On Index Selection Schemes for Nested Object Hierarchies

Sudarshan S. Chawathe*, Ming-Syan Chen and Philip S. Yu

Computer Science Department*
Stanford University
Stanford, CA 94305

IBM Thomas J. Watson Research Center
P.O.Box 704
Yorktown Heights, NY 10598

Abstract

In this paper we address the problem of devising a set of indexes for a nested object hierarchy in an objected-oriented database to improve the overall system performance. Note that the effects of two indexes could be entangled in that the inclusion of one index might affect the benefit achievable by the other index. Such a phenomenon is termed index interaction in this paper. Clearly, the effect of index interaction needs to be taken into consideration when a set of indexes is being built. The index selection problem is first formulated and four index selection algorithms are evaluated via simulation. The effects of different objective functions, which guide the search in the index selection algorithms, are also investigated. It is shown by simulation results that the greedy algorithm which is devised in light of the phenomenon of index interaction performs fairly well in most cases. It in fact agrees with the very nature of index interaction we identify in this study. Sensitivity analysis for various database parameters is conducted. We not only conduct an extensive performance study for index selection algorithms, but also explore the effect of index interaction to deal with this global optimization problem.

Index Terms: Object-oriented databases, indexing, nested object hierarchy, index interaction.

1 Introduction

Due to the increasing demand for sophisticated data-modelling capabilities by many database applications, object-oriented databases (OODB's) have recently attracted a significant amount of attention in academic and industrial communities [5] [12] [17]. Among others, OODB's have two major advantages. First, as opposed to the query-based (typically SQL) approach used by relational databases, an OODB renders efficient access to pointer-based data structures by permitting direct manipulation of data via program control. Second, an OODB supports declarative data access [20], which not only offers ease of programming but also allows the database system to improve the query processing for faster query execution.

Unlike the query optimization in a relational database which has well-developed theoretical results, query processing in an OODB is still in its infancy [11]. The problem is complicated due to the lack of universally-accepted data models and query languages [18]. In a "flat" (or first normal form, in relational database terminology) data model, an attribute can only be a primitive data type. However, in object-oriented data models the value of an attribute of one object may be a set of values or another object. This nesting of objects through attributes leads to the *nested object hierarchy* [8], also known as the *class-attribute hierarchy* [3]. An example of a nested object hierarchy is extracted from [3] and shown in Figure 1, where an attribute of any class can be viewed as a *nested attribute* of the root class. Note that the nested object hierarchy is intrinsically different from the class hierarchy which is given in Figure 2 for an illustrative purpose. In such an OODB environment, how to utilize the pointer-based data structures to devise proper indexing schemes and retrieve objects efficiently has been identified as a very important issue to further improve the system performance [12] [13] [15].

Several indexing schemes have been proposed for nested attribute queries [1] [2] [3] [8] [10] [16]. Three index organizations for use in the evaluation of a query in an OODB are introduced in [3]. As an extension to [3], performance of path indexes for queries containing several predicates is evaluated in [2]. In [10] query processing in an OODB system is improved by maintaining separate structures to redundantly store objects which are frequently traversed by database queries. A hybrid indexing technique, called a generalized index, is proposed in [8] to support class hierarchy with complex and primitive objects. Indexing in GemStone OODB is described in [16]. In [6], an optimal index configuration for a path is achieved by splitting the path into subpaths and optimally indexing each subpath. It is noted that most of these prior works only considered indexing along a single path in a nested object hierarchy. However, as will be shown later, the effects of two indexes could be entangled. Specifically, the inclusion of one index could affect the benefit achievable by the other index. Such a phenomenon is termed index interaction in this paper. A detailed example for index interaction is given in Section 3.3. Clearly, the effect of index interaction needs to be taken

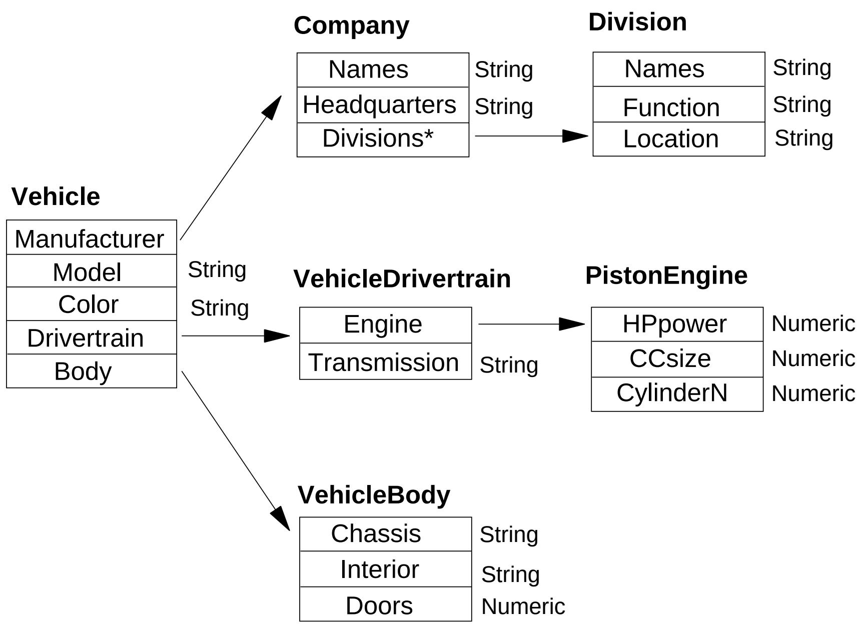

Figure 1: An example for nested object hierarchy.

into consideration when a set of indexes is being built[1]. Note that while affecting the indexing in a single query path, the index interaction phenomenon has a larger performance impact when the global effect of indexing multiple query paths is considered. However, whereas building an index along a single path has been extensively studied, the problem of building a set of indexes for a group of queries while considering index interaction, despite its importance, has not been fully explored. This is mainly due to the inherent difficulty of this problem, since when many indexes are evaluated as a whole, the interaction among indexes significantly complicates the method to evaluate their costs and benefits globally. Note that as the granularity of data objects in an OODB becomes finer and the database schema tends to be more sophisticated nowadays, it has become increasingly important to explore the effect of building a set of indexes.

Consequently, we address in this paper the problem of devising a set of indexes for a nested object hierarchy with its given profile[2] so as to improve the overall performance for executing a group of queries. Performance is measured using the metrics of retrieval, update and storage costs. Specifically, we shall focus on a common type of query, called the *nested attribute query*: Select all objects of a certain class that have a *nested* attribute equal to a given value. An example of such a path query is given in Figure 3. The index selection problem is first formulated and some

[1]The occurrence of index interaction will be justified in Section 4 by a performance study based on the 007 benchmark [4] and also by a theoretical characterization.

[2]The profile of a nested object hierarchy includes the cardinality of each class, the selectivity of each attribute, a certain amount of storage available for indexing, and some other information on access patterns.

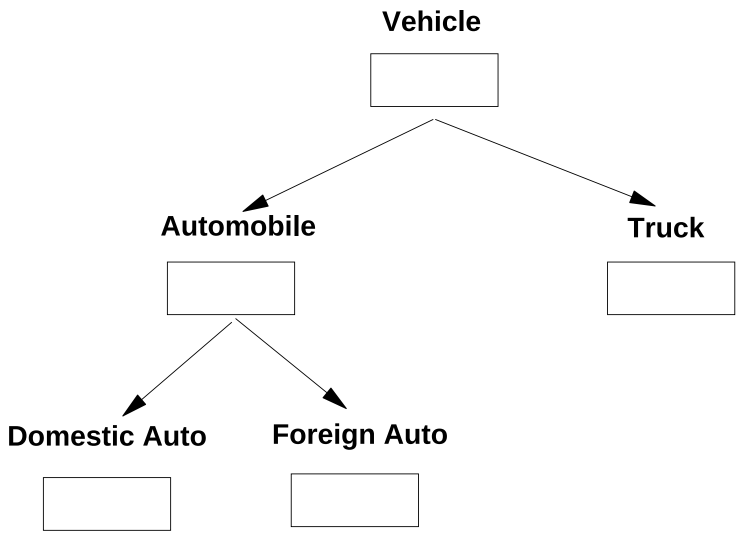

Figure 2: An example for class hierarchy.

important parameters are identified. Then, four index selection algorithms for queries in a nested object hierarchy are presented, i.e., a naive scheme, an algorithm based on profit ordering, a greedy algorithm, and then a more sophisticated look-ahead one. The naive scheme essentially corresponds to a random inclusion of indexes, which is used for a comparison purpose. The algorithm on profit ordering sorts the profits of individual indexes in descending order first, and then includes as many indexes as possible according to the sorted index list, subject to the storage constraint. The greedy algorithm is similar to the one on profit ordering in that it also includes as many indexes as possible based on a sorted index list, but different from the latter in that the sorted index list used by the greedy algorithm is revised after every inclusion of an index, thus taking index interaction into consideration. The look-ahead algorithm goes beyond the greedy algorithm by looking ahead to evaluate the combined benefit of several indexes before adding one into the index list. In addition, three objective functions, which guide the search in the index selection algorithms, are also proposed. A detailed description of index selection algorithms and objective functions can be found in Section 3.

To conduct the performance study, an OODB system simulator is coded in C++ to model the detail of data retrievals under different indexed environments. The four index selection algorithms and the three objective functions are comparatively evaluated. To conduct a sensitivity analysis for various parameters, different values for the storage constraint for indexing, update and storage costs, and attribute selectivity are employed in the simulation and their effects are evaluated. In

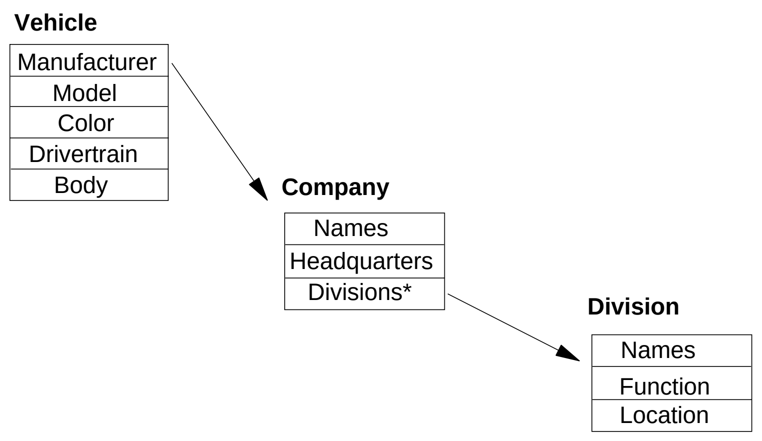

Figure 3: An example path query: **select** vehicle **where** manufacturer division location = "city name."

addition, some theoretical results on index interaction are derived to provide more insights into this index selection problem. It is shown by simulation results that despite their maintenance cost, indexes provide a net benefit over a wide range of database parameters. It is observed that the greedy algorithm devised performs fairly well in most cases, which in fact agrees with the very nature of index interaction we identify in this study. The contributions of this paper are twofold. We not only conduct an extensive performance study for index selection algorithms, but also explore the effect of index interaction to deal with this global optimization problem. To the best of our knowledge, no prior work has either explored the theoretical aspects of, or evaluated algorithms for building a set of indexes. This feature distinguishes our work from others.

This paper is organized as follows. Notation, cost model and assumptions are given in Section 2. Index selection algorithms and objective functions are described in Section 3. Performance study is conducted in Section 4. This paper concludes with Section 5.

2 Notation, Cost Model and Assumptions

As pointed out in [3], an important element common to an OODB is the view that the value of an attribute of an object can be an object or a set of objects. A class $C(1)$ consists of a number of attributes, and the value of an attribute A of an object belonging to class $C(1)$ can be an object or a set of objects belonging to another class $C(2)$. The class $C(2)$ is called the domain of attribute A of class $C(1)$. Certainly, $C(2)$ may in turn consist of a number of attributes whose domains are other classes. A *path* in the nested object hierarchy is represented as $C(1).A(1).A(2)\ldots.A(n)$, where $C(1)$ is the class whose objects will be retrieved based on the nested attribute lookup. $A(1)$ is an attribute of $C(1)$ and $A(i)$ is an attribute of the class associated with $C(1).A(1).A(2)\ldots.A(i-1)$, for $i = 2\ldots n$. We denote the length of the path by n. For example, in the path "vehicle.manufacturer.division.location," $C(1)$ is "vehicle," $A(1)$ is "manufacturer," $A(2)$ is "division," and $A(3)$ is "location." A nested index on the path vehicle.manufacturer.division.location will associate a distinct value of the location attribute, say "Ann Arbor", with a list of object identifiers of vehicles, each of which has its manufacturer that is an instance of the company class whose division's location is "Ann Arbor."

The OODB system considered in this study has read-only queries and also retrievals/updates using program controlled traversals. Note that queries can be quite complex, involving many attributes and Boolean combinations of lookup conditions. As mentioned earlier, we focus on a common type of query, called the *nested attribute query*: "Retrieve all objects of class $C(1)$ such that $C(1).A(1).A(2).\ldots.A(n) = v$," where v is a given value of interest. This has been referred to as the *implicit join* operation in the literature [2]. Note that despite their simplicity, such queries form building blocks for more complex queries, and it is thus very important to implement them efficiently. Updates by traversals are modeled by considering the update costs for those attributes on the path indexes being maintained. Insertions and deletions are modeled similarly. For example, with an index from Division to Vehicle in Figure 3 the following query can be answered efficiently.

Q1: **select** vehicle **where** manufacturer division location = "Ann Arbor"

For better readability of this paper, the formulas for the retrieval, update and storage costs of an index are given in the Appendix, which are basically the same as those for *nested indexes* on a B-tree implementation described in [3], except some modifications. Readers interested in the derivation of these formulas are referred to [3]. The difference between the formulas in [3] and those in this study lies in the estimation of the average number of instances of class $C(1)$ that have the same value for the nested attribute $A(n)$. (Such a number is denoted by $k(1,n)$.) Note that it is assumed in [3] that there are no partial instantiations of $C(1)$, and the formula of $k(1,n)$ is thus simplified in [3][3]. Such an assumption is not made in this paper. As a result, we employ the original formula for $k(1,n)$ without resorting to any simplification. Such an assumption relaxation in fact allows us to take into account the object reference topologies of different database populations in our simulation study, thus leading to more general results. The database and system parameters used in the cost

[3]The formula $k(1,n) = \prod_{i=1}^{n} k(i)$, for the average number of instances of class $C(1)$ that have the same value for the nested attribute $A(n)$, is simplified to $|C(1)|/|A(n)|$ in [3] under the assumption that there are no partial instantiations of $C(1)$. Note that $k(i)$ is the average number of instances of class $C(i)$ that have the same value for the nested attribute $A(i)$.

formulas for the indexes are summarized in Table 7 of Section 4 where the performance study is conducted.

Same as other related studies [3], some assumptions are made to facilitate our discussion. First, all attributes are bidirectional. Explicitly, for each attribute link from class $C(i)$ to class $C(j)$, there is a *reverse reference* from $C(j)$ to $C(i)$. Also, all key values have the same length, which in turn means that all nonleaf index records have the same length in all indexes. The values of an attribute are uniformly distributed among the objects of the class which defines that attribute [7]. In addition, each attribute is equally likely to be updated and all attributes have the same selectivity. Note that these assumptions are mainly made to ease our implementation as well as to simplify our discussion, and are believed not affect the relative merits of the index selection methods we shall evaluate in this paper.

3 Index Selection Schemes

In this section we describe the objective functions and the index selection algorithms that we shall evaluate. The index selection problem can be viewed as a search problem where the search space consists of all possible subsets of indexes. All the indexing schemes select indexes to optimize the objective function employed, subject to the constraint that the indexes included cannot consume more than a specified amount of storage. Three objective functions will be presented in Section 3.1, four index selection algorithms are described in Section 3.2, and illustrative examples are given in Section 3.3.

3.1 Objective Functions

We shall evaluate three different objective functions which guide the search for candidate indexes. The first objective function is based on profit, the second is based on return ratio, and the third is a combined version of the first two. These objective functions are applied to individual indexes to decide which index should be included into the set of selected indexes.

- The objective function on "profit," denoted by $P(\cdot)$, is based on the difference between the corresponding reduction in the retrieval cost provided by the index and the associated increase in the update cost. In other words, $P(I)$ corresponds to the reduction in the global dynamic cost due to the inclusion of index I, where the dynamic cost means the sum of the retrieval cost of database queries and the update cost for indexes in response to database updates. Note that because of the phenomenon of index interaction this value varies as the selected index set changes. Specifically, we have,

$$P(I) = \text{retreival\_benefit}(I) - \text{update\_cost}(I).$$

- The objective function on "return ratio," denoted by $R(\cdot)$, is based on the ratio of $P(\cdot)$ to the storage cost of the index. It can be seen that by taking into account the amount of storage required by an index, this function will prefer small indexes than large ones.

$$R(I) = \frac{P(I)}{\text{storage\_cost}(I)}.$$

- In order not to penalize large indexes unnecessarily when there is a lot of storage available, a mixed objective function $M(\cdot)$, which according to the amount of storage available, adaptively selects its formula to evaluate indexes, is also employed in our study. $M(\cdot)$ is formulated as below.

$$M(I) = \begin{cases} P(I) & \text{if } \frac{\text{remaining\_storage}}{\text{original\_available\_storage}} > \alpha, \\ R(I) & \text{otherwise,} \end{cases}$$

  where $0 < \alpha \leq 1$ denotes a threshold for the ratio of the remaining storage to the original available storage. $M(I)$ is initially the same as $P(I)$. However, when such a ratio on the remaining storage is less than α, meaning that there is no large amount of storage available, $M(I)$ will be used, instead of $P(I)$, as the objective function for index selection such that storage can henceforth be used more prudently.

3.2 Four Index Selection Algorithms

3.2.1 Naive algorithm (NV)

As mentioned earlier, the naive algorithm (NV) is used for a comparison purpose. NV tries to include as many indexes with positive profits as possible, until the amount of available storage is exhausted.

3.2.2 Algorithm on profit ordering (PO)

Clearly, indexes included could be more profitable than those selected by NV if some provisions are made during the index selection. The algorithm on profit ordering (PO) will first statically evaluate the objective function values for all the indexes and sort indexes in descending order of these values. PO then selects from the sorted index list as many indexes as allowed by the available storage in a top-down manner.

3.2.3 Greedy algorithm (GD)

This greedy algorithm (GD) used is essentially a greedy search applied to the index-subset search space. GD also sorts indexes according to their objective function values first. Then, at each step, GD adds the most profitable index to the current set of selected indexes, and revises the objective function values for all the remaining indexes, thereby taking the index interaction into account. GD can be outlined below, where S is the set of selected indexes and A is the set of remaining indexes.

```
Algorithm GD: Greedy index selection
Input: Set $A$ of all indexes and the objective function $F$.
Output: Set of indexes to be built.
$S := \emptyset$;
repeat {
    Evaluate the objective function values for all indexes in $A - S$;
    Let I be the index with the maximal objective function value;
    if $F(I) \leq 0$ return $S$;
    $S = S \cup \{I\}$;
    $A = A - \{I\}$;
}
```

3.2.4 Lookahead algorithm (LH)

Note that GD may choose a locally optimal solution and overlook those that are globally better. To remedy this, lookahead schemes, which explore more search space before making a decision on determining which index to be included into the selected set, are employed. Basically, by considering the effect of adding more than one index to the current set of indexes, the index chosen for inclusion can be thought of as the one that could lead to a better solution a few steps later. Based on this concept of looking ahead, we can obtain a family of search algorithms, denoted by LH(m, n), where m and n are two parameters associated with the search complexity. Let S be the current set of selected indexes. In each step, LH(m, n) considers the effect of adding to S an index subset which has a cardinality less than or equal to n and is made up of the m best indexes. After the effects of all such index subsets are evaluated, the most beneficial index subset is identified. Then, within this most beneficial index subset, the most beneficial index is added into S. Suppose we have four indexes to be considered, and (i_2, i_1, i_4, i_3) denotes the descending order of the objective function values of these four indexes. LH(3,2) will consider the benefits of the six index sets: $\{i_2\}$, $\{i_1\}$, $\{i_4\}$, $\{i_2, i_1\}$, $\{i_2, i_4\}$, and $\{i_1, i_4\}$. Suppose $\{i_2, i_1\}$ is the most beneficial one among them. Then the more beneficial one of i_1 and i_2 will be included into S. Formally, LH can be described as follows, where the objective function F could be P (on profit), R (on return ratio), or M (mixed) described in Section 3.1.

```
**Algorithm LH:** Lookahead(m,n) index selection
**Input:** Set $A$ of all indexes and the objective function $F$.
**Output:** Set of indexes to be built.
$S := \emptyset$;
repeat {
    Evaluate the objective function values for all indexes in $A - S$;
    Sort indexes in $A - S$ according to the descending order of their objective function values;
    Let L be the first $m$ indexes in the sorted index list;
    Identify all subsets with cardinalities no greater than $n$ from L. Let $T$ be the set of all such subsets.
    Let $IS$ be the subset with the maximal objective function value among $T$;
    if $F(IS) \leq 0$ return $S$;
    Let $I$ be the index with the maximal objective function value among those in $IS$.
    $S = S \cup \{I\}$;
    $A = A - \{I\}$;
}
```

3.3 Illustrative Examples

To illustrate the algorithms we described thus far, consider the schema graph in Figure 4 where queries q_1 and q_2 have occurrence frequencies 0.8 and 0.2, respectively. Suppose that the four indexes (i_1, i_2, i_3, i_4) shown in Figure 4 are the four most beneficial ones to consider for these queries. Let the maximal storage available for indexing be 12. Also, assume that the update cost, storage cost and retrieval benefit of each index are those given in Table 1. Note that the retrieval benefit is the reduction in the retrieval cost for a query utilizing that index. It can be seen that query q_2 benefits from neither i_1 nor i_2. Based on this profile, we shall show the operations of NV, PO, GD and LH(4,2), assuming the objective function on profit is employed.

Algorithm NV: First, consider algorithm NV. NV tries to include as many of these indexes as possible. Since the storage for indexing is limited to 12, it follows from Table 1 that the final selection by NV is $\{i_1, i_2, i_3\}$. It can then be seen that q_1 saves three units of retrieval cost by using i_1 (i.e., from 7 steps that are required without using indexes to 4 steps by using i_1), and q_2 saves

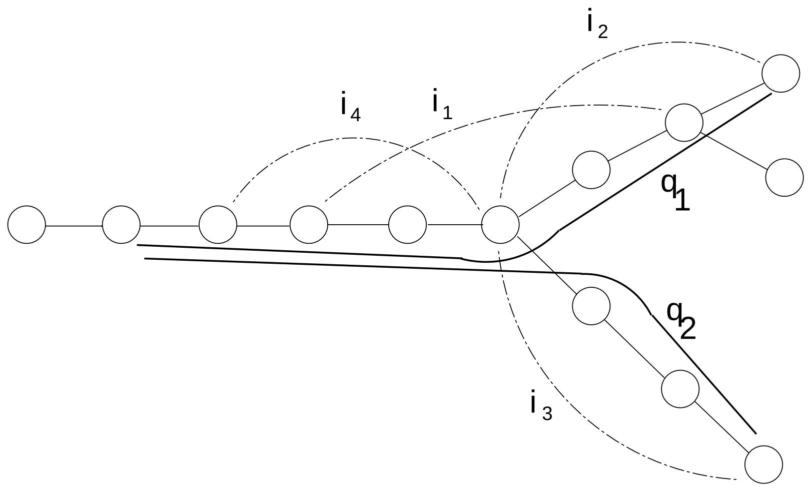

Figure 4: Schema graph with queries and indexes.

Index	Update Cost	Storage	Retrieval Benefit
i_1	0.25	6	3
i_2	0.2	4	2
i_3	0.1	2	2
i_4	0.2	5	2

Table 1: The costs, storage overheads and retrieval benefits for the four example indexes.

Index	Retrieval Benefit	Objective Function
i_1	$0.8 \times 3 + 0.2 \times 0 = 2.4$	$2.4 - 0.25 = 2.15$
i_2	$0.8 \times 2 + 0.2 \times 0 = 1.6$	$1.6 - 0.2 = 1.4$
i_3	$0.8 \times 0 + 0.2 \times 2 = 0.4$	$0.4 - 0.3 = 0.1$
i_4	$0.8 \times 2 + 0.2 \times 2 = 2$	$2 - 0.2 = 1.8$

Table 2: The original objective function value for each index.

Index	Retrieval Benefit	Objective Function
i_2	$0.8 \times 0 + 0.2 \times 0 = 0$	$0 - 0.2 = -0.2$
i_3	$0.8 \times 0 + 0.2 \times 2 = 0.4$	$0.4 - 0.1 = 0.3$
i_4	$0.8 \times 0 + 0.2 \times 2 = 0.4$	$0.4 - 0.2 = 0.2$

Table 3: The objective function values for indexes after i_1 is selected by GD.

two units of retrieval cost by using i_3. The reduction in retrieval cost of the queries provided by this index set is $0.8 \times 3 + 0.2 \times 2 = 2.8$. Note that i_2 is redundant since q_1 can be evaluated with a lower cost by i_1, and that i_1 and i_2 cannot be used together to evaluate q_1. Also, q_2 only benefits from i_3. The update cost of the selected index set is $0.25 + 0.2 + 0.1 = 0.55$. The overall reduction in the cost resulted from using $\{i_1, i_2, i_3\}$ is thus $2.8 - 0.55 = 2.25$.

Algorithm PO: Next, consider the index selection algorithm based on profit ordering. Table 2 indicates the retrieval benefit and the objective function value (i.e., retrieval benefit – update cost) for the four indexes prior to any index selection.

By sorting the indexes according to the descending order of their objective function values, PO obtains (i_1, i_4, i_2, i_3). It then selects as many indexes as possible from the top of the sorted list. Subject to the available storage 12, it can be verified that the final selection by PO is $\{i_1, i_4\}$. The reduction in retrieval cost of the queries provided by this index set is $0.8 \times 3 + 0.2 \times 2 = 2.8$. The update cost of the selected index set is $0.25 + 0.2 = 0.45$, and the overall reduction in the cost resulted from using $\{i_1, i_4\}$ is thus $2.8 - 0.45 = 2.35$.

Algorithm GD: In the first iteration of the greedy algorithm the retrieval benefits and the objective function values for these indexes are the same as those shown in Table 1. Index i_1 has the maximal value for the objective function and is thus added to the set of selected indexes. Hence, we have $S = \{i_1\}$, and the retrieval benefits of the remaining indexes are revised accordingly. The objective function values after the inclusion of i_1 are shown in Table 3. Note that selecting i_1 reduces the benefits of i_2 and i_4, showing the effect of index interaction. Index i_3 now has the maximal objective function value and is thus added to S in the second iteration, leading to $S = \{i_1, i_3\}$.

Index	Retrieval Benefit	Objective Function
i_2	$0.8 \times 0 + 0.2 \times 0 = 0$	$0 - 0.2 = -0.2$
i_4	$0.8 \times 0 + 0.2 \times 2 = 0.4$	$0.4 - 0.2 = 0.2$

Table 4: The objective function values for indexes after i_1 and i_3 are selected by GD.

Index set	Retrieval Benefit	Objective Function
$\{i_1, i_2\}$	$0.8 \times 3 + 0.2 \times 2 = 2.8$	$2.48 - (0.25 + 0.2) = 1.95$
$\{i_1, i_3\}$	$0.8 \times 3 + 0.2 \times 2 = 2.8$	$2.48 - (0.25 + 0.1) = 2.45$
$\{i_1, i_4\}$	$0.8 \times 3 + 0.2 \times 2 = 2.8$	$2.48 - (0.2 + 0.25) = 2.35$
$\{i_2, i_3\}$	$0.8 \times 2 + 0.2 \times 2 = 2$	$2 - (0.2 + 0.1) = 1.7$
$\{i_2, i_4\}$	$0.8 \times (2 + 2) + 0.2 \times 2 = 3.6$	$3.6 - (0.2 + 0.2) = 3.2$
$\{i_3, i_4\}$	$0.8 \times 2 + 0.2 \times (2 + 2) = 2.4$	$2.4 - (0.2 + 0.1) = 2.1$

Table 5: The objective function values of index subsets of cardinality no greater than two.

In the third iteration, the benefits for the remaining indexes are again re-evaluated and there is actually no change for the objective function values of i_2 and i_4. We then obtain Table 4. Note that at this point, set S uses $6 + 2 = 8$ units of storage. Since the maximal storage allowed is 12, there are only 4 units of storage available. Because i_4 is the only beneficial index remaining and needs a storage of 5, GD terminates at this stage, giving $S = \{i_1, i_3\}$. The update cost of the selected index set is $0.25 + 0.1 = 0.35$. The reduction in the query retrieval cost is $0.8 \times 3 + 0.2 \times 2 = 2.8$. The overall reduction in the cost resulted from using $\{i_1, i_3\}$ is thus $2.8 - 0.35 = 2.45$, larger than those resulted from NV and PO.

Algorithm LH: Now, consider the application of LH(4,2) to this problem. In contrast to considering individual indexes as GD, LH(4,2) takes into consideration all subsets of the set $\{i_1, i_2, i_3, i_4\}$ of cardinality less than or equal to two. The four singleton sets corresponding to the four indexes will have their costs and benefits identical to those shown in Table 1. In addition, the following subsets of indexes are evaluated in Table 5. The set $\{i_2, i_4\}$ has the largest objective function value among all the candidate sets, and is therefore selected as the set IS. After i_2 and i_4 in IS being evaluated, i_4 is included into S for its better performance. It is worth mentioning that the lookahead has led us to select i_4 first, which has a smaller individual objective function value than i_1. Given $S = \{i_4\}$, we obtain the results in Table 6 by revising the benefit numbers. Next, it is obtained from Table 6 that the set $\{i_2, i_3\}$ is the one with maximal objective function value to be chosen as IS, which in turn leads to the inclusion of i_2. Thus, $S = \{i_2, i_4\}$,

Following the above procedure, i_3 will be added into S in the next iteration, completing the search by LH(4,2). The final solution obtained by LH(4,2) is $S = \{i_2, i_3, i_4\}$. This set has an

Index set	Retrieval Benefit	Objective Function
$\{i_1\}$	$0.8 \times 1 + 0.2 \times 0 = 0.8$	$0.8 - 0.25 = 0.55$
$\{i_2\}$	$0.8 \times 2 + 0.2 \times 0 = 1.6$	$1.6 - 0.2 = 1.4$
$\{i_3\}$	$0.8 \times 0 + 0.2 \times 2 = 0.4$	$0.4 - 0.1 = 0.3$
$\{i_1, i_2\}$	$0.8 \times 2 + 0.2 \times 0 = 1.6$	$1.6 - (0.25 + 0.2) = 1.15$
$\{i_1, i_3\}$	$0.8 \times 1 + 0.2 \times 2 = 1.2$	$1.2 - (0.25 + 0.1) = 0.85$
$\{i_2, i_3\}$	$0.8 \times 2 + 0.2 \times 2 = 2.0$	$2.0 - (0.2 + 0.1) = 1.7$

Table 6: The objective function values for LH(4,2) after i_4 is included.

update cost of $0.2 + 0.2 + 0.1 = 0.5$, and the overall reduction in the query evaluation cost is $0.8 \times (2 + 2) + 0.2 \times 2 = 3.6$, meaning that the net benefit of utilizing $\{i_2, i_3, i_4\}$ is $3.6 - 0.5 = 3.1$, larger than those resulted by the previous schemes. As a matter of fact, it can be verified that the solution by LH(4,2) is the optimal one for the given database profile.

4 Performance Study

We conduct a performance study for index selection algorithms in this section. The methodology employed is described in Section 4.1. Experiments and their results are shown in Section 4.2. Different values for the amount of storage for indexing, the update and storage costs, and the attribute selectivity are used in the simulation to conduct a sensitivity analysis for these parameters. Theoretical results on index interaction are presented in Section 4.3.

4.1 Methodology

An OODB system simulator is built in C++ to model the detail of data retrievals under different indexed environments. The input to the simulator consists of a schema graph and a number of logical database parameters. Using these database parameters, the database population for our simulation is randomly generated. However, since we believe OODB schema graphs have certain important properties that random graphs do not possess in general, the schema graph is not generated randomly. Instead, we employ in our simulation the schema graph shown in Figure 5, which is essentially based on the one reported in the 007 benchmark [4], except two modifications. First, for ease of exposition, we do not consider the effect of subtyping which is in fact orthogonal to the main theme of this study. Hence, the superclass of two classes in the benchmark in [4] is represented as a separate class, denoted by an extra node (node 9) in Figure 5. Second, we have

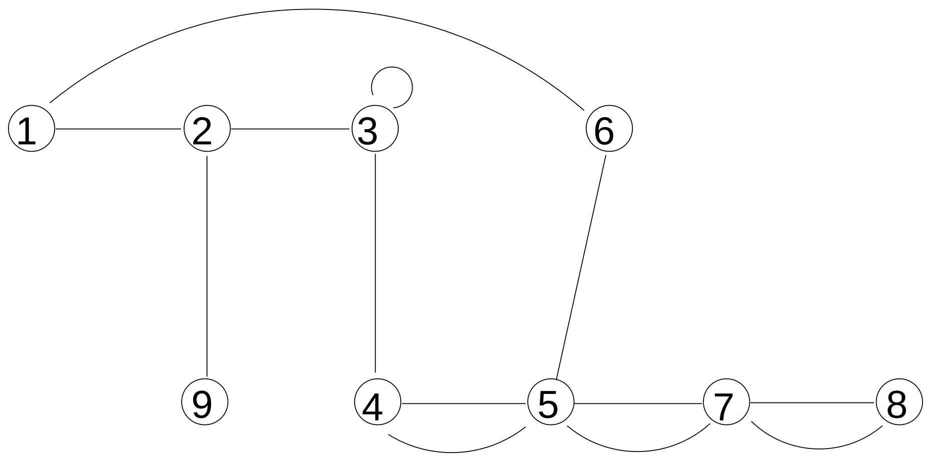

Figure 5: Schema graph used in the simulation.

included an additional attribute, represented by the arc between node 1 and node 6 in Figure 5, in order to have large cycles in the schema graph, thus providing more general results.

Note that each edge actually represents a pair of attributes, i.e., the forward attribute and the corresponding reverse reference. A path query is specified by a path in the schema graph. Since the schema graph is actually a multigraph, there can be many different queries with the same starting and ending nodes. Queries are randomly generated as follows. First, the query length is randomly determined between the parameters $Qlen_{min}$ and $Qlen_{max}$. Then, the starting node of the query is randomly selected from those in the schema graph. A path is thus formed by a random walk in the schema graph, which starts from the starting node and moves via a random outgoing arc to its neighboring node in each step until the path length is reached. The number of queries generated is equal to a predetermined number, $NumQueries$. Each query is assigned a frequency for its occurrence in such a way that the sum of all the frequencies is equal to one. Node cardinalities and sizes are determined randomly from the ranges $[Card_{min}, Card_{max}]$ and $[Size_{min}, Size_{max}]$, respectively. Similarly, arc update frequencies are randomly assigned such that their sum is equal to one. Each attribute is assigned with a selectivity, k, which corresponds to the ratio of the number of different attribute values to the cardinality of the target class. Simulation parameters and their typical values are given in Table 7.

Parameter	Typical Value	Meaning
NumQueries	10	number of queries generated
$[\text{Qlen}_{min}, \text{Qlen}_{max}]$	$[2, 10]$	query length range
$[\text{Card}_{min}, \text{Card}_{max}]$	$[10, 1000]$	node cardinality range
$[\text{Size}_{min}, \text{Size}_{max}]$	$[100, 1000]$	node size range (bytes)
wtRetrBen	512	ratio of retrieval queries to updates
valFrac	0.8	attribute selectivity
arcInsFrac	0.3	fraction of arc modifications that are insertions
arcDelFrac	0.2	fraction of arc modifications that are deletions
arcUpdFrac	0.5	fraction of arc modifications that are updates
OIDL	8	length of object identifier
kl	8	key length
kll	2	size of key-length field
rl	2	size of record-length field
nuid	2	size of "number of OIDs" field
P	4096	page size
pp	4	page pointer size
d	146	order of a nonleaf node
fanout	218	average fanout from a nonleaf B-tree node

Table 7: Values of some database and system parameters used in the simulation.

4.2 Experiments and Their Results

Four algorithms, NV, PO, GD and LH, will be comparatively studied in Section 4.2.1. Three objective functions are evaluated in Section 4.2.2. The effect of update frequency and storage overhead is studied in Section 4.2.3 and that of attribute selectivity is investigated in Section 4.2.4. Due to the nature of random generation, two simulation runs based on the same set of parameters might yield different results. Therefore, for the same set of data, several simulation runs are performed, and the final statistics are obtained by averaging those from all runs.

4.2.1 Comparison for index selection algorithms

Four algorithms, NV, PO, GD and LH, are comparatively studied. Recall that NV corresponds to a random inclusion of indexes, and PO selects indexes according to their individual profits without dynamically revising those profits. In contrast, GD revises the profits of all the remaining indexes after every inclusion of an index, thus taking index interaction into account. At the cost of higher search complexity, LH evaluates the profits of indexes several steps ahead before their potential inclusion into the index list. Performance of four index selection schemes using the objective on

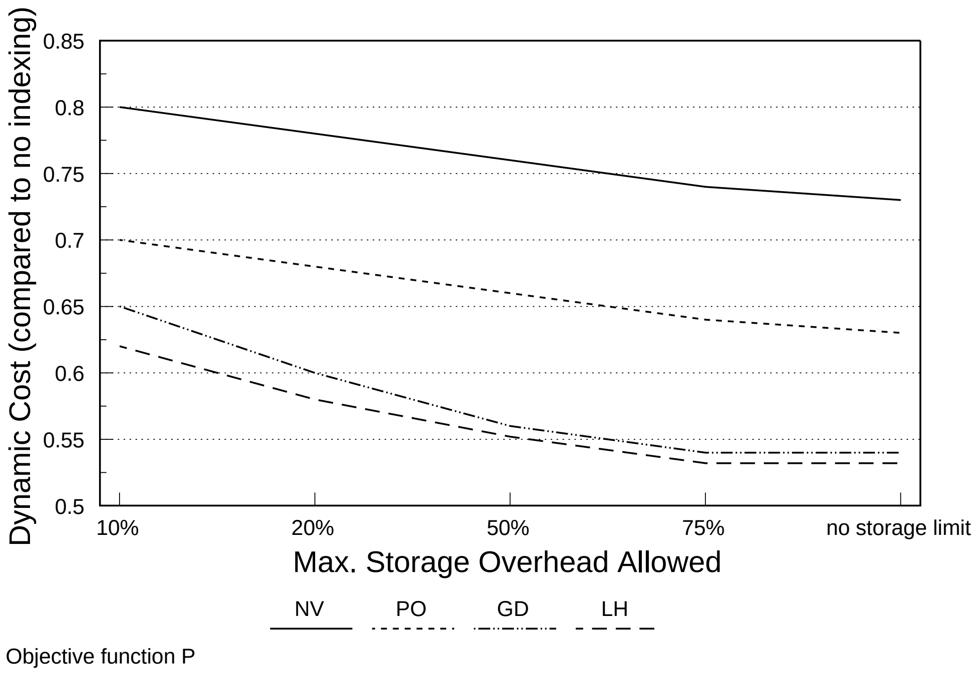

Figure 6: Performance of four index selection schemes.

profit[4] is given in Figure 6, where the ordinate is the ratio of the dynamic cost with indexing to that without indexing and the abscissa denotes the amount of storage overhead allowed[5]. Recall that the dynamic cost is the sum of the retrieval cost of database queries and the update cost for indexes in response to database updates.

It can be seen from Figure 6 that the dynamic cost required by an indexed system is in general decreasing as the amount of storage available for indexing increases, meaning that more improvement on the dynamic cost can be achieved by allowing a larger storage for indexing. Failing to consider the effect of index interaction, NV and PO are clearly outperformed by GD and LH. In this experiment LH is implemented as LH(10,50), a very extensive search, which we believe will mostly lead to the optimal solution. However, it can be seen that even with the high search order of LH, GD performs fairly close to LH, except when the amount of storage for indexing is small, showing that GD is very practically useful and the necessity of employing a high order search needs further justification. More insights into the reason for the good performance of GD will be provided in Section 4.3. Nevertheless, when the amount of storage for indexing is small, meaning that only very few indexes could be built, LH outperforms GD for its prudent selection for indexes.

[4]Results from using the other two objective functions do not provide additional insights, and are thus omitted here.

[5]For clarity, the amount of storage overhead allowed indicated in the abscissa is the one normalized by the original database size.

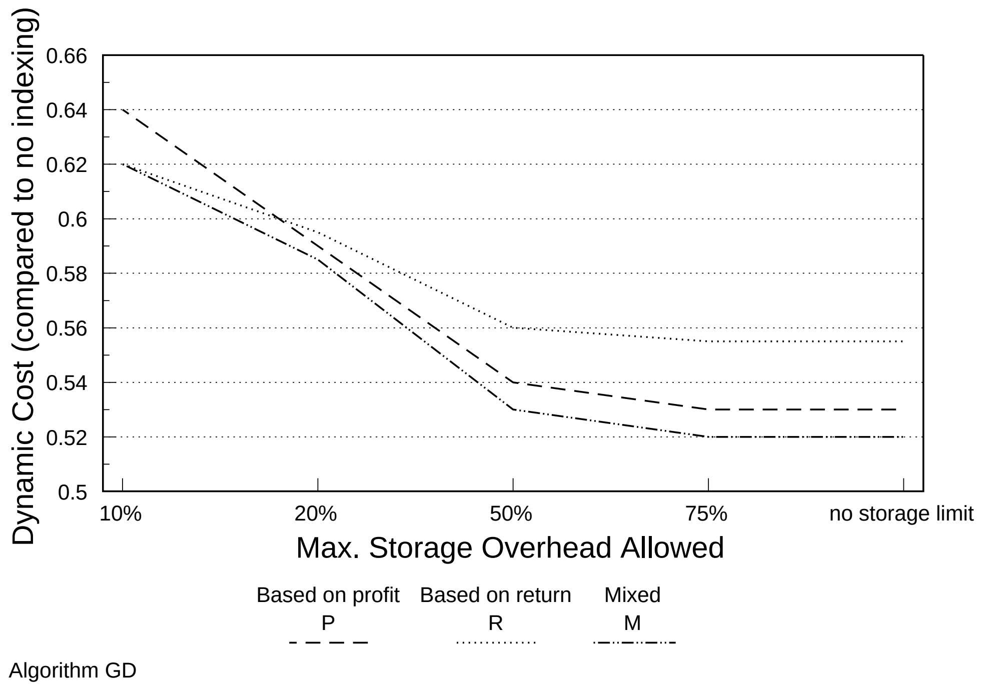

Figure 7: Comparison of three objective functions.

4.2.2 Comparison for the objective functions

Three objective functions, which guide the search for profitable indexes, are evaluated. The effects of the three objective functions are shown in Figure 7, where GD is used and the parameter α for $M(\cdot)$ is chosen to be 0.5 for its good performance. It can be seen from Figure 7 that the mixed objective function $M(\cdot)$, which based on the amount of storage available, adaptively selects its formula to evaluate indexes, emerges as the winner. It is interesting to see that $P(\cdot)$ performs better than $R(\cdot)$, except when the amount of storage is small. Clearly, when the amount of storage is limited, it is important to consider the storage overhead to select indexes. On the other hand, when there is an adequate amount of storage available it is better to consider the profit than the return ratio for index selection. Note that for the same amount of available storage for indexing, $P(\cdot)$ usually includes a few indexes with large profits, whereas $R(\cdot)$ tends to include more indexes, each of which, though consuming a fewer amount of storage, is less profitable. The more indexes, the more severe the effect of index interaction could be, thus accounting for the results in Figure 7. Performance of the three index selection schemes under different objective functions is shown in Figure 8, where it can be seen that the relative performance of these objective functions is consistent over these index selection schemes.

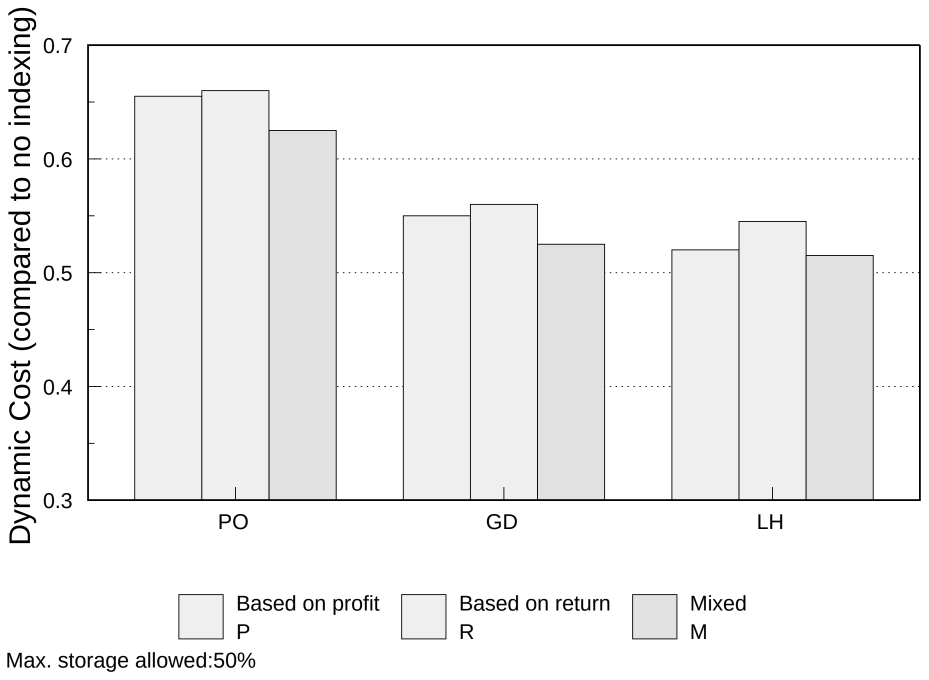

Figure 8: Performance of the index selection schemes under different objective functions.

4.2.3 Effect of update frequency and storage overhead

Indexing in the nested object hierarchy could be costly in terms of the storage required and the update cost incurred. In this experiment we study the effect of varying the amount of storage allowed and also that of varying the retrieval-update ratio on the performance of indexing. Basically, indexing will facilitate query retrieval, but incurs an additional update cost in response to database updates. The relative dynamic cost for different storage overheads allowed for indexing is shown in Figure 9, where different retrieval-update ratios are investigated under GD with the objective function $M(\cdot)$. It is again observed that increasing the amount of storage allowed increases the benefit of indexing in general. However, this benefit is essentially bound by the update rate. For a retrieval-update ratio of 32, which corresponds to an environment with a high update rate, it can be seen from Figure 9 that having more storage does not yield any improvement since the solution index set is bound by the update cost. As the retrieval-update ratio increases, meaning that the relative update ratio decreases, the solutions tend to become more storage-bound, and having more storage thus yields better solutions.

Figure 10 shows the same data set plotted with the retrieval-update ratio on the abscissa. It can be observed that having fewer updates leads to a better solution up to the point where the solution becomes storage-bound. This is confirmed by Figure 11 which shows the actual storage used by the selected indexes. As a matter of fact, it can be verified from Figure 11 that as the

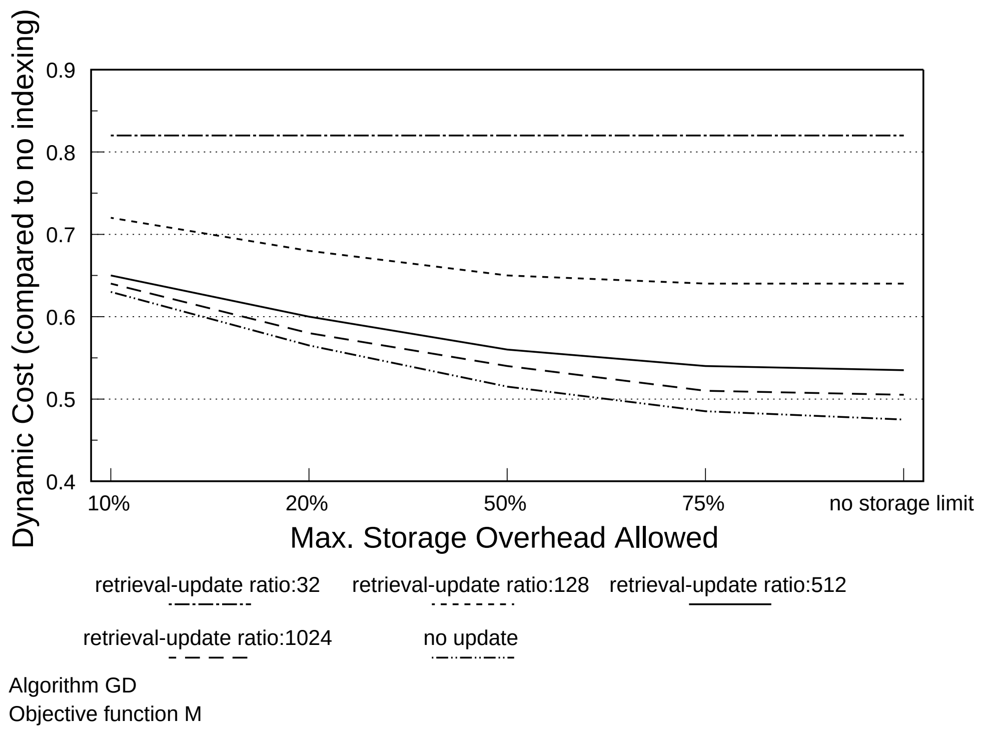

Figure 9: The effect of indexing for different storage overheads.

amount of storage increases and also as the update rate increases, the ratio of the actual storage used to the maximal storage allowed (i.e., the slope of a curve) decreases, explaining the saturation of performance improvement in Figure 9 and Figure 10.

4.2.4 Effect of attribute selectivities

Note that the term $k(1, n)$, i.e., the product of the selectivities of attributes in the index path, is an important term to determine the costs for path indexes. It is therefore essential to study the effect of varying attribute selectivities on the performance of indexing. Without loss of generality, algorithm GD and objective function $M(\cdot)$ are used to conduct this experiment. It is shown in Figure 12 that larger values of the attribute selectivity lead to better indexing performance. Note that as the selectivity ratio increases, meaning that the number of instances of the referenced objects on a path in the schema graph increases, the potential benefit achievable by indexing increases, accounting for the results in Figure 12.

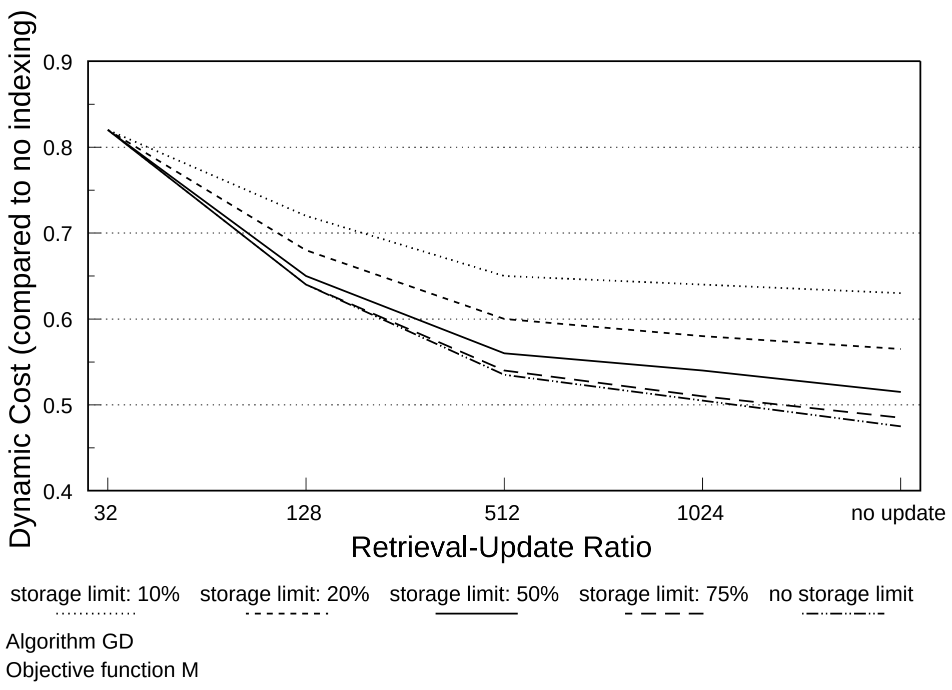

Figure 10: The effect of indexing for different update costs.

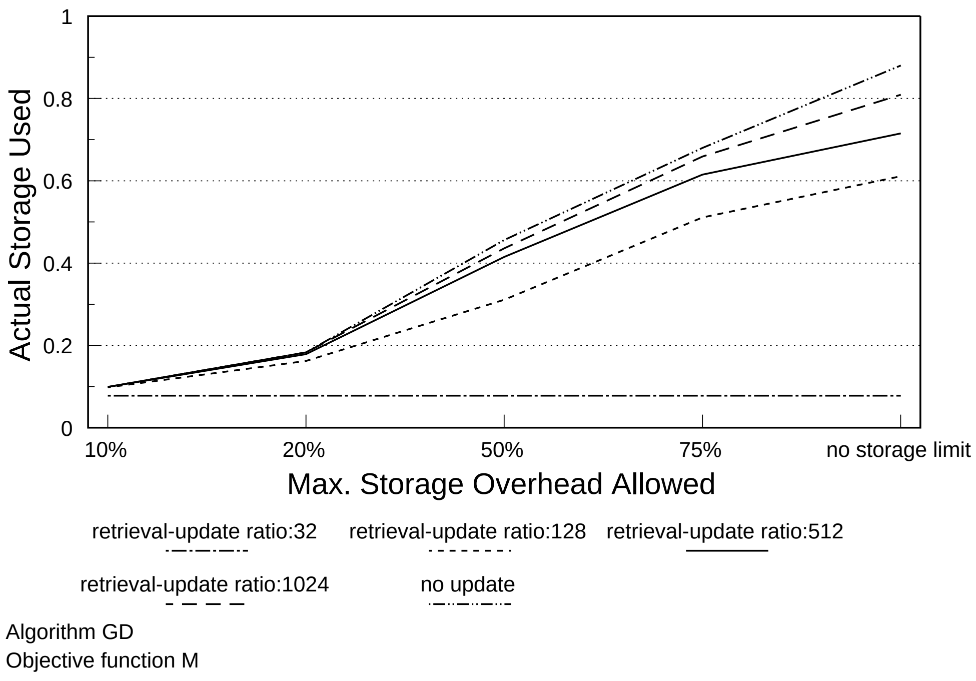

Figure 11: Storage overhead by indexing schemes.

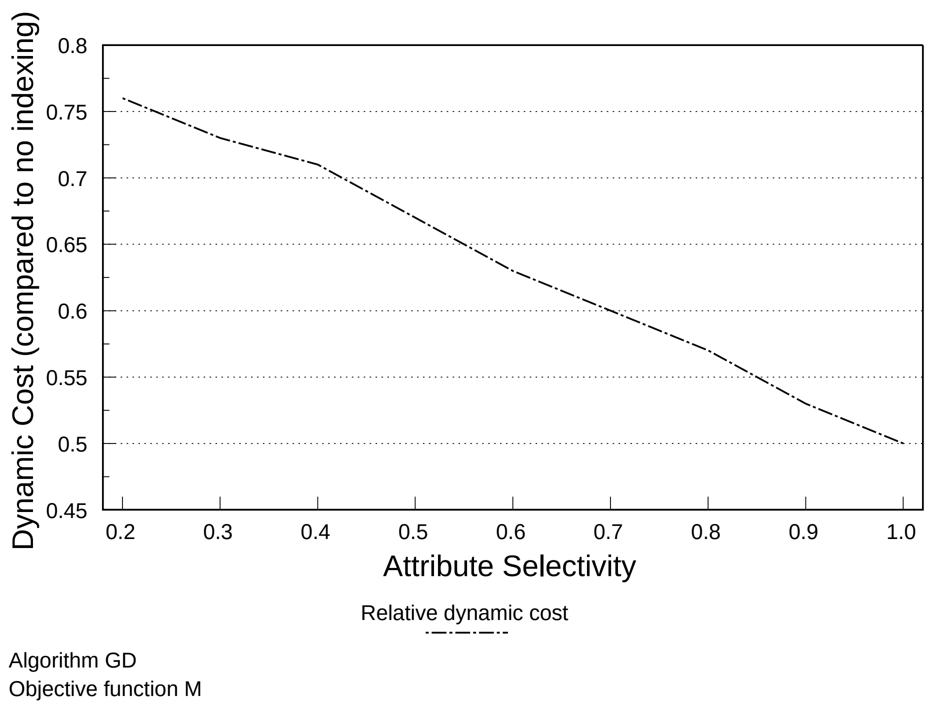

Figure 12: Effect of attribute selectivity.

4.3 Characteristics for Index Interaction

It is noted that in addition to the greedy method used in this study, such techniques as state-transition search and simulated annealing can also be employed to tackle this global optimization problem [9] [14]. However, it is observed that the quality of the solutions provided by GD is fairly good. Explicitly, from the simulation results, it is observed that the quality of the solutions resulted by GD is very close to the one resulted by LH which requires a much higher search cost. This is in fact due to the very nature of this optimization problem. Note that if we ignore the storage cost constraint and also assume that the benefit of one index is not affected by the inclusion of others, then GD will lead to the optimal solution. With the storage constraint, the problem becomes a variation of the well-known knapsack problem which is NP-hard. The phenomenon of index interaction further complicates the search problem, and clearly hinders GD from resulting in the optimal solution. This indicates that the complexity of the problem studied is indeed comparable to that encountered by similar optimization problems [19].

However, we have observed in our simulation that in many cases the storage limit is not the limiting factor for the solution. For example, in Figure 11 the solutions found by these index selection algorithms usually consume less storage than the maximum allowed. In these cases, the solution is bound by the index update cost, meaning that the complication due to the storage constraint is absent in many cases. Furthermore, as explicitly characterized by the theorem below,

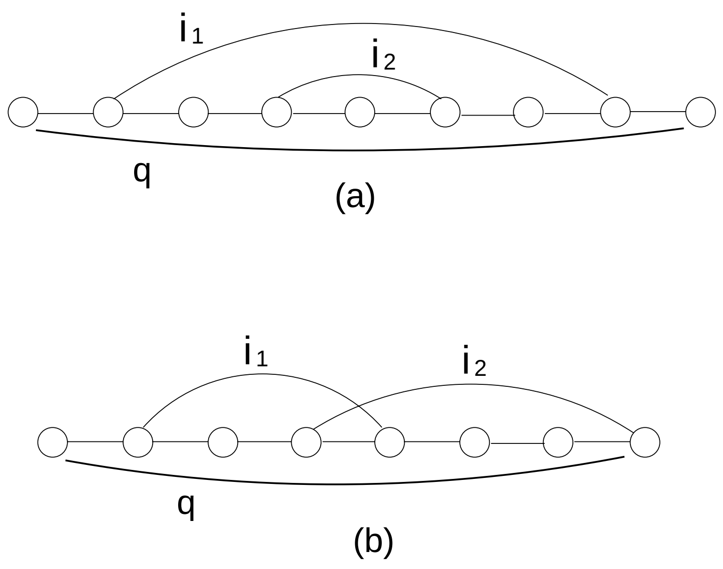

Figure 13: Index interaction.

it can be seen that the index interaction only occurs in very limited patterns.

Theorem 1: Suppose p_1 and p_2 are the paths indexed by i_1 and i_2, respectively. Then indexes i_1 and i_2 interact if and only if (1) there exists a query q in the set of given queries such that both i_1 and i_2 can be used in evaluating q, and (2) by considering p_1 and p_2 as strings of arc-identifiers, either of the following two conditions must hold:

- One is a substring of the other (such as the situation shown in Figure 13(a)).
- There exists a string s that is a suffix of one and a prefix of the other (such as the situation shown in Figure 13(b)).

Note that for i_1 and i_2 to interact with each other, Theorem 1 requires that paths p_1 and p_2 be contained in the path of q, and also that these two indexes not be in tandem. Therefore, cases such as i_1 and i_2 shown in Figure 14 do not have index interaction. It can be seen that Theorem 1 imposes a strict limitation on the patterns for the occurrences of index interaction, which are in fact within the degree of sophistication that GD could cope with, thus explaining the good performance of GD in most cases.

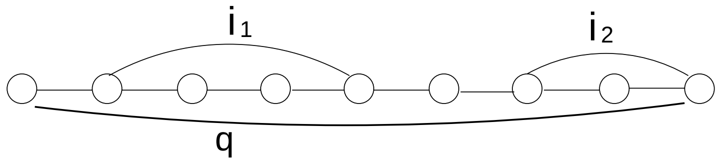

Figure 14: Two indexes that do not interact with each other.

5 Conclusion

We studied in this paper the problem of devising a set of indexes for a nested object hierarchy to improve the overall system performance. Performance was measured in terms of the retrieval, update and storage costs of an indexed system. The index selection problem was first formulated and four index selection algorithms were evaluated via simulation. The effects of objective functions, which guide the search for candidate indexes, were also investigated. It has been shown by simulation results that GD which is devised in light of the phenomenon of index interaction performs fairly well in most cases, which in fact agrees with the very nature of index interaction we identified in this study. Sensitivity analysis for various parameters was conducted. We not only conducted an extensive performance study for index selection algorithms, but also explored the effect of index interaction to deal with this global optimization problem.

References

[1] E. Bertino. Optimization of Queries Using Nested Indices. In *International Conference on Extending Database Technology*, March 1990.

[2] E. Bertino and C. Guglielmina. Optimization of Object-Oriented Queries Using Path Indices. In *Second International Workshop on Research Issues on Data Engineering: Transaction and*

Query Processing, pages 140–149, February 1992.

[3] E. Bertino and W. Kim. Indexing Techniques for Queries on Nested Objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):196–214, June 1989.

[4] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. *Proceedings of ACM SIGMOD*, May, 1993.

[5] R. G. G. Cattell. *Object Data Management: object-oriented and extended relational database systems.* Addison-Wesley Publishing Company, Inc., 1991.

[6] S. Choenni, E. Bertino, H. M. Blanken, and T. Chang. On The Selection of Optimal Index Configuration in OO Databases. *Proceedings of the 10th International Conference on Data Engineering*, pages 526–537, February 1994.

[7] S. Christodoulakis. Implication of Certain Assumptions in Database Performance Evaluation. *ACM Transactions on Database Systems*, 9(2):163–186, June 1984.

[8] F. Fotouhi, T.-G. Lee, and W. I. Grosky. The Generalized Index Model for Object-Oriented Database Systems. In *10th Annual International Phoenix Conference on Computers and Communication*, pages 302–308, 1991.

[9] Y. E. Ioannidis and E. Wong. Query Optimization by Simulated Annealing. *Proceedings of ACM SIGMOD*, pages 9–22, May, 1987.

[10] A. Kemper and G. Moerkotte. Access Support Relations: An Indexing Method for Object Bases. *Information Systems*, 17(2):117–145, 1992.

[11] K.-C. Kim, W. Kim, and A. Dale. Cyclic Query Processing in Object-Oriented Databases. In *IEEE International Conference on Data Engineering*, pages 564–571, 1989.

[12] W. Kim. *Introduction to Objected-Oriented Databases.* The MIT Press, Cambridge, Massachusetts, 1990.

[13] W. Kim, K.-C. Kim, and A. Dale. Indexing in an Object-Oriented Database. Technical Report DB-134-87, MCC, 1987.

[14] S. Lafortune and E. Wong. A State Transition Model for Distributed Query Processing. *ACM Transactions on Database Systems*, 11(3):294–322, September 1986.

[15] C. C. Low, H. Lu, and B. C. Ooi. Efficient Access Methods in Deductive and Object-Oriented Databases. In *International Conference on Deductive and Object-Oriented Databases*, 1991.

[16] D. Maier and J. Stein. Indexing in an Object-Oriented DBMS. In *Proc. Int. Workshop on OODB Systems*, pages 171–182, 1986.

[17] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an Object-Oriented DBMS. In *OOPSLA, conference on Object-oriented Programming Systems, Languages and Applications*, pages 472–482, July 1986.

[18] M. T. Ozsu. Query Processing Issues in Object-Oriented Database Systems—Preliminary Ideas. In *1991 Symposium on Applied Computing*, pages 312–324, 1991.

[19] T. K. Sellis. Efficiently Supporting Procedures in Relational Database Systems. *Proceedings of ACM SIGMOD*, pages 278–291, May, 1987.

[20] J. D. Ullman. A Comparison Between Deductive and Object-Oriented Database Systems. In *International Conference on Deductive and Object-Oriented Databases*, December 1991.

APPENDIX: Formulas for storage, retrieval and update costs

Storage Cost

Average number of C(1) nodes with the same value of a nested attribute A(n):

$k(1,n) = \prod_{i=1}^{n} k(i)$.

Note that we do not simplify $k(1,n)$ to $N(1)/D(n)$ as in [3] since we take into consideration the existence of partial instantiations of attributes[6].

Average length of a leaf-node (of B-tree) index record:

$\mathrm{XN} = k(1,n) \times OIDL + kl + ol$.

Number of B-tree leaf pages in index:

$$LP = \begin{cases} \lceil D(n)/\lfloor P/\mathrm{XN} \rfloor \rceil, & \text{if XN} \le P, \\ D(n) \times \lceil (\mathrm{XN} + DS)/P \rceil, & \text{otherwise.} \end{cases}$$

where $DS = \lceil (k(1,n) \times OIDL + kl + ol)/P \rceil \times (OIDL + pp)$ is the length of the directory for records larger than one page.

Number of non-leaf pages in index:

$NLP = \lceil LO/f \rceil + \lceil \lceil LO/f \rceil / f \rceil + \ldots + X$,
where X is the first term less than f .

Total storage cost of index is $LP + NLP$ pages.

Retrieval Cost

[6]Given a path $C(1).A(1).A(2).\ldots.A(n)$, [3] uses $N(1)$ to denote for the cardinality of $C(1)$, and $D(i)$ to denote for the number of different values of attribute $A(i)$.

Using the corresponding index the cost of retrieving the objects of class $C(1)$ that have a nested attribute $C(1).A(1).\ldots.A(n)$ equal to a given value is the number of pages accessed during a B-tree lookup, which is determined by:

$$A = \begin{cases} h+1, & \text{if XN} \leq P, \\ h + \lceil \text{XN}/P \rceil, & \text{otherwise.} \end{cases}$$

where h = number of terms in expression for NLP is the height of the B-tree.

Update Cost

Every update to an attribute can potentially cause a path index to be updated. The overall update cost of an index is the sum of the update costs over all attributes. That is,

$\text{UpdCost}(I) = \Sigma_{\text{attributes\_A}} \text{update\_frequency(A)} \times (\text{cost\_of\_updating\_I\_when\_A\_is\_updated}).$

The index update cost when i^{th} attribute on the path is updated is determined by:

$U = CFT + pdiff \times (CBT + CBM).$

Cost of forward traversal: $CFT = 2(n - i)$.

Cost of backward traversal: $CBT = 2 * NO$.

NO is the number of objects accessed in backward traversal:

$NO = \sum_{j=1}^{i-1} (\prod_{p=i}^{j} k(p))$.

$$pdiff = \begin{cases} 1 - \left[\frac{(\prod_{j=i+1}^{n} k(j)) - 1}{N(i+1) - 1} \right], & \text{if } i < n, \\ 1, & \text{otherwise.} \end{cases}$$

Cost of B-tree update:

$CBM = CO \times (1 + pl)$.

Cost of B-tree operation:

$$CO = \begin{cases} h+2, & \text{if XN} \leq P, \\ h + 2 + (np - 1)/np, & \text{otherwise.} \end{cases}$$

Prob. that old and new values of A(n) are on different leaf nodes:

$$pl = \begin{cases} 1 - \left[\frac{\lceil P/\text{XN} \rceil - 1}{D(n) - 1} \right], & \text{if XN} \leq P, \\ 1, & \text{otherwise.} \end{cases}$$