ELSEVIER

Information and Software Technology 42 (2000) 731–741

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.nl/locate/infsof

Using genetic algorithms to work out index configurations for the class-hierarchy indexing in object databases☆

H.-J. Song*, J.-H. Ahn, H.-J. Kim

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

Received 2 January 1999; received in revised form 15 March 2000; accepted 31 March 2000

Abstract

Efficient indexing on a class hierarchy is essential for the achievement of high performance in query evaluation for object databases. In this paper, we present a practical indexing scheme, Partition Index Configuration Scheme (PINS), which provides good index configurations for any real database environment. PINS considers the distribution of key values, as well as query patterns such as query frequency on each class. PINS can easily be applied to any database system, since it uses the B^+-tree structure. We develop a cost model and, through experiments, demonstrate the performance of the proposed policy over various class hierarchies. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Partition index configuration scheme; Class-hierarchy index; Genetic algorithm; Object database

1. Introduction

1.1. Problems of indexing in object databases

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 [6] 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.

To support queries on class hierarchies, several index structures, including the H-tree [15] and the HcC-tree [17], have been proposed. However, these new structures are seldom used, since they are complex and require new concurrency mechanisms. A practical alternative was proposed, namely, an indexing scheme based on class division [16], 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 not 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, Partition Index Configuration Scheme (PINS), 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 [1,2,12–17]. The simplest approach for class-hierarchy indexing is the class-hierarchy index [14]. 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 [15], the HcC-tree [17], and the CG-tree [13]. However, as mentioned above, these approaches are new and it is unclear whether the performance gains justify the nonstandard data-structure introduced by them. In addition, they are lacking for concurrency control and recovery

☆ This research was supported by the Brain Korea 21 Project.

* Corresponding author. Fax: + 82-2-882-0269.
E-mail address: hjsong@oopsla.snu.ac.kr (H.-J. Song).

0950-5849/00/$ - see front matter © 2000 Elsevier Science B.V. All rights reserved.
PII: S0950-5849(00)00113-0

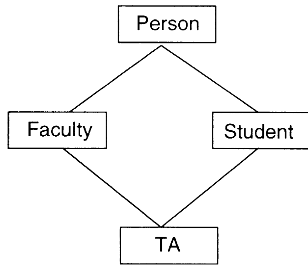

Fig. 1. An example class hierarchy.

schemes that are essential for them to be accepted in commercial database systems. Therefore, only the class-hierarchy index based on the B^+-tree structure has been widely used in real object database systems [16].

Ramaswamy and Kanellakis [16] 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 distribution of key values.

Many indexing techniques for queries on composite hierarchies have also been studied. Bertino and Kim [2] proposed the nested index, the path index, and the multi-index for queries on composite objects, and compared their performance. Ishikawa et al. [12] suggested an indexing scheme using the signature file technique, and Bertino and Foscoli [1] 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 Sections 2 and 3, we introduce our new indexing scheme, PINS, for indexing on class hierarchies and a method of using the genetic algorithm to support PINS. Section 4 describes the cost model used in this study, and Section 5 presents the results of the performance evaluation. Finally, conclusions from our study and areas for future research are given in Section 6.

2. The partition index configuration scheme

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

Example 1. A query "Find all students who are 21 years old'' on the class hierarchy of Fig. 1 has the search condition that he or she should be a `Student` as well as the predicate that he or she should be 21 years old. Here, all `TA`s can also be considered `Students` 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.

To enhance the performance of this kind of queries on class hierarchies, it is necessary to investigate a new indexing technique that considers 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 like the R-tree [10] does not provide satisfactory performance with the query on a class hierarchy, which is a special case of two-dimensional search [16]. Thus, we are sure that the practical alternative is to index using B^+-tree indexes with little modification, since the B^+-tree index is the best general-purpose index structure for one-dimensional queries.

Fig. 2 shows the leaf record structure of the modified B^+-tree which is used in the proposed scheme. It was first introduced in Ref. [14], and used in Refs. [1,16]. Since only the leaf records are slightly modified, the modified B^+-tree does not require a new concurrency control and recovery scheme.

As far as retrieval performance alone is concerned, the best solution is to build indexes on the full extent of each class. The *full extent* of a class is the set of instances of the class and all of its subclasses. Therefore, the best index configuration for Example 1 is the set of indexes on the full extent of each of `Person`, `Faculty`, `Student`, and `TA`; a set of indexes (I_{full})

$$I_{full} = \{\{Person\}\ \{Faculty\}\ \{Student\}\ \{TA\}\ \{Faculty\ TA\}\ \{Student\ TA\}\ \{Person\ Faculty\ Student\ TA\}\}$$

Here, $\{C\}$ denotes an index on the extent of class C, and $\{C_i \ldots C_j\}$, an index on the union of extents of class $C_i,\ldots,$ and C_j, respectively.

Using this index configuration, an index that covers the target class and all of its sub-classes can be used to process a query on a class. However, this solution naturally requires a high storage overhead in return for improving the retrieval performance. That is, in Example 1, the instances of `TA` should be replicated in the full indexes on `Person`, `Faculty`, and `Student`, as well as `TA`. Moreover, if

record header	*key value*	*# classes*	*(CID, offset, #OIDs) ... (CID, offset, #OIDs)*	$OID_1 \ldots OID_N$

Fig. 2. The leaf record structure of a modified B^+-tree index.

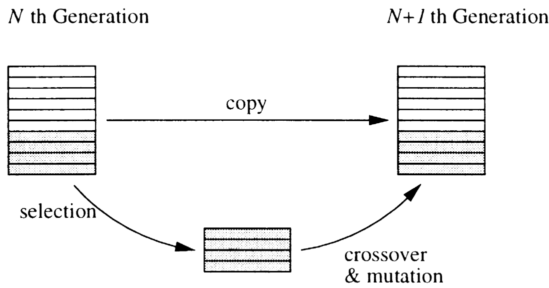

Fig. 3. Evolution process of genetic algorithm.

there is any new TA object created (or deleted), or the index field has been changed, four indexes must be updated instead of one. Considering that index objects are frequently used and have high update costs, this kind of index configuration will decrease the performance of DBMS significantly.

As an alternative, we introduce PINS as a new indexing scheme for object databases, which significantly improves the retrieval performance without additional update costs and high storage overhead. PINS uses several indexes, each of which consists of instances of groups of classes in a class hierarchy, but with no instance of a class being duplicated. PINS takes the class hierarchy, key distribution, query frequency on classes, and number of indexes to be used as input, and returns an index configuration that minimizes the expected retrieval cost. Given a class hierarchy with N classes, and the maximum number of indexes to be used, $i_{\max}$ ($1 \leq i_{max} \leq N$), PINS partitions the N classes into $i_{\max}$, or less than $i_{\max}$ disjoint subsets, and builds an index for each subset so that the expected retrieval cost is minimized.[1]

Example 2. For a class hierarchy in Fig. 1, we can obtain 15 possible partition index configurations as follows:

$I_1 = \{\{Person\}\ \{Faculty\}\ \{Student\}\ \{TA\}\}$

$I_2 = \{\{Person\ Faculty\}\ \{Student\}\ \{TA\}\}$

$I_3 = \{\{Person\ Student\}\ \{Faculty\}\ \{TA\}\}$

$I_4 = \{\{Person\ TA\}\ \{Faculty\}\ \{Student\}\}$

$\vdots$

$I_{14} = \{\{Person\ Faculty\}\ \{Student\ TA\}\}$

$I_{15} = \{\{Person\ TA\ Faculty\ Student\}\}$

Assuming that I_{14} is selected as the optimum partition index configuration, the index $\{Student\ TA\}$ can be used for looking up the class `Student` and `TA`, and both `{Person Faculty}` and `{Student TA}` for the class `Person` and `Faculty`, respectively.

However, it is intractable to traverse the entire search space since there are a huge number of cases.[2] Therefore, we have devised a genetic algorithm to find a near optimal partition index configuration.

3. PINS using genetic algorithm

In this section, we describe the genetic algorithm briefly and explain how we configured it to find the near optimal partition index configurations.

3.1. Genetic algorithm

Genetic algorithms (GAs) are search algorithms based on the mechanics of natural selection and natural genetics [8,18]. An individual corresponds to a solution for a problem, and consists of an array of gene values, its 'chromosome', and as in nature, an individual that is optimized for its environment is created by successive modification over a number of generations. Fig. 3 and Algorithm 1 illustrate the process of genetic algorithm. A algorithm starts with a set of initial individuals, called a population. This population then evolves gradually into different populations for several (typically hundreds of) iterations. At the end of the algorithm, it returns the best individual of the population as the solution to the problem [4]. In each iteration or generation, the evolution process is carried out in two steps, *crossover* and *mutation*. Firstly, with a predefined probability, some individuals of the population are selected, and they produce offspring by the crossover operator. Secondly, with a very low probability, these offspring are modified by the mutation operator. The mutation operator expands the search space by increasing the diversity of the individuals in the population. Then, by the replacement scheme, the modified offspring replace some of the individuals in the population, forming a new generation: the new population. This process is repeated until a certain condition is met. The number of iterations is frequently used as the termination condition.

Algorithm 1 Genetic algorithm

1: create initial population of fixed size;
2: **repeat**
3: choose *parent1* and *parent2* from population;
4: *offspring* = **crossover** (*parent1*, *parent2*);
5: **mutation** (*offspring*);
6: **replace** (population, *offspring*);
7: **until** (stopping condition);
8: report the best individual.

[1] Notice that PINS with $i_{max} = 1$ is the same as the class hierarchy index (CH) because in both cases only one index is maintained for a class hierarchy.

[2] Let $P(N)$ denote the number of possible partitioning with N classes in a class hierarchy. $P(N) = D(N)f(0)$, where $f(x) = e^{e^x - 1}$ and $D(N)f(x)$ is the N-th derivative of $f(x)$.

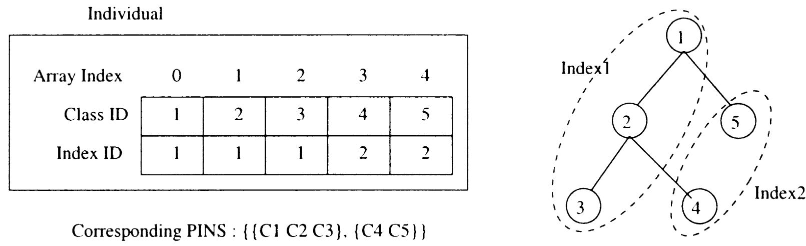

Fig. 4. An example of an individual.

Genetic algorithms have been frequently accepted as optimization methods in various database areas such as query optimization [9,11], database design [19], and mining [7]. Genetic algorithms have also proved their excellence in solving partitioning problems [3–5], and this is why we use GA to implement PINS.

3.2. Implementation

In this subsection, we describe the details of the genetic algorithm tailored to solve PINS.

3.2.1. Problem encoding

Individuals consist of an array of integers, of which the size is the same as the number of classes in the class hierarchy. The i-th array item corresponds to the i-th class in the hierarchy in depth-first order, and it contains an integer from 1 to the maximum number of indexes to be used (i_{max}). The integer denotes the index that the corresponding class should use. Thus, an individual denotes a partitioning (index configuration) of the classes in a class hierarchy. Fig. 4 illustrates an individual that denotes a possible partitioning in a class hierarchy composed of five classes. In this figure, the classes C_1, C_2, and C_3 use `INDEX1`, and classes C_4 and C_5 use `INDEX2`.

3.2.2. Initialization

We configure the genetic algorithm so that it creates a fixed number of initial individuals at random. Hence, every element in an individual is set to an integer value of which the range is between 1 and the number of indexes to be used. Usually a larger population implies a better final solution and a longer running time. Rather than using a fixed initial population size, we perform several experiments with varying the initial population, and select the best one.

3.2.3. Fitness function

The fitness of an individual is a measure of quality of its adaptation to the environment, and is calculated by the fitness function. In GAs, the optimization goal is to find the individual that has the highest fitness value, in other words, the fittest individual for the environment. The fitness value F_i of an individual i is calculated as follows:

$$F_i = (ARCR_w - ARCR_i) + (ARCR_w - ARCR_b)/3,$$

where $ARCR_b$, $ARCR_w$ and $ARCR_i$ denote the average retrieval cost ratio (ARCR) of the best individual, the worst individual, and individual i, respectively. Therefore fitness definition gives individuals with the higher fitness to be selected more often than the ones with the lower fitness, resulting in a proportional parent selection scheme. A clear definition of ARCR will be given in Section 4.

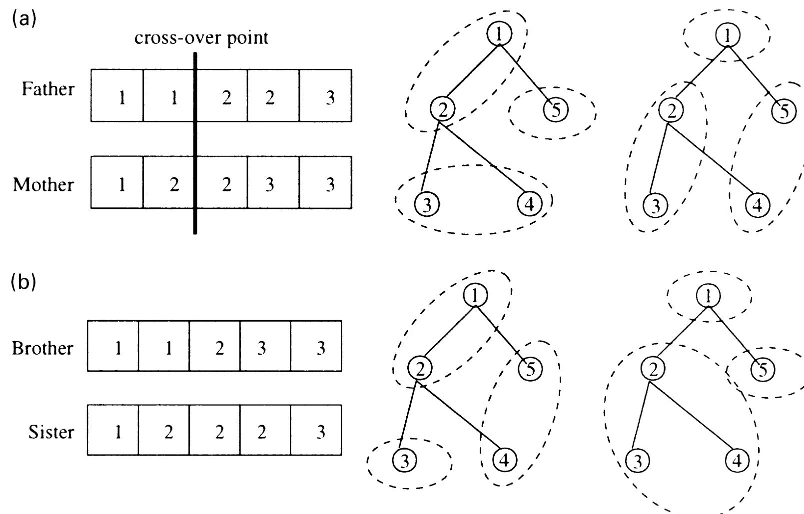

Fig. 5. Single point crossover operator.

3.2.4. Crossover operator

The crossover operator is the basic search method of GAs that produces offspring by combining parts of the chromosomes of the parents. We have used the single-point crossover operator. The single-point crossover operator randomly selects one cut point in the parent chromosome. The cut point divides the chromosome into two disjoint parts: the left and the right part. The left (right) part of the father chromosome and the right (left) part of the mother chromosome form a new offspring. Fig. 5 illustrates the process of crossover. In this figure, the point between the second and the third elements is selected as a crossover point. The graphs beside the individuals denote the corresponding partitioning before and after the crossover. We have used the proportional selection scheme to select the individuals to be used as the parents.

3.2.5. Mutation operator

Using only the crossover operator often results in premature convergence. Therefore, we applied the mutation operators to offspring produced by the crossover operator. The mutation operator visits array items in the chromosome and determines whether it should be changed or not, using a predefined mutation probability. When one item is selected, a randomly generated index number (or a partition number) is assigned to it. Although the mutation operator prevents premature convergence, if the probability of mutation is too large, the genetic algorithm becomes a random search algorithm. Therefore, the mutation probability must be a relatively small value—usually less than or equal to 1%.

3.3. Post-processing using the greedy algorithm

Generally, GAs are efficient at global search, but have poor performance in fine-tuning around a local optimum. Therefore, we post-processed the result of the genetic algorithm by the greedy optimization algorithm. Algorithm 2 shows the greedy algorithm we used for the post-processing. The greedy algorithm starts from the index configuration produced by the genetic algorithm. It then calculates ARCR of merging some of the indexes. We have used the term 'merging the indexes' when we built one index on all classes on which merged indexes are built. For each index merging, the greedy algorithm compares the retrieval cost and selects the cheapest one. If the selected merging helps, then the greedy algorithm repeats the same process from the new index configuration. This process is continued until there is no improvement. The resulting partition index configuration is the greedy optimum.

Algorithm 2 Greedy algorithm

1: $O_{new} := \{\{C_1\}\{C_2\}\ldots\{C_n\}\}$
2: **repeat**
3: $\quad O_{old} := O_{new}$
4: $\quad$ **for all** I, J such that $I, J \in O_{old}$ and $I \neq J$ **do**
5: $\quad\quad O := (O_{old} \cup \{\mathrm{I} \cup \mathrm{J}\}) - \{\mathrm{I\ J}\}$
6: $\quad\quad$ **if** $ARCR_O < ARCR_{O_{new}}$ **then**
7: $\quad\quad\quad O_{new} := O$
8: $\quad\quad$ **end if**
9: $\quad$ **end for**
10: **until** $O_{old} := O_{new}$
11: return O_{new} as the greedy-optimum

3.4. Parallel genetic algorithm

Since our genetic algorithm spends most of the time in the evaluation step, we parallelize the evaluation step. Our parallel genetic algorithm works as follows: If there are N available processors in a system, we select one master processor and let it process the selection, crossover, and mutation steps. Thereafter, it spreads all the individuals that must be evaluated to other processors, including itself. After each processor finishes calculating the evaluation, the master gathers the results and performs the next iteration based on them.

4. The cost model

In this section, we present a cost model that evaluates the retrieval and storage costs of an index configuration. 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 configuration uses the structure of the class-hierarchy index for each member index, which is a variation of the B^+-tree structure where class identifiers are stored with index entries in the leaf nodes. Therefore, we have developed our cost model based on the discussion in Ref. [14].

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 or all larger than the size of an index page.

4.1. Parameters

4.1.1. Database parameters

- D_{c_j}—number of distinct index key values in class C_j
- D_i—number of distinct index key 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 key value of an index i

$$NC_i = \sum_{\text{for all classes } C_i \text{ in index } i} \frac{D_{C_j}}{D_i}.$$

4.1.2. 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

$$XL_i = header_length + kl + (sizeof(\text{CLASSID}) + sizeof(\text{offset}) + sizeof(\text{number_of_OIDs})) \times NC_i + sizeof(\text{OID}) \times K_i,$$

  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

$$\text{if } XL_i \le P \quad LP_i = \lceil (D_i \times XL_i)/P \rceil$$
$$\text{if } XL_i > P \quad LP_i = D_i$$
$$LP_i + OP_i = D_i \times \lceil XL_i/P \rceil$$

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

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

4.2. Storage cost model

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

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

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

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

4.3. Retrieval cost model

4.3.1. 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 the index i is

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

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

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

4.3.2. 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 the index i as follows:

$$RC_i^{range} = \begin{cases} H_i + \lceil query_range \times LP_i \rceil & \text{if } XL_i \le P \\ H_i + \lceil query_range \times (LP_i + OP_i) \rceil & \text{if } XL_i > P \end{cases}$$

The total retrieval cost for query q is given by

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

4.4. Average retrieval cost ratio

The performance of an index configuration is very dependent on the target classes. Therefore, we need to introduce a single metric for performance comparison between index configurations. For this consideration, we define the average retrieval cost ratio (ARCR), which is the ratio of the average retrieval cost provided by an index configuration to that of the full index configuration (without considering the storage cost of the latter). The full index configuration here is a set of indexes on the full extent of each class and therefore, it is the upper bound of the performance of PINS.

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 the cost evaluation are the ratio of the single-key queries and the query frequency on each class.

Table 1
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

The retrieval cost ratio RC_i of each class C_i is given by

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

where $I^{full}_{C_i}$ is the index on the full extent of a class C_i.

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

$$\text{ARCR} = \sum_{\forall C_i} R_{C_i} \times \text{query frequency on } C_i.$$

4.5. 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 distribution of key values, the number of instances, the single key query ratio, and the query range. Table 1 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 seen in Table 1, the error ratios between the experimental and the expected results are always within 5%. The error ratio increases somewhat in the 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.

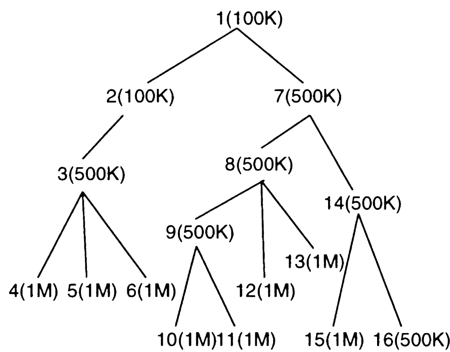

Fig. 6. Test class hierarchy H16.

5. Performance analysis

We evaluated the performance of PINS mainly on the class hierarchy depicted in Fig. 6. This class hierarchy (Hierarchy H16) consists of 16 classes. The number in each parenthesis denotes the number of instances of the class. In addition, we used class hierarchies that have eight classes: 8 (H8), 24 (H24), 40 (H42), 48 (H48), 56 (H56), and 60 (H60). We designed the class hierarchy roughly so that leaf classes and classes close to leaves have more instances than the classes close to the root, and classes in the same depth have the same number of instances. We examined the retrieval cost and storage cost while varying the number of indexes. We also compared performances of PINS with CH. Finally, we tested time-saving by parallelizing the genetic algorithm.

For each indexing configuration, we computed the retrieval cost ratio for a query on each class, ARCR, and the total storage cost ratio against the full index configuration. Table 2 lists the parameter settings used in the experiments. In all cases, the single key query ratio is 10% and queries are distributed uniformly over all classes. The distribution of key values was totally inclusive in the range of 1–10,000.

Table 2
Parameters used in the experiment

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

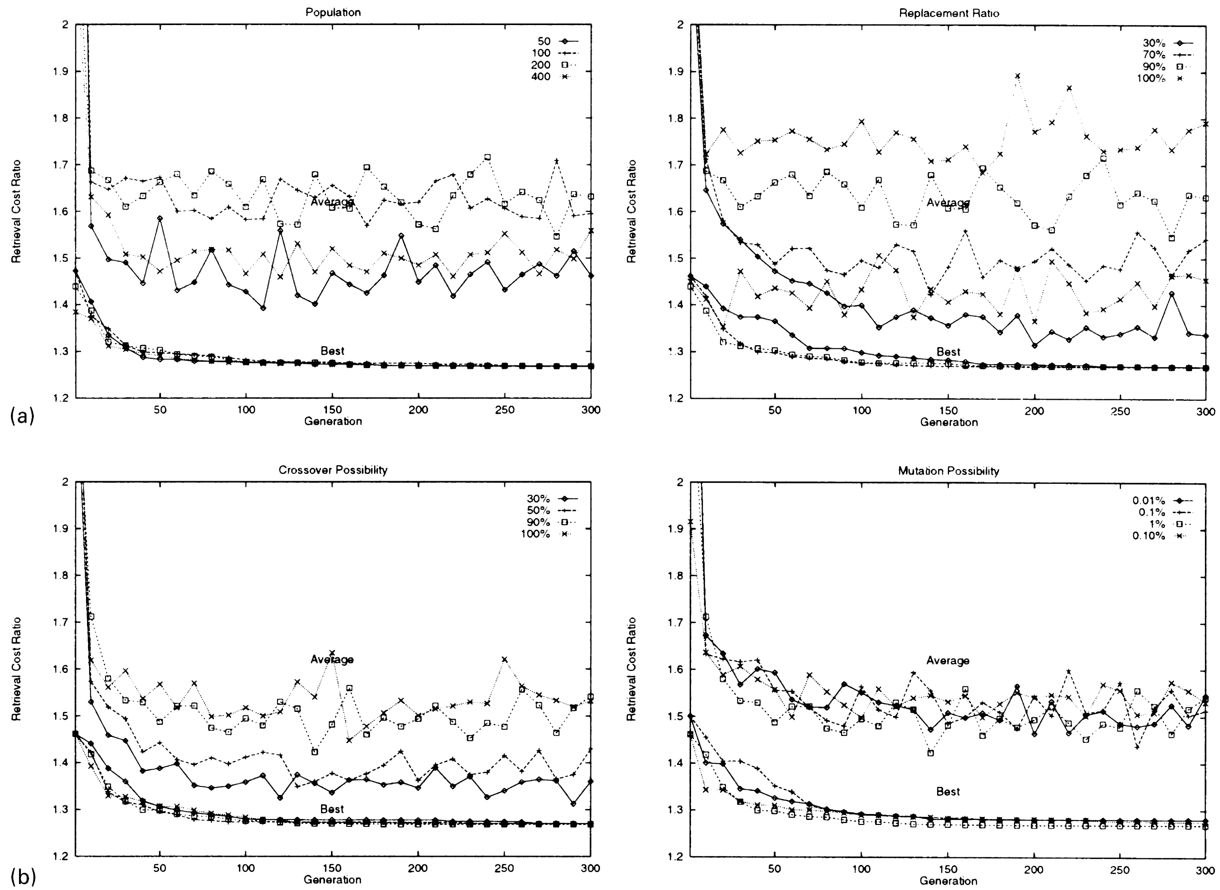

Fig. 7. Fitness changes of average and best individual with varying parameters: (a) size of population and replacement ratio; (b) crossover ratio and mutation ratio.

Table 3
PINS with varying number of indexes to be used

i_{max}	$\|I_{PINS}\|$	I_{PINS}	ARCR	SC
1	1	{{1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16}}	7.13	16,513
2	2	{{1 7 8 9 10 11 12 13} {2 3 4 5 6 14 15 16}}	3.74	16,568
4	4	{{1 7 8 12 13} {2 3 4 5 6} {9 10 11} {14 15 16}}	2.04	16,675
6	6	{{1 2 3 4} {5 6} {7 14 15} {8 12 13} {9 10 11} {16}}	1.60	16,785
8	8	{{1 7 8 13} {2 3 5} {4 6} {9 11} {10} {12} {14 15} {16}}	1.37	16,893
10	10	{{1 7 8 9} {2 3 4} {5} {6} {10} {11} {12} {13} {14 15} {16}}	1.26	17,001
12	12	{{1 2 3} {4} {5} {6} {7 14} {8 9} {10} {11} {12} {13} {15} {16}}	1.23	17,109
14	12	{{1 2 3} {4} {5} {6} {7 14} {8 9} {10} {11} {12} {13} {15} {16}}	1.23	17,109
16	12	{{1 2 3} {4} {5} {6} {7 14} {8 9} {10} {11} {12} {13} {15} {16}}	1.23	17,109

We performed our experiment on an IBM SP2 parallel processing system.[3]

As a preparation for the main experiments, we performed the following experiments to find the adequate population size, the replacement ratio, the crossover ratio, and the mutation ratio. We determined these parameters by varying one parameter while keeping the other three parameters fixed.[4] Fig. 7 shows the best and ARCR of each generation (iteration) when the parameters are varied. We used a population size of 100 and a replacement ratio of 70%, because with these values the experiments do not take a long time, and the best-of-generation does not converge prematurely. We selected 0.9 for crossover probability and 0.01 for mutation probability because in most experiments these values produced the best results, and did not introduce premature convergence. All experimental results are performed with these parameter settings.

We also tested several different schemes of creating initial population and selected the scheme that creates the

[3] The SP2 system has 40 nodes and each node is equipped with 266 MFLOPS CPU and 128 MB RAM so that the system as a whole provides a peak performance of 10 GFLOPS and 4.8 GB of memory.

[4] Genetic algorithms can be used to determine these parameters.

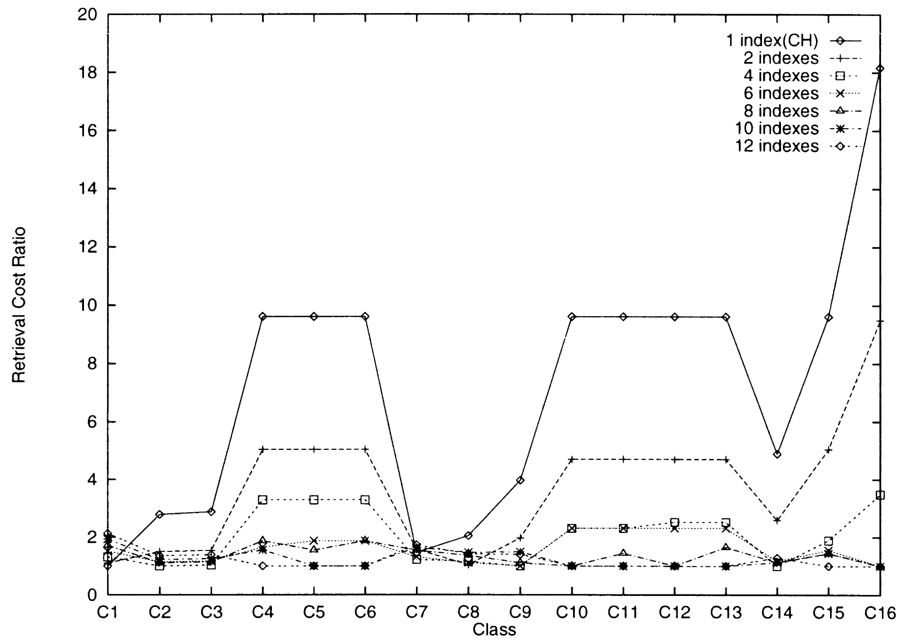

Fig. 8. Expected retrieval of classes.

initial population at random because it produced the best result with the minimum time.

Table 3 illustrates experimental results for H16 as we change the number of indexes to be used. In this table, the first column denotes the maximum number of indexes to be used (i_{max}), and the other columns show the number of indexes produced ($|I_{PINS}|$), partition index configurations (I_{PINS}), average retrieval cost ratio with full indexes (RC), and storage cost in pages. Because an index is created for each partition, the number of indexes can also be considered as the number of partitions. Table 3 shows that no performance gain is achieved using more than 12 indexes ($i_{max} > 12$). Therefore, it is optimal to use 12 indexes for H16. Table 3 also shows that as the number of indexes is increased, the retrieval cost decreases significantly, whereas the storage cost increases very slowly. This means that splitting a large index into several small indexes does not require a significant space overhead. Comparing PINS with $i_{max} = 1$ (CH) and PINS with $i_{max} = 12$, we find that there is around 700% performance enhancement, whereas there is only 4% additional storage cost. So, we can safely use $i_{max} = N$ for a class hierarchy with N classes, since, in most cases, it can cover the PINS result with smaller maximum indexes, and the number of indexes does not have a great influence on the storage cost.

Fig. 8 illustrates the retrieval cost of every class in H16. PINS with $i_{max} = 1$ is drawn in a solid line, and it can be regarded as CH. CH gives good performance for classes near the root class, but when it comes to classes close to leaves and leaf classes, the performance degrades significantly. In particular, the retrieval cost of C_{16} is higher than other leaves, such as C_{11}, C_{12} and C_{15}, because it has fewer instances than others. As the number of indexes used is increased, the performance of the classes close to leaves improves greatly, whereas the performance of those around the root class degrades. This is because we assigned the same priority (or access ratio) to every class so that PINS tried to provide equal performance for the classes.

Table 4 shows the optimal number of partition index configurations, the storage cost ratio with full index (SC_{PINS}/SC_{full}), ARCR with full index ($ARCR_{PINS}/ARCR_{full}$), the storage cost ratio with class-hierarchy index ($SC_{PINS}/$

Table 4
PINS result with test hierarchies

Hierarchy	$\|I_{PINS}\|$	SC_{PINS}/SC_{full}	$ARCR_{PINS}/ARCR_{full}$	SC_{PINS}/SC_{CH}	$ARCR_{PINS}/ARCR_{CH}$
H8	5	0.37	1.19	1.2233	0.40
H16	12	0.25	1.23	1.2321	0.17
H24	19	0.38	1.31	1.2697	0.11
H32	22	0.21	1.49	1.2685	0.10
H40	27	0.43	1.75	1.3167	0.10
H48	33	0.40	1.77	1.3570	0.08
H56	37	0.43	1.74	1.3156	0.07
H60	40	0.43	1.75	1.3332	0.07

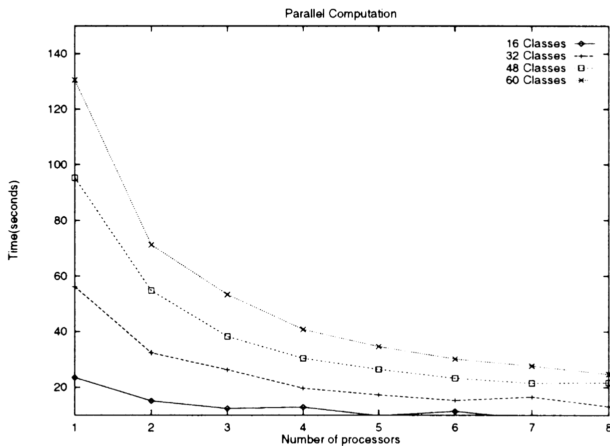

Fig. 9. Elapsed time in the genetic algorithm with varying the number of CPUs.

SC_{CH}), and ARCR with class-hierarchy index ($ARCR_{PINS}/ARCR_{CH}$) with various class hierarchies. In most cases, PINS entails retrieval cost of up to 130% of the full index, while consuming less than half storage space. Considering that we used hierarchies with only three or four levels depth, the storage benefit would be greater than this result if we were to test hierarchies with greater depths. Compared with CH, PINS consumes somewhat larger storage space (119–177%) but provides far better retrieval performance by reducing the retrieval cost up to 7–40% of CH. In particular, PINS gives better performance improvement as the number of classes increases.

Besides the test results shown, we also tested many different cases of workload (query frequencies among classes). We found that PINS selects CH when most of the queries are centered at root class, while it produces a set of single class indexes for the reverse case. This result shows that PINS can adaptively design index configurations according to the workload, while CH and single class index are fixed regardless of the workload.

Fig. 9 illustrates the execution time in relation to the number of processors being used. In this figure, we include only the time for running the genetic algorithm, excluding the time for the data loading and program initialization, because the genetic algorithm takes most of the program execution time. In our parallel genetic algorithm, we parallelized the evaluation step. If there are N individuals that have to be evaluated and there are k CPUs available, the master CPU allocates N/k individuals to each CPU including itself. After the evaluation is completed in each CPU, the result is sent back to the master CPU. The master CPU then continues with the next step. As expected, the execution time decreases as the number of CPUs increases. Parallelism is more noticeable when it comes to large hierarchies with relatively more classes (H48, H60). In most cases, PINS shows good parallelism until a certain number of CPUs are used (four CPUs with H16, and seven CPUs with H32). This is due to the fact that only the evaluation step is parallelized, while selection, crossover and mutation are processed sequentially by the master processor. Therefore, although the evaluation step takes less time as the number of processors increases, the total execution time for the genetic algorithm does not decrease significantly.

6. Conclusion and future work

The performance of the object databases is the key to their commercial success, and thus efficient indexing on a class hierarchy is essential for them.

In this study, we present the partition index configuration scheme (PINS), which finds a near optimal index configuration, within a specified number of indexes, using the genetic algorithm and the greedy algorithm. PINS 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 frequency on each class. Essentially, our PINS 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, PINS showed a significant performance enhancement compared to the class-hierarchy index, with little additional space overhead. Although PINS does not give as good a performance as the full index (FI), it is more practical because it requires far less storage space than FI, and does not introduce additional update costs.

Since PINS assume the uniformity in the distribution of key values, we are currently devising a statistical method to estimate it more precisely, and will extend our method for parallel database systems.

Acknowledgements

We wish to thank the referees for their valuable comments and suggestions.

References

[1] E. Bertino, P. Foscoli, Index organizations for object-oriented database systems, IEEE Transactions on Knowledge and Database Engineering 7 (2) (1995) 193–209.

[2] E. Bertino, W. Kim, Indexing techniques for queries on nested objects, IEEE Transactions on Knowledge and Database Engineering 1 (2) (1989) 196–214.

[3] T.N. Bui, B.R. Moon, A fast and stable hybrid genetic algorithm for the ratio-cut partitioning problem on hypergraphs, in: Proceedings of the Design Automation Conference, 1994, pp. 664–669.

[4] T.N. Bui, B.R. Moon, Genetic algorithm and graph partitioning, IEEE Transactions on Computers 45 (7) (1996) 841–855.

[5] I.T. Christou, R.R. Meyer, Fast distributed genetic algorithms for partitioning uniform grids, in: Proceedings of the Workshop on Parallel Algorithms for Irregularly Structured Problems, 1996, pp. 89–103.

[6] D. Commer, The ubiquitous b-tree, ACM Computing Surveys 11 (2) (1979) 121–137.

[7] I.W. Flockhart, N.J. Radcliffe, A genetic algorithm-based approach to

data mining, in: Proceedings of Knowledge Discovery and Data Mining, 1996, pp. 299–302.
[8] D.E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, Reading, MA, 1989.
[9] Postgres Global Development Group, PostgreSQL Programmer's Guide, 1999.
[10] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 1984, pp. 47–57.
[11] J.T. Horng, B.J. Liu, A genetic algorithm for database query optimization, in: Proceedings of the First IEEE Conference on Evolutionary Computation, 1994, pp. 350–355.
[12] Y. Ishikawa, H. Kitagawa, N. Ohbo, Evaluation of signature files as set access facilities in oodbs, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 1993, pp. 247–256.
[13] C. Kilger, G. Moerkotte, Indexing multiple sets, in: Proceedings of the International Conference on Very Large Data Bases, 1994, pp. 180–191.
[14] W. Kim, K.-C. Kim, A. Dale, Object-oriented Concepts, Databases, and Applications, Addison-Wesley, Reading, MA, 1989 (Chapter: Indexing Techniques for Object-oriented Databases).
[15] 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, 1992, pp. 134–143.
[16] S. Ramaswamy, P.C. Kanellakis, Oodb indexing by class-division, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 1995, pp. 139–150.
[17] 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, pp. 203–213.
[18] M. Srinivas, L.M. Patnaik, Genetic algorithms: a survey, IEEE Computer 27 (6) (1994) 17–26.
[19] P. van Bommel, Th.P. van der Weide, Towards database optimization by evolution, in: Proceedings of the International Conference on Information Systems and Management of Data (CISMOD), 1992, pp. 273–287.