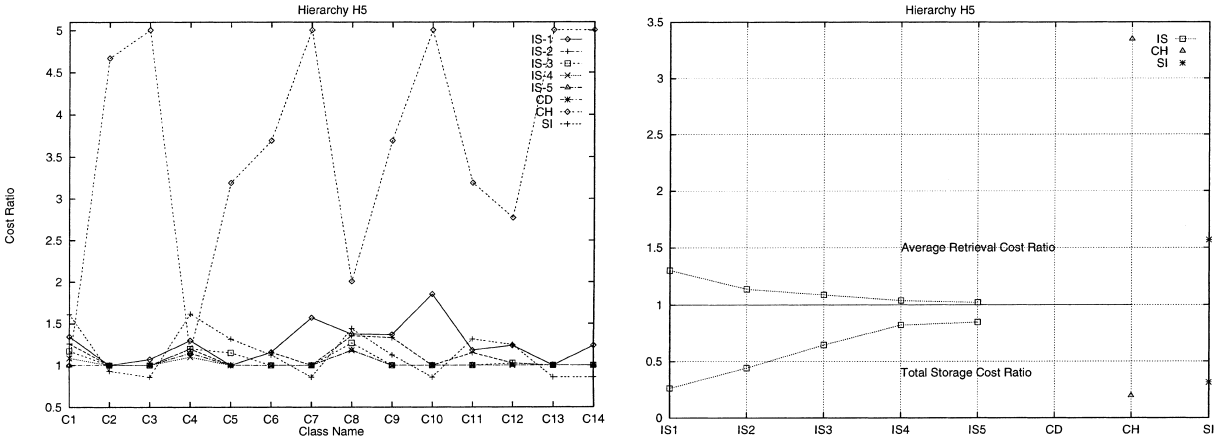


(d) Hierarchy H4

Fig. 4. (*continued*).

4.2. Performance of the greedy algorithm

The results of our experiments indicate that our greedy algorithm can generate good index sets on various hierarchies. However, the greedy algorithm may naturally stop at a local optimum. To avoid this behavior, we can use a look-ahead technique in the greedy algorithm. In additional experiments for evaluating the performance of our greedy algorithm, we obtained optimal solutions with 1-look-ahead in almost all cases. Even without look-ahead, our algorithm also gives optimal configurations in most cases. In addition, locally optimal results are comparable to optimal ones. This is due to the fact that queries on a class hierarchy usually target classes that are connected directly or indirectly with each other. The greedy algorithm can, therefore, find a good configuration simply by merging indexes step by step.



(e) Hierarchy H5

Fig. 4. (*continued*).

5. Conclusion and future work

The performance of the object database systems is the key to the commercial success of them and efficient indexing on a class hierarchy is essential for improving the performance.

In this study, we present the index set scheme, which finds a near optimal index configuration, within a specified replication constraint using the greedy algorithm. The index set provides a good index configuration for any real database environment, since it considers the distribution of key values, as well as query patterns such as query weight on each class. Essentially, our index set can also be easily applied to a system since it uses the B^+-tree structure.

We have developed a cost model and analyzed the performance of the new index technique with various class hierarchies. In these experiments, the index set showed the best retrieval performance than any other technique. In particular, the index set provided better space-time tradeoff than the class-division scheme. The index set scheme also showed very good retrieval performance even without replication in comparison to the class-hierarchy indexing and the single-class indexing schemes. Although the performance of the index set technique improved with more replications, we could get the most performance enhancement in the case of replication 2 or 3.

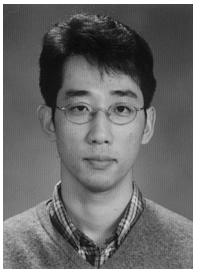
We are currently developing a new algorithm which uses both greedy algorithm and genetic algorithm to produce better index sets for nonreplication cases, which can provide relatively good performance without additional update cost. We are also devising a new method to gather the statistical informations more efficiently.

Acknowledgement

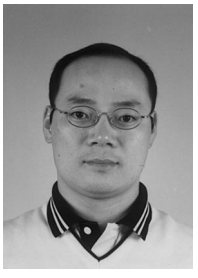
This research was supported by the Brain Korea 21 Project.

References

[1] E. Bertino, B. Catania, L. Chiesa, Definition and analysis of index organizations for object-oriented database systems, Information Systems 23(2) (1998).
[2] E. Bertino, P. Foscoli, Indexing organizations for object-oriented database systems, IEEE Trans. Knowledge Database Eng. 7 (2) (1995).
[3] E. Bertino, W. Kim, Indexing techniques for queries on nested objects, IEEE Trans. Knowledge Database Eng. 1(2) (1989).
[4] D. Comer, The ubiquitous B-tree, ACM Comput. Surveys 11 (2) (1969).
[5] E. Gudes, A uniform indexing scheme for object-oriented databases, in: Proceedings of the International Conference on Data Engineering, 1996.
[6] A. Guttman, R-TREES: a dynamic index structure for spatial searching, in: Proceedings ACM SIGMOD Conference, 1984.
[7] K.A. Hua, Object skeletons: an efficient navigation structure for object-oriented database systems, in: Proceedings of the International Conference on Data Engineering, 1994.
[8] Y. Ishikawa, H. Kitagawa, N. Ohbo. Evaluation of signature files as set access facilities in OODBs, in: Proceedings of the ACM SIGMOD International Converence on Management of Data Washington, DC, USA, 1993.
[9] A. Kemper, G. Moerkotte, Access support relations: an indexing method for object bases, Information Systems 17(2) (1992).
[10] C. Kilger, G. Moerkotte, Indexing multiple sets, in: Proceedings of the International Conference on Very Large Data Base, 1994.
[11] W. Kim, Introduction to Object-Oriented Databases, MIT Press, Cambridge, 1990.
[12] W. Kim, K.-C. Kim, A. Dale, Object-oriented Concepts Databases and Applications, Addison-Wesley, Wokingham, 1989 (Chapter: Indexing techniques for object-oriented databases).
[13] C.C. Low, B.C. Ooi, H. Lu, H-trees: a dynamic associative search index for OODB, in: Proceedings of the ACM SIGMOD International Conference on Management of Data San Diego, CA, 1992.
[14] S. Ramaswamy, P.C. Kanellakis, OODB indexing by class-division, in: Proceedings of the ACM SIGMOD International Conference on Management of Data San Jose, CA, USA, 1995.
[15] B. Sreenath, S. Seshadri, The hcC-tree: an efficient index structure for object oriented databases, in: Proceedings of the International Conference on Very Large Data Bases, 1992.
[16] M. Stonebraker, P. Brown, D. Moore, Object-relational DBMSs: tracking the next great wave, Morgan Kaufmann, Los Altos, CA, 1998.
[17] Z. Xie, J. Han, Join index hierarchies for supporting efficient navigations in object oriented databases, in: Proceedings of the International Conference on Very Large Data Bases, 1994.



Jung-Ho Ahn received his B.S., M.S., Ph.D. degrees in computer engineering from Seoul National University, Seoul, Korea, in 1991, 1993, and 1998, respectively. He is currently a senior engineer in the Telecommunication R&D Center at Samsung Electronics. His research interests include object-oriented databases, real-time databases, and telecommunication.



Ha-Joo Song received his B.S. and M.S. degree in computer engineering from Seoul National University, Seoul, Korea, in 1993 and 1995, respectively. He is currently enrolled in the Ph.D. program in computer engineering at Seoul National University. His research interests include object-oriented databases, transaction processing, and multimedia databases.

Hyoung-Joo Kim received his B.S. degree in computer engineering from Seoul National University, Seoul, Korea, in 1982 and his M.S. and Ph.D. in computer engineering from University of Texas at Austin in 1985 and 1988, respectively. He was an assistant professor of Georgia Institute of Technology, and is currently a professor in the Department of Computer Engineering at Seoul National University. His research interests include object-oriented databases, multimedia databases, HCI, and computer-aided software engineering.