INDEXING TECHNIQUES FOR OBJECT-ORIENTED DATABASES

Won Kim,* Kyung-Chang Kim,* and Alfred Dale

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-14 May 1987

ABSTRACT

In conventional database systems, an index is maintained on an attribute of a single class (or a relation) to speed up associative searches. In object-oriented databases, the access scope of a query against a class in general includes not only the class but also all subclasses of the class. This means that to support the evaluation of a query, the system must maintain one index on the attribute for each of the classes involved in the query. An alternative, and a new, approach is to maintain one index on the attribute for the class and all its subclasses. In this paper, we formulate the cost model for the size and performance of a B-tree index, and present the results of simulation studies which quantify the tradeoffs between these two indexing techniques in an object-oriented database system.

*Microelectronics and Computer Technology Corporation, Austin, Texas 78759

1. INTRODUCTION

In recent years, object-oriented programming [GOLD81, GOLD83, BOBR83, CURR84, SYMB84, LMI85] has gained a tremendous popularity in the design and implementation of a variety of data-intensive application systems. These include artificial intelligence (AI) [STEF86], computer-aided design and manufacturing (CAD/CAM) [AFSA86], and office information systems (OIS) with multi-media documents [IEEE85, AHLS84, WOEL86]. Object-oriented programming offers a number of important advantages for these applications over traditional control-oriented programming. One is the modeling of all conceptual entities with a single concept, namely objects. An object represents anything from a simple number, say, the number 25, to a complex entity such as an automobile or an insurance agency. The state of an object is captured in the instance variables. The behavior of an object is captured in messages to which an object responds. The messages completely define the semantics of an object. Another advantage of object-oriented programming is the notion of a *class hierarchy* and *inheritance* of properties (instance variables and messages) along the class hierarchy. The class hierarchy captures the IS-A relationship between a class and its *subclass* (equivalently, a class and its *superclass*). All subclasses of a class inherit all properties defined for the class, and can have additional properties local to them. The notion of property inheritance along the hierarchy facilitates top-down design of the database as well as applications.

In the Database Program at MCC, we have built a prototype object-oriented database system, called ORION. Presently, it is being used in supporting the data management needs of PROTEUS, an expert system development environment prototyped in the AI / KBS Program at MCC. In ORION we have directly implemented the object-oriented paradigm, added persistence and sharability to objects through transaction support, and provided various advanced functions that applications from the CAD/CAM, AI, and OIS domains require. Important features supported in ORION include predicate-based queries, versions, composite objects [BANE87a], dynamic schema evolution [BANE87b], and multimedia data management [WOEL87].

ORION supports, just as most conventional database systems do, secondary indexes on user-specified attributes (columns) of specified classes (relations) to speed up associative searches of the database for queries with search criteria. In formalizing a model of queries under the ORION object-oriented data model, we recognized that, while the scope of access of

a query against a single relation R in a relational database is just R, that of a query against a class C in an object-oriented database is in general the class C and all subclasses of C, and their subclasses, etc. This means that, since an attribute of a class C is inherited into all its descendant classes, it may make sense to maintain an index on an attribute for all classes on a class hierarchy rooted at class C, rather than maintaining a separate index on the attribute for each of the classes in the class hierarchy.

We will refer to an index which is maintained on an attribute of a single class as a *single-class index*, and an index on an attribute of all classes on a class hierarchy rooted at a particular class as a *class-hierarchy index*. ORION presently supports only single-class indexing. To determine the merit of class-hierarchy indexing, we have decided to quantify the tradeoffs between class-hierarchy indexing and single-class indexing, by formulating a reasonably simple cost model for the size (number of nodes) and height of a B-tree index. We have used the cost model for two extensive sets of experiments. One set of experiments was conducted to compare the size of a class-hierarchy index and the sum of the sizes of the corresponding set of single-class indexes. The other set of experiments compared the height of a class-hierarchy index and the sum of the heights of single-class indexes on the corresponding class hierarchy. The height of an index is a direct measure of performance of an index in evaluating a query which includes a predicate on an indexed attribute. Our experiments have led us to conclude that a class-hierarchy index outperforms single-class indexing when the scope of a query exceeds two classes. The results on the size of an index were inconclusive; sometimes a class hierarchy requires more storage space, and other times the corresponding set of single-class indexes takes up more space.

In this paper, we make two original contributions. First, we provide a formal model of a query under an object-oriented data model, and identify the utility of a class-hierarchy index. To our knowledge, the concept of a class-hierarchy index has not been discussed before in the literature. Second, we present the preliminary results of simulation experiments we have conducted on the size and performance tradeoffs between a class-hierarchy index and a corresponding set of single-class indexes.

The remainder of this paper is organized as follows. In Section 2, we provide a brief review of the basic object-oriented concepts which are relevant to understanding the subject matter of this paper. In Section 3, we describe a formal model of an object-oriented query, and identify

the utility of a class-hierarchy index in evaluating an object-oriented query. We present our model of a B-tree index in Section 4. Then in Section 5 we discuss the assumptions we have made in the cost model of an index we have formulated, and organization of the simulation experiments we have conducted based on the cost model. In Section 5 and 6, we formulate the cost model for the size and performance of an index, respectively, and present the experimental results. Section 7 summarizes the paper.

2. REVIEW OF OBJECT-ORIENTED CONCEPTS

In this section, we review basic object-oriented concepts that are relevant to our discussions in the remainder of this paper. This subsection is extracted from our full paper on the ORION data model in [BANE87a].

objects, attributes (instance variables), methods, and messages

In object-oriented systems, all conceptual entities are modeled as objects. An ordinary integer or string is as much an object as is a complex assembly of parts, such as an aircraft or a submarine. An object consists of some private memory that holds its state. The private memory is made up of the values for a collection of attributes (often called instance variables). The value of an attribute is itself an object, and therefore has its own private memory for its state (i.e., its attributes). A primitive object, such as an integer or a string, has no attributes. It only has a value, which is the object itself. More complex objects contain attributes, through which they reference other objects, which in turn contain attributes.

The behavior of an object is encapsulated in *methods*. Methods consist of code that manipulate or return the state of an object. Methods are a part of the definition of the object. However, methods, as well as attributes, are not visible from outside of the object. Objects can communicate with one another through messages. *Messages* constitute the public interface of an object. For each message understood by an object, there is a corresponding method that executes the message. An object reacts to a message by executing the corresponding method, and returning an object.

classes, class hierarchy, inheritance, and domains

If every object is to carry its own attribute names and its own methods, the amount of information to be specified and stored can become unmanageably large. For this reason, as

well as for conceptual simplicity, 'similar' objects are grouped together into a *class*. All objects belonging to the same class are described by the same set of attributes and methods. They all respond to the same messages. Objects that belong to a class are called *instances* of that class. (In this paper, we will use the terms instances and objects interchangeably.) A class describes the form (attributes) of its instances, and the operations (methods) applicable to its instances. Thus, when a message is sent to an instance, the method which implements that message is found in the definition of the class.

Grouping objects into classes helps avoid the specification and storage of much redundant information. The concept of a *class hierarchy* extends this *information hiding* capability one step further. A class hierarchy is a hierarchy of classes in which an edge between a pair of nodes represents the IS-A relationship; that is, the lower level node is a specialization of the higher level node (and conversely, the higher level node is a generalization of the lower level node). For a pair of classes on a class hierarchy, the higher level class is called a *superclass*, and the lower level class a *subclass*. The attributes and methods (collectively called properties) specified for a class are *inherited* (shared) by all its subclasses. Additional properties may be specified for each of the subclasses. A class inherits properties only from its immediate superclass. Since the latter inherits properties from its own superclass, it follows that a class inherits properties from every class in its *superclass chain*.

In object-oriented systems, the *domain* (which corresponds to data type in conventional programming languages) of an attribute is a class. The domain of an attribute of a class C may be explicitly bound to a specific class D. Then instances of the class C may take on as values for the attribute instances of the class D as well as instances of subclasses of D.

3. EVALUATION OF AN OBJECT-ORIENTED QUERY

The object-oriented data model, in its conventional form, is powerful enough to represent a complex object as a recursively nested object. An object may be defined with a set of instance variables. A class may be specified as the *domain* of an instance variable, and the domain class, unless it is a *primitive class* (such as the string, integer, or boolean class), in turn consists of a set of instance variables, and so on. The internal state of an object consists of the values of all its instance variables. The value of an instance variable is an instance of its domain, if the domain is a primitive class; and a reference to (*object identifier* of) an instance of

the domain, otherwise. For example, in Figure 1, we show the schema of a Vehicle class in terms of the instance variables Manufacturer, Body, Drivetrain, and Color. The domain of the Color instance variable is the primitive String class. The domain of the Manufacturer instance variable is the class Company, the instance variable Body has Autobody as its domain, and the domain of Drivetrain is the AutoDrivetrain class. The classes Company, AutoBody, and AutoDrivetrain each consist of their own set of instance variables, which in turn have associated domains (which for simplicity we do not show).

The nesting of an object through the domains of its attributes immediately suggests that to fully fetch an instance, the instance and all instances the instance references through its attributes must be recursively fetched. This means that to fetch one or more instances of a class, the class and all classes specified as non-primitive domains of the attributes of the class must be recursively traversed. For example, to fetch instances of the class Vehicle in Figure 1, the classes which need to be traversed include not only Vehicle, but also the non-primitive domains of Vehicle, namely, Company, AutoBody, AutoDrivetrain, as well as non-primitive domains of these classes.

In general, a query may be formulated against an object-oriented schema, which will fetch instances of a class which satisfy certain search criteria and to output only specified attributes of

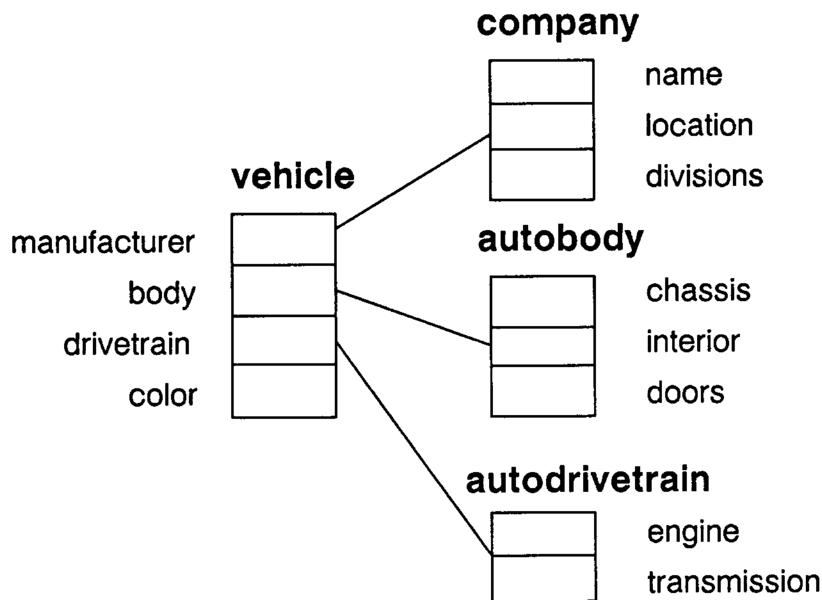

Figure 1. Nested Attributes of the Vehicle Class

the instances fetched. A query may restrict the instances of a class to be fetched by specifying predicates against any instance variables of the class. An example of a query, against the schema of Figure 1, is the following

Q1. Find all blue vehicles manufactured by Ford Motor Company

In an object-oriented database, an attribute may be one of two types: simple and complex. A *simple attribute* is one whose domain is a primitive class. A *complex attribute* is one whose domain is a class with one or more attributes, including complex attributes. A predicate on a simple attribute will be called a *simple predicate*; while one on a complex attribute will be called a *complex predicate*. Further, a query that involves only simple predicates will be called a *simple query*; and one that involves one or more complex predicates will be called a *complex query*.

We may represent a class and the domains of all its complex attributes in the form of a directed graph, which we will call the *query graph*. Each node on a query graph represents a class, and an edge from a node A to a node B means that the class B is the domain of a complex attribute of a class A. A query graph has only one root, the class whose instances are to be fetched. Each leaf node of a query graph has only simple attributes. A query graph may contain cycles.

The process of fetching nested objects, which we will call *object instantiation*, is similar to relational query evaluation. We may view a class as a relation and an attribute of a class as a column of a relation, and a relation to be augmented with a system-defined *unique identifier* (UID) column for the identifier of the tuples. Then the retrieval of an instance of the domain class D of an attribute A of a class C is similar to the relational join of a tuple of a relation C with a tuple of a relation D; where the join columns are column A of relation C and the UID column of relation D. We hasten to remark that, despite these similarities, there are a few significant differences between relational query evaluation and object instantiation. We will present a detailed discussion of these issues in a forthcoming report.

There is in general more than one way (often called a query-evaluation plan [SELI79]) for evaluating a query which will yield the correct result. However, each plan incurs a different cost. There are two fundamental options in plans for traversing the nested classes for object instantiation: forward and reverse traversal. The *query optimizer* of a database system is to consider a number of reasonable plans based on these options (and their combinations) for evaluating any given query, and to select one with the minimum expected cost.

In the *forward traversal*, the classes on a query graph are traversed in a depth-first order starting from the root of the query graph, and following through the successive domains of each complex instance variable. As an example, let us consider the example query Q1. A forward traversal of the query graph will start off with the set of all instances of the class Vehicle in which the Color attribute has a value 'blue.' For each of these instances, the value of its attribute Manufacturer is extracted; that value is an instance of the class Company. The value of the attribute Name in the Company instance is then examined. If the value is the string "Ford", the Company instance qualifies, and in turn, the Vehicle instance which has that Company instance as its manufacturer satisfies the query.

Another way to perform object instantiation is the *reverse traversal*, in which the leaf classes of a query graph are visited first, and then their parents, working toward the root class. As an example, let us consider once again the query Q1. Instead of starting with the set of all instances of Vehicle, query evaluation starts with the class Company. All instances of Company are identified which have the string "Ford" in the Name attribute. The UIDs of these instances are then looked up in the Manufacturer attribute of the class Vehicle. The result of the query is the set of instances of Vehicle which has the string 'blue' in the Color attribute and which contain in the Manufacturer attribute a UID that is in the list of UIDs for Ford Motor Company instances.

To support efficient retrieval of tuples that satisfy search predicates, the storage subsystem of relational database systems usually support secondary indexes on user-specified columns of relations [STON76, IBM81]. Similarly, object-oriented database systems may maintain an index on an attribute of a class. For example, if an index is maintained on the primitive attribute Name of the class Company, it can be used to advantage in a reverse traversal of the query graph for our example query Q1. On the other hand, if there is an index on the Color attribute of the class Vehicle, it may be used in a forward traversal of the query graph. In either case, the use of an index can significantly reduce the I/O cost of traversing the query graph for object instantiation.

One of the major differences between a relational database and an object-oriented database is that in an object-oriented database a class may be specialized into a number of subclasses. For example, in Figure 2, we show a database class hierarchy which includes the class Vehicle and the domain classes of the attributes of the class Vehicle. The class Vehicle is shown to have been specialized into the class Automobile and the class Truck. Similarly, the class Company has subclasses VehicleCompany and ComputerCompany. In general, a class

may have any number of subclasses and/or superclasses. The root of a class hierarchy is a system-defined class OBJECT, and any class the user defines without a superclass is by default a subclass of the class OBJECT [BANE87b].

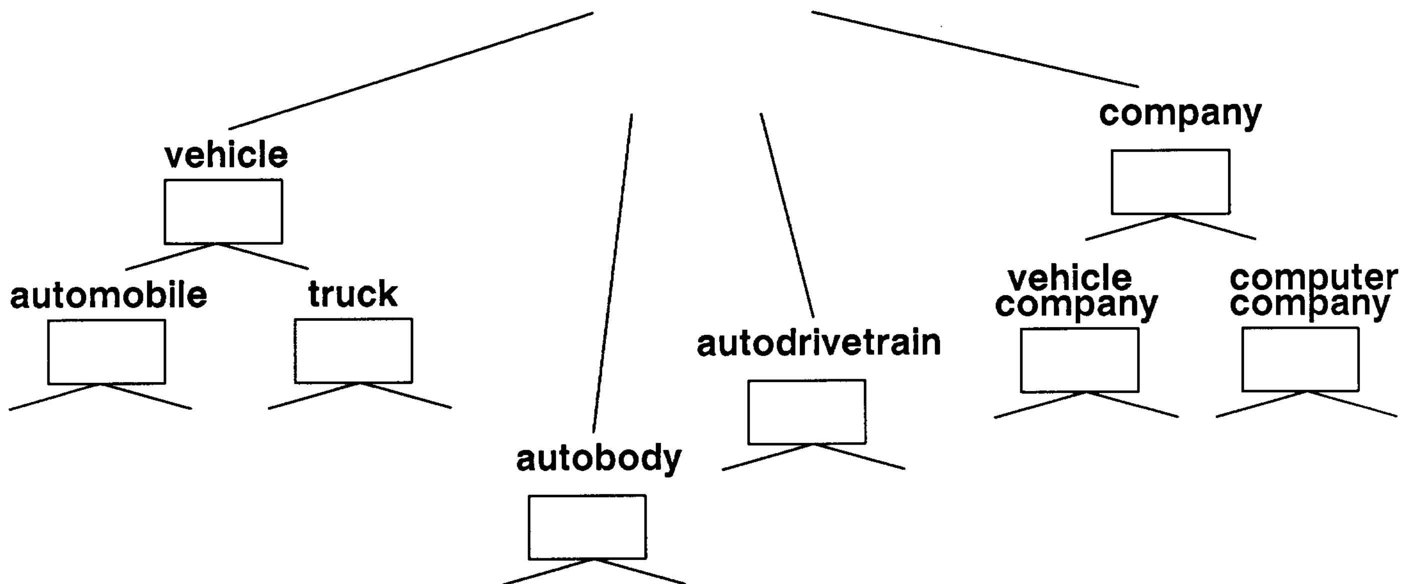

Figure 2. A Class Hierarchy

The fact that an object-oriented database schema explicitly captures the IS-A relationship between a pair of classes has two major impacts on the semantics of object instantiation. One is that the access scope of a query against a class may be only the instances of the class, or it may encompass the instances of the class and those of all subclasses of the class. For example, the user may issue a query against the class Vehicle to fetch only the instances of the class Vehicle, or may issue a single query against the class Vehicle to fetch all qualified instances of the class Vehicle and subclasses of Vehicle.

Another major impact is that the domain D of an attribute of a class C is really the class D and all subclasses of D. For example, the Manufacturer attribute of the class Vehicle may take on as value an instance of the class Company or an instance of any subclass of Company. This means that in the reverse traversal of the query graph for Q1, the class Company and all its subclasses must be traversed.

These semantics of object instantiation force major changes in the way a database system can use indexes. Traditionally, an index has been maintained on an attribute of a single class

(or a relation). This means that to support the evaluation of a query whose access scope is a class hierarchy, the system must maintain one index on the attribute for each of the classes in the class hierarchy. However, it is clear that often it may make sense to maintain one index on the attribute for a class hierarchy, and use it to evaluate queries against any single class in the class hierarchy or any sub-hierarchy of the class hierarchy. We will call the traditional approach of maintaining one index per class *single-class indexing*, and refer to the alternative approach of maintaining one index on an attribute for a hierarchy of classes *class-hierarchy indexing*.

Intuitively, it appears that a class-hierarchy index may in general be more effective in evaluating a query whose access scope spans a major subset of the classes in the indexed class hierarchy, while a single-class index should be more appropriate for a query against a single class. In the remainder of this paper, we quantify the tradeoffs between these two indexing techniques in an object-oriented database in terms of storage requirements and I/O performance in object instantiation.

4. INDEX STRUCTURE

In this section we describe the formats of the nodes of the B-tree index which we will model. These formats are based on the single-class B-tree index we have implemented in ORION. It is also similar to that used in IBM's relational database system SQL/DS [IBM81]. In a relational database, the columns have primitive data types; as such, the key values in an index are primitive data such as integers or strings. In an object-oriented database, the domain of an attribute may be either a primitive class or some user-defined class. Therefore, the key values in an index can be either the UIDs of the instances of the domain class or some primitive values.

Figure 3 shows the format of a non-leaf node. The node consists of f records, where each record is a pair (key, pointer), and key in turn is a pair (key-length, key-value), where key-length is the length in bytes of the key-value. The fanout, f, is between d and 2d, where d is the *order* of a B-tree. The fanout of the root node can be between 2 to 2d records. The pointer in each record contains the physical address of the next-level index node. If a record needs to be inserted into a node that contains 2d records, the node is split and the 2d+1 records are distributed to 2 nodes.

A leaf node of an index has a different format from that of a non-leaf node. Further, the format of a leaf node of a single-class index is different from that of a class-hierarchy index, as

shown in Figures 4a and 4b. An index record in a leaf node of a single-class index consists of the record-length, key-length, key-value, overflow-page pointer, the number of elements in the list of UIDs of the objects which hold the key-value in the indexed attribute, and the list of UIDs.

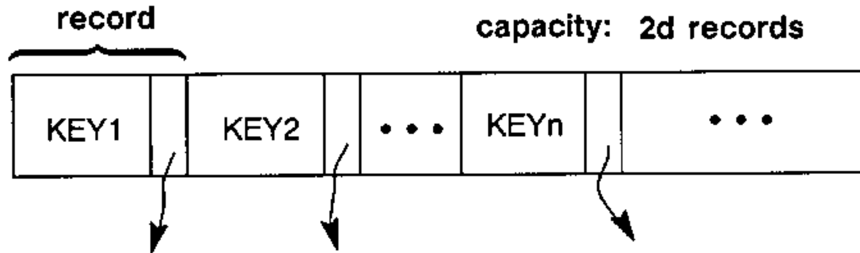

Figure 3. A Non-Leaf Node

An index record in a leaf node of a class-hierarchy index consists of the record-length, key-length, key-value, overflow-page pointer, key-directory, and, for each class in the class hierarchy, the number of elements in the list of UIDs for the objects which hold the key-value in the indexed attribute, and the list of UIDs. The key-directory consists of the number of classes

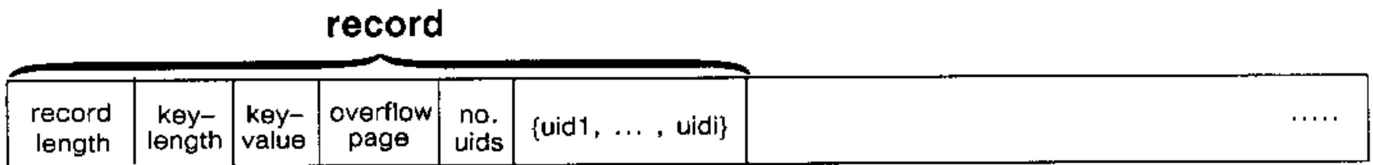

Figure 4a. A Leaf Node of a Single-Class Index

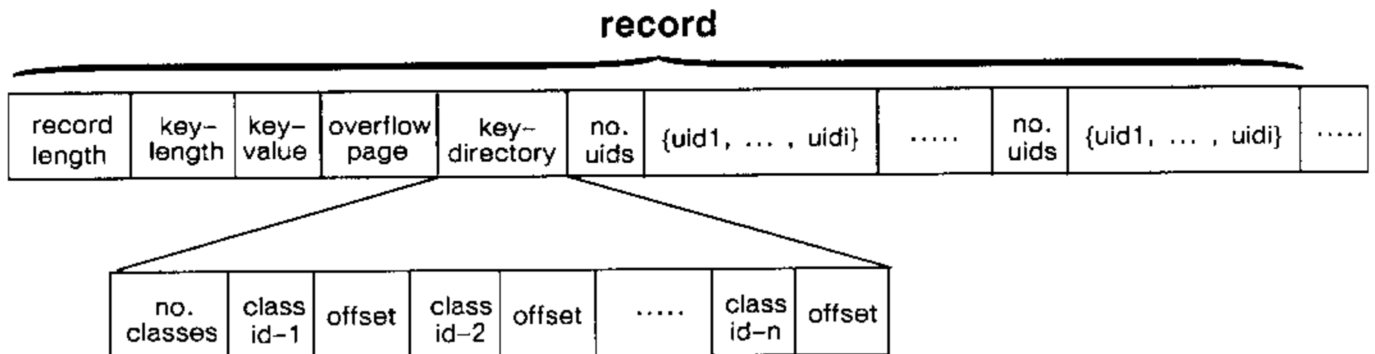

Figure 4b. A Leaf Node of a Class-Hierarchy Index

shown in Figures 4a and 4b. An index record in a leaf node of a single-class index consists of the record-length, key-length, key-value, overflow-page pointer, the number of elements in the list of UIDs of the objects which hold the key-value in the indexed attribute, and the list of UIDs.

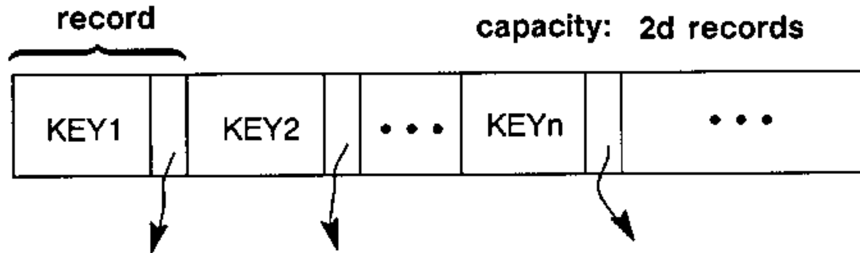

Figure 3. A Non-Leaf Node

An index record in a leaf node of a class-hierarchy index consists of the record-length, key-length, key-value, overflow-page pointer, key-directory, and, for each class in the class hierarchy, the number of elements in the list of UIDs for the objects which hold the key-value in the indexed attribute, and the list of UIDs. The key-directory consists of the number of classes

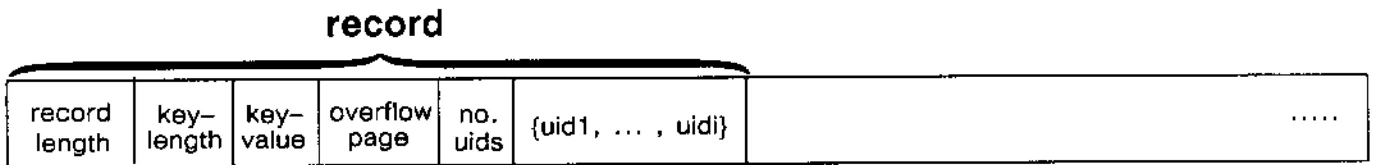

Figure 4a. A Leaf Node of a Single-Class Index

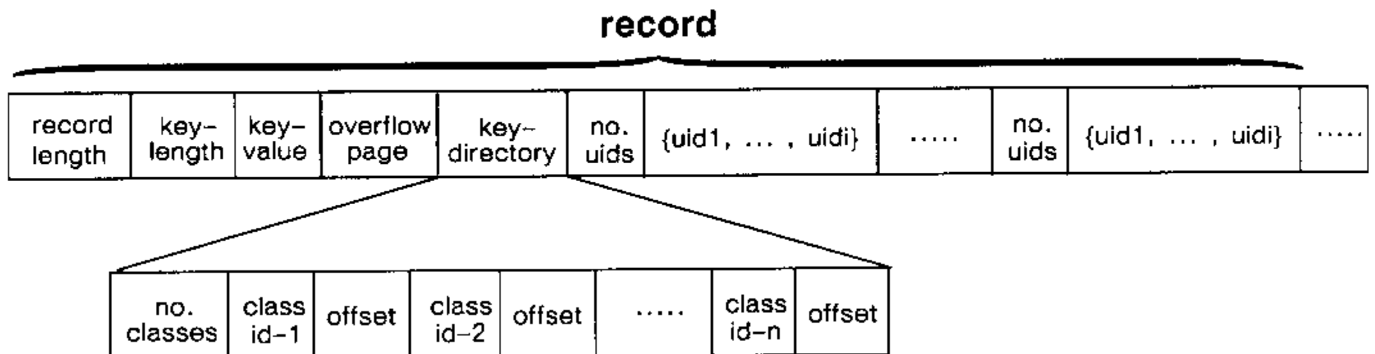

Figure 4b. A Leaf Node of a Class-Hierarchy Index

which contain objects with the key-value in the indexed attribute, and, for each such class, the class identifier and the offset in the index record which hold the list of UIDs of the objects. The leaf node of a class-hierarchy index groups the list of UIDs for a key-value in terms of the classes to which they belong.

The rationale for this organization is that a class-hierarchy index is maintained on an attribute for a class hierarchy consisting of n classes rooted at a class C, and that the index may often need to be used for a query which is directed to a subclass of the class C. If the leaf node is organized as in a single-class index, an exhaustive scan of the entire list of UIDs for a key-value is necessary to screen the UIDs which do not belong to the classes relevant to a query. Further, if a class in the class hierarchy is dropped, the UIDs of instances for the class must be deleted from the class-hierarchy index; and the organization shown in Figure 4b facilitates deletion of a list of UIDs for any class on a class hierarchy.

A leaf-node index record may be *small* (not larger than the size of an index page) or *large* (larger than the index-page size). A small index record can grow to a large index record or simply grow out of bounds of its current index page. There are a number of ways to deal with these *leaf-node overflow* situations. The approach we have adopted for purposes of the present study is as follows. On one hand, if a small index record grows out of bounds of its index page, but remains a small record, the index page is split. On the other hand, if an index record becomes a large record, an entire leaf node is assigned to it, and the part of the record which still does not fit in the node is stored in overflow page(s). This is the use of the overflow-page pointer field in a leaf-node index record; if the value of this field is zero, the index record can be presumed to be fully contained in the current index page.

5. SIMULATION EXPERIMENTS

The cost model we developed for our simulation experiments uses a number of parameters. The following set of parameters captures the characteristics of the database.

Di : number of distinct keys for an attribute of a class Ci

Ni : cardinality of a class Ci (number of instances of Ci)

Ki : average number of UIDs per key in an attribute of a class Ci ($Ki = \lceil Ni / Di \rceil$)

The next set of parameters represents the characteristics of the index nodes. We note that the first four parameters apply to both a single-class index and a class-hierarchy index

d: order of a non-leaf node (minimum number of non-leaf node index records)

f: average fanout from a non-leaf index node
(d <= f <= 2d, for any non-leaf node other than the root node)
(2 <= f <= 2d, for the root node)

P: size of an index page in bytes

kl: average length of a key value for an indexed attribute

L: average length of a non-leaf node index record in bytes

XS: average length of a leaf-node index record in bytes for a single-class index

XC: average length of a leaf-node index record in bytes
for a class-hierarchy index

c: average number of classes for a key in a leaf node index record
for a class-hierarchy index

cn: total number of classes indexed by a class hierarchy index

Most of the parameters in our cost model can affect in varying degrees the size and performance of an index. To keep the cost model and our experiments tractable and to allow a fair comparison of the two types of indexing techniques, we have made the following assumptions.

1. All key values have the same length. This implies that all non-leaf node index records in both a single-class index and a class-hierarchy index have the same length.

2. In a class-hierarchy index, the number of classes containing instances with the indexed key value is the same for each key value; that is, the number-of-classes field has the same value in each leaf-node index record in a class-hierarchy index.

3. The key values of an attribute are uniformly distributed among instances of a class. This, along with assumption 1, implies that all leaf node index records in a given single class index have the same length. Further, this assumption, along with assumptions 1 and 2, implies that all leaf-node index records in a given class-hierarchy index have the same length. In particular,

in any given index, leaf-node index records are either all *small* (not larger than the size of an index page) or all *large* (larger than the index-page size).

Without this assumption, we have the unenviable task of having to incorporate into the cost model the fact that each leaf-node index record has a different size. This assumption may not be realistic; however, most of the simulation studies of the performance of database systems have made the same assumption, and the results of such studies nonetheless have proven to be useful.

We note that in the next sections we provide separate cost formulations for the two sizes of leaf-node index records. The separate treatment was deemed necessary to account for the overflow pages for the large index records.

4. The root class C of a class hierarchy for which a class-hierarchy index is maintained is also the class against which a query is directed. This assumption is necessary to allow a fair comparison between single-class indexing and class-hierarchy indexing. In general, a query may be directed against a descendant class of C, and the class-hierarchy index on C may have to be used to evaluate the query. Then only a part of the index contains entries which correspond to the classes relevant to the query. The choice of a class for which a class-hierarchy index is maintained must be carefully made, possibly with computerized physical database design tools.

5. Each non-leaf (and non-root) index node has the same fanout f, both in a single-class index and a class-hierarchy index. This assumption is also necessary to allow a fair comparison of the two types of indexing techniques.

6. The cardinality of a class in a class hierarchy is independent of the cardinality of any of its superclass or subclass; that is, there is no correlation between the number of instances of a class and that of any other class on the same class hierarchy. When a class S is created as a subclass of a class C, a subset of the instances of C may migrate to S. If the class C has not been partitioned into a sufficient number of subclasses, C may have more instances than any of its subclasses. However, a class C may sometimes be an abstract class and have no instances associated with it, while its subclass S may have many instances.

The distribution of the key values across the classes of a class hierarchy has significant impacts on the tradeoffs between single-class indexing and class-hierarchy indexing. For

example, if the key value of an indexed attribute is confined to instances of only one class C, a class-hierarchy index may be less efficient than a single-class index on the class C. However, if the key value is contained in instances of all the classes in the class hierarchy, traversing a class-hierarchy index may be more efficient than traversing a single-class index on each of the classes in the class hierarchy.

In general, each class of a class hierarchy may contain a unique value for the key. There are two extreme cases for the distribution of key values: disjoint distribution and inclusive distribution. In a *disjoint distribution*, each key value of an indexed attribute is found in only one class; that is, all values of the attribute are unique to each class of the class hierarchy. In a disjoint distribution, each class in a class hierarchy has unique values on the indexed attribute. The total number of unique key values is

SUM OF (D1, D2, ..., Di) for classes C1, C2, ..., Ci

In an *inclusive distribution*, one class of the class hierarchy has all key values for an indexed attribute. The total number of unique key values with an inclusive distribution is

MAX (D1, D2, ..., Di) for classes C1, C2, ..., Ci

Obviously, these distributions represent two extreme cases. Nevertheless, they represent the best or worst cases for the indexing techniques with which we are concerned. Further, a realistic distribution of key values will be somewhere between these extremes. Therefore, we have made some efforts to analyze the behavior of these two extreme distributions of key values.

To simplify the presentation of our cost model, we have used explicit figures in the expression of some of the parameters, as follows. These figures are based largely on the B+-tree index implementation in ORION.

1. For the key-length and next-level-page pointer fields in a non-leaf node index record, we use 2 and 4 bytes, respectively. The record-length, number of UIDs, and overflow-page pointer fields in a leaf-node index record take up 2, 2, and 4 bytes, respectively. The number-of-classes field in the key-directory in a leaf-node record of a class-hierarchy index needs 2 bytes, and each offset field takes up 2 bytes. The class id requires 4 bytes.

2. The index page size used was 4K (4096) bytes. Further, we will assume that average length of a key value is equal to the size of a UID. The length of a UID is 8 bytes. This in turn means that $L = 14$, $XS = Ki * 8 + 18$, and $XC = Ki * 8 + c * 10 + 16$ bytes.

3. For the value of f we used 218, assuming a UID size of 8 bytes and the block size of 4k bytes. The order d of a B-tree is P / L = 146.

The following summarize the organization of the simulation experiments we have conducted. A detailed description of it is provided in the Appendix.

1. We constructed 9 experiments. Within each experiment, we made 20 simulation runs by varying the parameters Ni and Di, as well as cn. The total number of simulation runs we made was thus 180.

2. The number of classes in the class hierarchy was varied between 2 and 6.

3. In the first 5 experiments, we used the values for Ni between 20,000 and 200,000; and in the last 4 experiments, Ni was varied between 50,000 and 255,000.

4. Di ranged, for the 9 experiments, from 100 to 400 (so that Ki was varied from 200 to 500), 40 to 1000 (for Ki between 500 and 200), 100 to 1000 (for Ki to be about 200), 40 to 400 (for Ki to be about 500), 60 to 575 (for Ki to be about 350), 350 and 600 (for Ki between 150 and 500), 500 and 1300 (for Ki to be around 200), 100 to 600 (for Ki to be about 500), and 150 to 750 (for Ki to be about 350).

6. INDEX SIZES

To compare the secondary storage requirements of a class-hierarchy index on a class hierarchy and a corresponding set of single-class indexes, we have formulated a cost model for the size of an index. The size of an index is the total number of index nodes, where each node occupies a physical page on secondary storage. In this section, we will present our cost model for the size of both a single-class index and a class-hierarchy index, and then present the results of our simulation experiments.

6.1 Cost Model for the Size of an Index

[KNUT73] derives bounds for the height of a B-tree and the number of nodes in each level of the B-tree, given the order and the number of keys. [COME79] also provides asymptotic bounds for the height and number of nodes in each level of a B-tree using a slightly different definition. We have derived our own formulas largely to reflect the implementation details of a B-tree index in a database system. Further, the variation of the B-tree we are considering is somewhat different from that used in [KNUT73] and [COME79].

The cost model formulated below uses the parameters we defined in Section 5. We provide separate sets of formulas for a single-class index and a class-hierarchy index. Further, for each type index, we provide separate formulas for small and large leaf-node index records. We also introduce the following additional symbols:

- L0: number of leaf-level index pages, including overflow pages
- L1: number of leaf-level index pages, excluding overflow pages
- NL: number of non-leaf level index pages

(1) Single-Class Index

small leaf-node index records (XS <= P)

F1: number of leaf pages, $L0 = \lceil Di / \lfloor P / XS \rfloor \rceil$

F2: number of non-leaf pages, $NL = \lfloor L0 / f \rfloor + \lfloor \lfloor L0 / f \rfloor / f \rfloor + \ldots + X$
where each term is successively divided by f until the last term X is less than f. If the last term X is not 1, then 1 is added to the total (for the root node).

Example 1: Let Di = 100, Ni = 20000, and f = 5. Then Ki = 20000/100 = 200. The number of records in a leaf page is 2. Thus L0 = 100/2 = 50. NL = 50/5 + (50/5)/5 + 1 = 13.

large leaf-node index records (XS > P)

F3: number of leaf pages, excluding the overflow pages, $L1 = Di$

F4: number of non-leaf pages, $NL = \lfloor L1 / f \rfloor + \lfloor \lfloor L1 / f \rfloor / f \rfloor + \ldots + X$
where F4 is defined similarly as in F2

F5: number of leaf pages, including the overflow pages, $L0 = L1 * \lceil XS / P \rceil$

Example 2: Let Di = 50, Ni = 30000 and f = 5. Then Ki = 30000/50 = 600. L1 = 50. NL = 50/5 + (50/5)/5 + 1 = 13. L0 = 50 * 2 = 100.

F6: total number of index pages for a class Ci = $L0 + NL$

(2) Class-Hierarchy Index

small leaf-node index records (XC <= P)

F7: number of leaf pages, $L0 = \lceil Di / \lfloor P / XC \rfloor \rceil$

F8: number of non-leaf pages, $NL = \lfloor L0 / f \rfloor + \lfloor \lfloor L0 / f \rfloor / f \rfloor + \ldots + X$

where each term is successively divided by f until the last term X is less than f. If the last term X is not 1, then 1 is added to the total, as in F2.

Example 3: Let Di = 220, Ni = 50000, c=3, and f = 5. Then Ki = 50000/220 = 228. The number of records in a leaf page is 2. Thus L0 = 220/2 = 110. NL = 110/5 + (110/5)/5 + 1 = 27.

large leaf-node index records (XC > P)

F9: number of leaf pages excluding overflow pages, $L1 = Di$

F10: number of non-leaf pages, $NL = \lfloor L1 / f \rfloor + \lfloor \lfloor L1 / f \rfloor / f \rfloor + \ldots + X$
where F10 is defined similarly as in F8

F11: number of leaf pages including overflow, $L0 = L1 * \lceil XC / P \rceil$

Example 4: Let Di = 200, Ni = 110000, c=3, and f = 5. Then Ki = 110000/200 = 550. L1 = 200. NL = 200/5 + (200/5)/5 + ((200/5)/5)/5 = 49. L0 = 200.

F12: total number of class-hierarchy index pages = $L0 + NL$

6.2 Results of Experiments

In this section we present and analyze the results of our experiments with the cost model presented in Section 6.1. We implemented a simulation program and made a large number of simulation runs by varying the parameters of the cost model.

Figure 5 shows the results of one of our nine experiments, in which Ni varied between 50,000 and 255,000, Di between 500 to 1300, and Ki about 200. The details of this experiment is described in Table 7 in the Appendix. The figure shows the number of index pages created for a class-hierarchy index and single-class indexes; for class-hierarchy indexing, we show the

results for 3 different distributions of key values: inclusive, disjoint, and mixed. The index size of a class-hierarchy index, for any distribution, is sometimes larger and sometimes smaller than the sum of the sizes of the corresponding set of single-class indexes. For different distributions of key values, the size of a class-hierarchy index may be larger than the sum of the sizes of the corresponding single-class indexes, largely because of the overflow pages created as the leaf-node index records become large.

The number of index pages created for a class-hierarchy index ranges from a 37% decrease to a 50% increase, when compared to single-class indexing. With an inclusive distribution, the range is from a decrease of 27% to a 48% increase, relative to single-class indexing. With a disjoint distribution, the range is from a 16% decrease to an increase of 15% compared to single-class indexing.

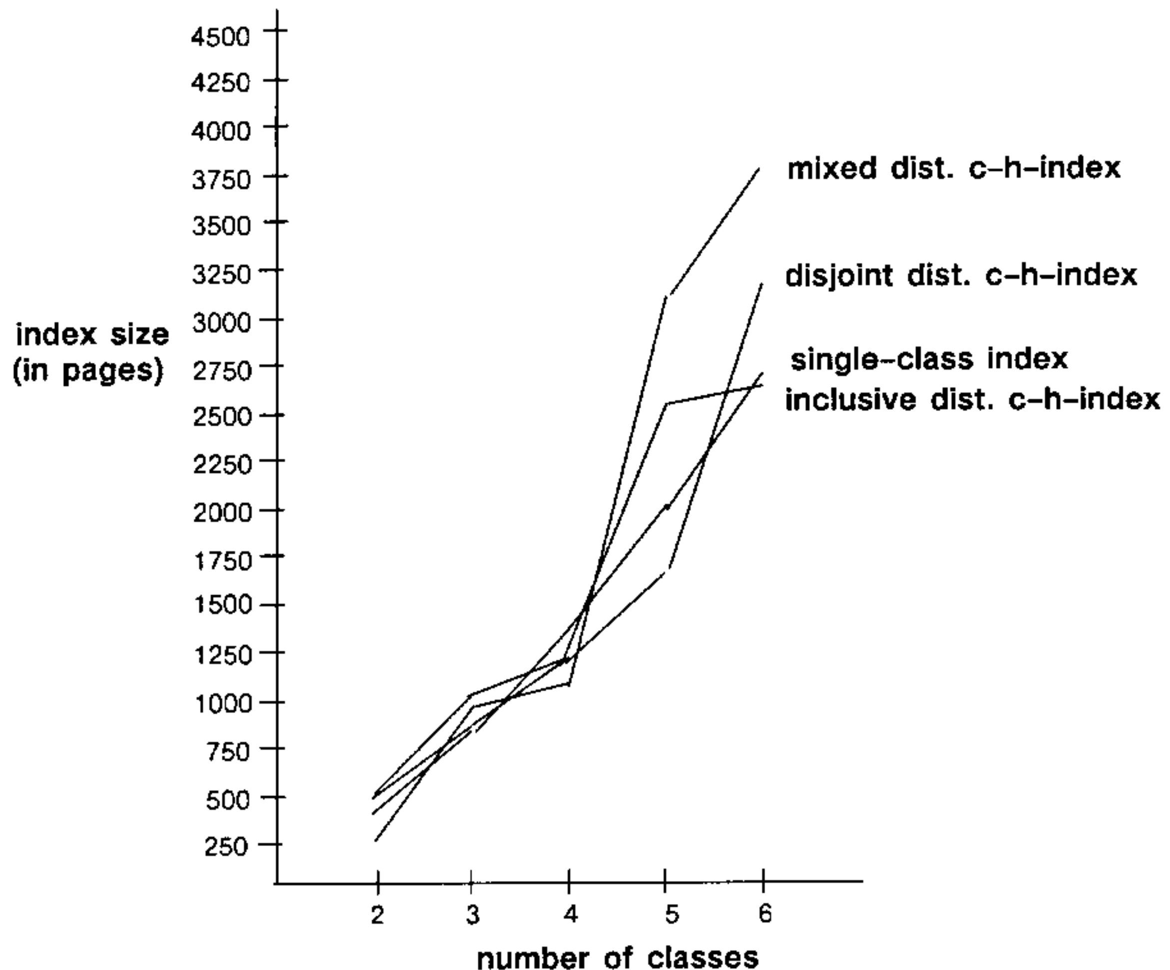

Figure 5. Comparison of Index Sizes

7. PERFORMANCE

In this section, we first formulate a cost model for the performance of a single-class index and a class-hierarchy index. We then present the results of performance comparison between the two types of indexes. The performance comparison was conducted for two types of queries: single-key queries and range queries. For either type of query, we computed the number of index pages which need to be fetched to evaluate a given query. A *single-key query* is one in which the search condition consists of a single predicate of the form (key = value). A *range query* is one in which the predicate is of the form (key < value), (key > value), or (key between value-1 and value-2). Since our results for a query with a single predicate readily generalize to a query involving a conjunction (predicate AND predicate AND ...) or a disjunction (predicate OR predicate OR ...) of single predicates, we do not consider the latter explicitly.

7.1 Single-Key Query Evaluation

7.1.1 Cost Model

The number of index pages fetched to evaluate a query is precisely the height of the index used. To compute the height of an index, a formula similar to F2 or F4 used to calculate the number of non-leaf pages can be used.

F13: height of an index = number of terms in $(L0 + \lfloor L0 / f \rfloor + \lfloor \lfloor L0 / f \rfloor / f \rfloor + \ldots + X)$

where a previous term is successively divided by f. L0 is the number of leaf pages. The division by f stops when the value of the last term X is less than f. If X is not 1, add 1 to the height. If the index contains overflow pages, average number of overflow pages per leaf page needs to be added to the height. The formula is used for both single-class indexes and a class-hierarchy index.

Example 5: Let Di = 100, Ni = 20000, and f = 5. Then Ki = 20000/100 = 200. Using F1, L0 = 100/2 = 50. The number of terms in (50 + 50/5 + (50/5)/5) is 3. Since the last term is not 1, and there are no overflow pages, the height of the index is 4.

7.1.2 Simulation Experiments

In Figure 6, we observe that irrespective of key distributions and the number of classes in the class-hierarchy, the number of index pages fetched with a class-hierarchy index is always

equal or smaller than in single-class index if there are at least two classes in a class-hierarchy. Figure 6 summarizes the results of our experiment described in Table 7 in the Appendix.

In the case of single-class indexing, with an inclusive distribution of the key values among all classes in the indexed class hierarchy, the height of a single-class index must be traversed for each of the classes. With a disjoint distribution, only those single-class indexes need to be traversed which contain the desired key value. In all other indexes, only the root page needs to be accessed.

In the case of class-hierarchy indexing, the number of index pages that need to be accessed for a single-key query is equal to the height of the index, regardless of how the key values are distributed. However, the values of parameters Di, Ni, Ki, and XC cause the height of the index to be different for inclusive and disjoint distributions.

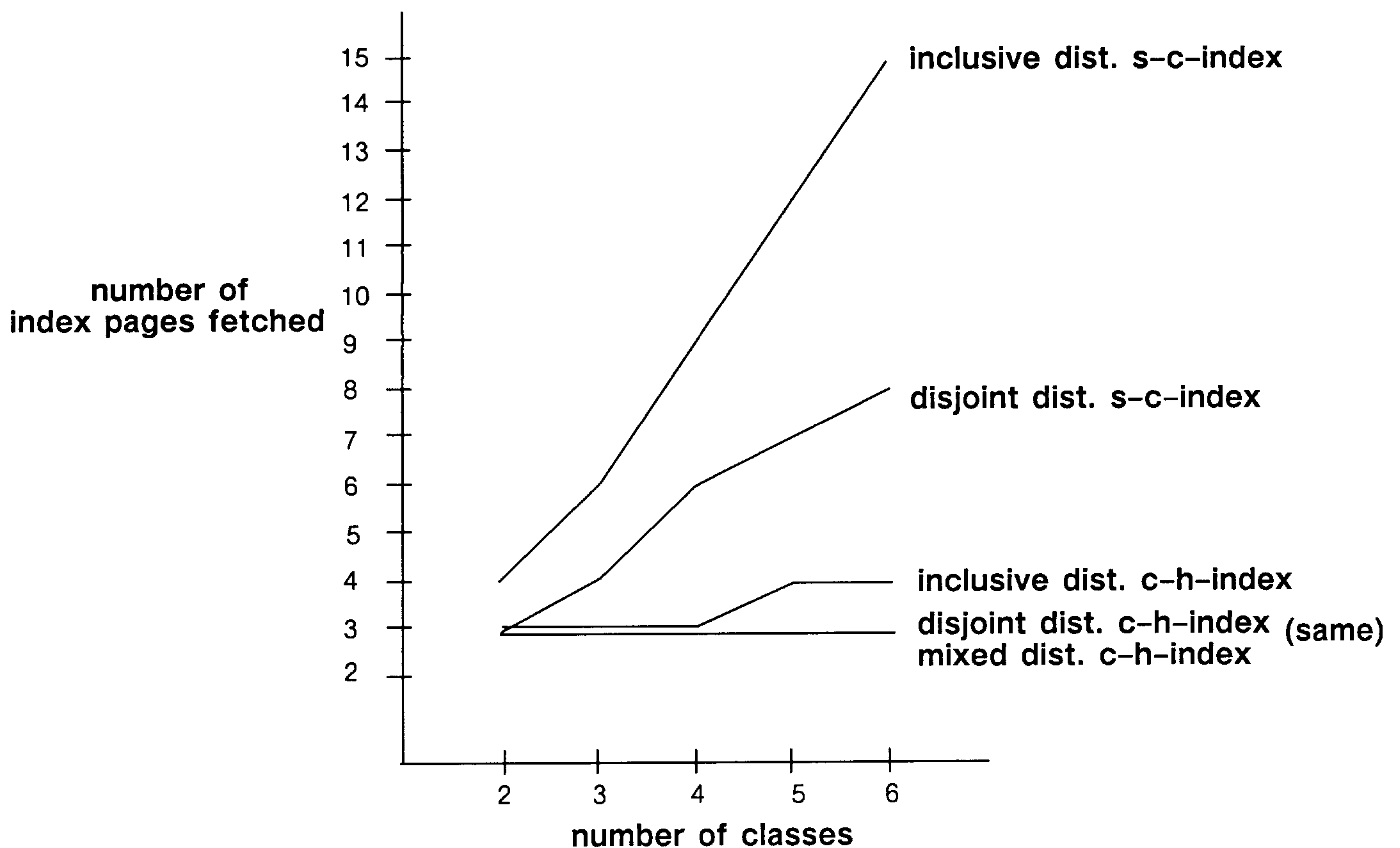

Figure 6. Comparison of Single-Key Query Performance

For a class-hierarchy index, with an inclusive distribution of key values, the reduction in the number of index pages fetched is between 25% to 73% over single-class indexing. The reduction is between 0% to 63% for a disjoint distribution. For a mixed distribution, the reduction is between 25% and 80% over single-class indexing with an inclusive distribution, and between 0% and 63% over single-class indexing with a disjoint distribution.

7.2 Range Query Evaluation

7.2.1 Cost Model

An important consideration in range query evaluation is the distribution of the key values in the range among the classes in a class hierarchy. In the case of an inclusive distribution, the range of key values is obviously confined to one class. This means that, in the case of single-class indexing, we have to fully traverse an index for each class on a class hierarchy. With a disjoint distribution, the range values are scattered among all classes of a class hierarchy. In the case of single-class indexing, only the indexes for those classes in the class hierarchy that contain a value in the range need to be fully traversed; only the root pages need to be accessed for all other indexes on the classes.

Before we derive the formula for the number of index pages fetched, we need to introduce symbols UKi and NRQ, as follows.

UKi: *number of unique keys in a leaf-level index node*

NRQ: *number of key values in the range specified for a given query*

In a single-class index,

$UKi = \lfloor P / XS \rfloor$

for small leaf-node index records (XS <= P), and

$UKi = 1$

for large leaf-node index records (XS > P).

In a class-hierarchy index,

$UKi = \lfloor P / XC \rfloor$

for small leaf-node index records (XC <= P), and

UKi = 1

for large leaf-node index records (XC > P).

We now derive formulas for the number of index pages to be fetched. We need the following additional symbols:

RL:	number of leaf pages fetched, excluding overflow pages
nRL:	sum of leaf pages fetched, excluding overflow pages, from n single-class indexes
SRL:	sum of leaf pages fetched, including overflow pages, from n single-class indexes
SRN:	sum of non-leaf pages fetched from n single-class indexes
IP1:	total number of index pages fetched, excluding overflow nodes
IP2:	total number of index pages fetched, including overflow nodes

(1) range values in one class

First, we consider the case when all range values are in one class. As mentioned earlier, indexes on the attribute for all classes in a class hierarchy have to be searched for an inclusive distribution of the key values; while, for a disjoint distribution, only those indexes of the classes which contain any value in the range need to be fully traversed.

If NRQ = 1, that is, in the limiting case where the range consists of only one key value,

RL = 1

F14: If UKi = 1, RL = NRQ

Example 6: For a single-class index, let Di = 50, Ni = 30000, and f = 5. Ki = 30000/50 = 600. Since Ki*8+18 > 4096, UKi = 1. If we have NRQ = 20, RL is 20. Since UKi, the number of different keys in a leaf page, is 1, the number of leaf pages that needs to be fetched is NRQ.

F15: If UKi > 1, $RL = \lfloor NRQ / UKi \rfloor + 1$*, if n is 0 or 1*

$RL = \lfloor NRQ / UKi \rfloor + 2$*, if n > 1,*

where n = mod (NRQ, UKi)

Example 7: For a class-hierarchy index, let Di = 220, Ni = 50000, c = 3, and f = 5. Ki = 50000/220 = 228. Since Ki*8+46 < 4096, UKi = 4096/(228*8+42) = 2. If NRQ = 15, n = 1 and thus RL = 15/2 + 1 = 8 pages.

F16: If XS <= P for a single-class index, or if XC <= P for a class-hierarchy index, maximum number of index pages fetched,

$$IP1 = RL + (\lfloor RL / f \rfloor + X) + (\lfloor \lfloor RL / f \rfloor + X / f \rfloor + X) + \ldots + 1$$

where the total number of additions of the terms in the formula is equal to the height of the index for the class. X is 1 if the remainder of division by f in the term is either 0 or 1; otherwise X is 2.

Example 8: For a single-class index, let Di = 100, Ni = 20000, and f = 5. Then Ki = 200. Since Ki*8+18 <= 4096, UKi = 4096/(200*8+18) = 2. If NRQ = 10, RL = 6. The height of the index with the above parameter values is 4. Thus IP1 = 6 + (6/5+1) + ((6/5+1)/5 +2) + 1 = 6 + 2 + 2 + 1 = 11.

F17: For large leaf-node index records (XS > P) in a single-class index,

$$IP2 = IP1 + RL * \lfloor XS / P \rfloor$$

where RL and IP1 have been defined in F14 (or F15), and F16, respectively.

F18: For large leaf-node index records (XC > P), in a class-hierarchy index,

$$IP2 = IP1 + RL * \lfloor XC / P \rfloor$$

where RL and IP1 have been defined in F14 (or F15) and F16, respectively.

Example 9: For a class-hierarchy index, let Di = 200, Ni = 110000, c = 5, and f = 5. Then Ki = 550. Since Ki*8+66 > 4096, UKi = 1.
If NRQ = 10, RL = 10. Since the height of the index is 4, IP1 = 10 + (10/5+1) + (3/5+2) + 1 = 16 pages. IP2 = 16 + (10*((550*8+66)/4096)) = 26 pages.

(2) range values scattered among more than one class

For single-class indexing, if the key values in a given range are scattered among more than one class, each of the indexes of the classes whose instances contain any of the values in the range needs to be fully traversed. SRL, the number of leaf pages to be fetched, is the sum of the RL's, including overflow pages, for all indexes that must be fully searched. To compute SRL, we can make use of the formulas for RL, namely, F14 and F15.

F19: $SRL = \text{sum of } RL * \lceil XS / P \rceil$ *of each index.*

In computing SRN, the nRL used is the sum of the RL's for all indexes which must be fully traversed.

F20: $SRN = (\lfloor nRL / f \rfloor + X) + \ldots + Vi + Vj + \ldots + Wi + Wj + \ldots + Xi$

Each term (except the first) is successively divided by f, and then 1 or 2 is added to the result, as in F16, except for the terms whose position in the formula is the same as the heights of their respective indexes, such as Vj and Wj. Let Vi be the term produced when the number of additions of terms is equal to the (height−1) of one of the index. Vj is then (((Vi−1) / f) + X). The next term to be added after Vj is then ((Vj / f) + X). The addition of terms continues until the total number of additions of terms in the formula is next equal to the (height−1) of another index involved. Let that term be Wi. The value of Wj is then (((Wi−1) / f) + X). This addition of terms continues until the total number of additions of terms equals (height−1) of the largest index. Hence Xi is the last term added when (height−1) of the largest index is reached. If two indexes have the same height and both are the largest index, the terms are added until the number of additions is equal to (height−1) of any one of them.

F21: maximum number of index pages fetched = SRL + SRN

Example 10: For a single-class index, let Di = 50, Ni = 10000, and f = 5. Then Ki = 200. RL = 6, if NRQ = 10. The height of this index is 3. Assume that there is another index with Di = 120, Ni = 30000 and f = 5. The Ki for the second index is 250 and RL = 6, if NRQ = 10. The height of this index is 4. SRL = 12. The new RL = 6 + 6 = 12. SRN = (12/5+2) + (4/5+2) + ((2−1)/5+1) = 4 + 2 + 1 = 7 pages. Thus if the range values are scattered over two classes as defined above, the total number of index pages is 12 + 7 = 19.

When comparing the number of index pages fetched between single-class indexing and class-hierarchy indexing for a range query, the following observations can be made. In single-class indexing, with an inclusive distribution of key values, indexes for all classes in the class hierarchy must be searched. For each class, the number of index pages that need to be fetched is given in formula F16 or F17. With a disjoint distribution, indexes for only those classes containing instances with one of the values in the range are searched. When the key values in the range are all in one class, the number of index pages to be fetched is given in formula F16 or F17. When the values are scattered over several classes, the maximum number of index pages fetched is given in formula F21.

In class-hierarchy indexing, since we have a single index, the situation is identical to the case where all range values are in one class. The number of index pages to be fetched, given in

formula F16 or F18, is not affected by the type of key-value distribution. It is affected only by the values of the parameters Di, Ni, Ki and UKi.

7.2.2 Simulation Experiments

We studied two different cases, both for the same set of range values which we varied for each experiment. In one case, all the values are assumed to be in any one of the classes in the class hierarchy. Figure 7, which summarizes the results of our experiment described in Table 7 in the Appendix, shows that, when the number of values in the range is 20, class-hierarchy indexing requires fewer index pages to be fetched than single-class indexing for the same key-value distribution, if there are at least two classes in the class hierarchy. In class-hierarchy indexing, disjoint distribution generally requires fewer index pages to be fetched than is the case with an inclusive key distribution. One reason for this is because of the overflow nodes being created in an inclusive distribution. Another reason is that the value of UKi is normally greater in a disjoint distribution than in an inclusive distribution; this results in more keys being packed per leaf page, and as a consequence in fewer leaf pages to be fetched for a fixed number of range values.

In another case, the range key values are scattered evenly in two classes, that is, each class contains half the key values. Figure 8, which once again represents the experiment described in Table 7, shows that the result is similar to the first case, for 20 key values in the range. We only consider a disjoint distribution, since inclusive distribution is meaningless. Sometimes, single-class indexing requires fewer index pages to be fetched than class-hierarchy indexing. This is because, in a disjoint distribution of key values, only the indexes for the classes containing the range values need to be searched, and the size of the individual single-class index is smaller than the class-hierarchy index.

We have also compared the maximum number of index pages fetched for class-hierarchy indexing and single-class indexing. Class-hierarchy indexing results in a reduction of between 5% and 39% in the number of index pages fetched compared to single-class indexing, in the case of an inclusive distribution. For a disjoint distribution, the range is from a reduction of 38% to an increase of 10%. In the case of a mixed distribution, the range is from a reduction of 20% to an increase of 28% over single-class indexing with a disjoint distribution. The reduction is between 40% and 65% compared to single-class indexing with an inclusive distribution of key values among classes in the class hierarchy.

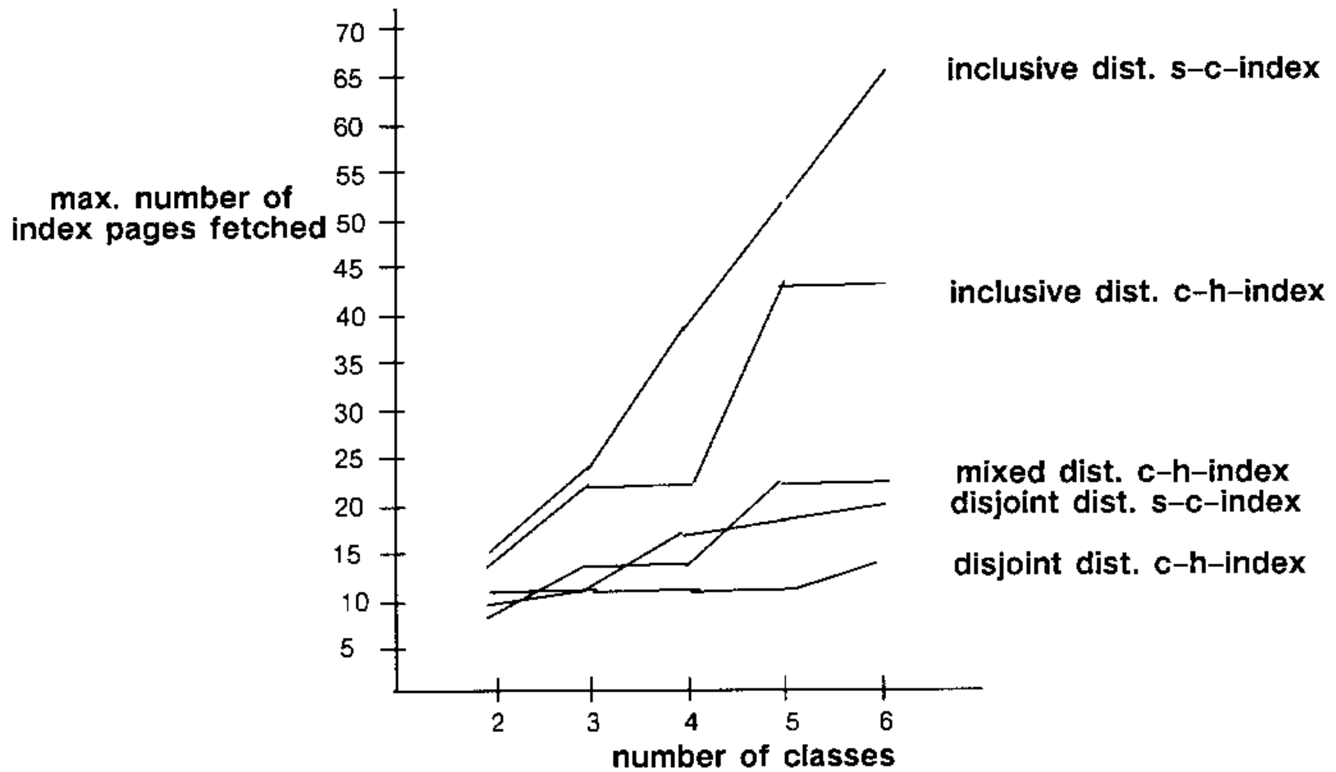

Figure 7. Comparison of Range-Query Evaluation (key values distributed in one class)

For a range query in which the key values are scattered over two classes, we only consider disjoint and mixed key distribution because inclusive key distribution is not meaningful for this scenario. The maximum number of index pages fetched are as follows for class-hierarchy index and single-class index. For disjoint key distribution, the reduction in the number of index pages fetched is 42% and the increase is 20% for class-hierarchy indexing over single-class indexing. For mixed key distribution class-hierarchy index, the range is between a reduction of 8% to an increase of 27% over disjoint key distribution single-class index.

We note that a disjoint distribution represents the best case for single-class indexing, in terms of the number of index pages that needs to be fetched, for both single-key query and range-query evaluation. Only indexes for the classes containing the key values need to be traversed, and as such only the single-class indexes for the classes need to be fetched.

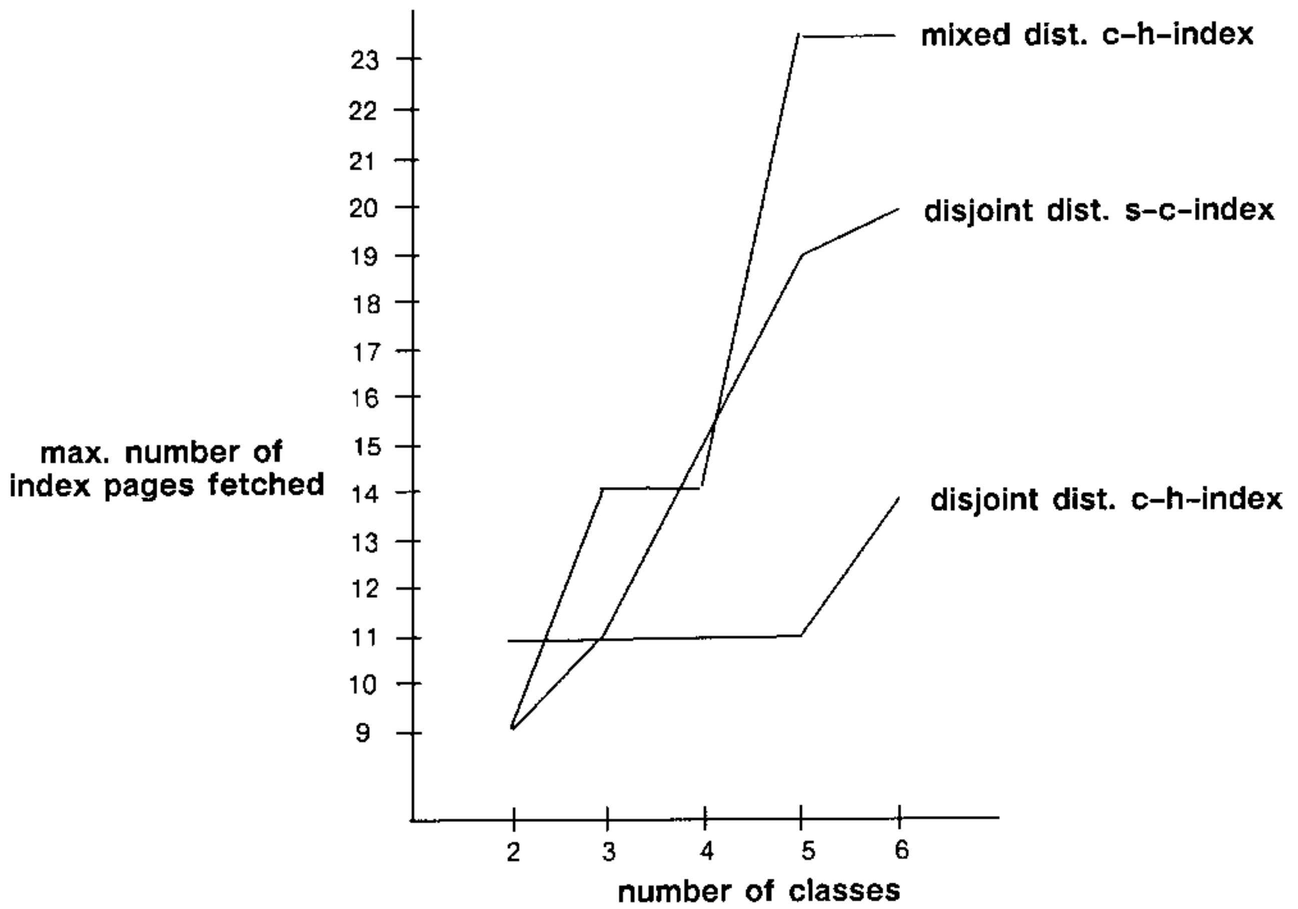

Figure 8. Comparison of Range-Query Evaluation (key values split between two classes)

8. SUMMARY

In an object-oriented database, the scope of a query is often a class-hierarchy rooted at a particular class. This means that to support the evaluation of a query whose access scope is a class hierarchy, the system must maintain one index for each of the classes involved in the query. A single-class index is the traditional index which is maintained on an attribute of a single class. A class-hierarchy index is one which may be maintained on an attribute of all classes on a class hierarchy.

In this paper, we formulated the cost model for the size and performance of a single-class index and a class-hierarchy index. We then presented the results of simulation experiments we have conducted to quantify the tradeoffs between the two types of index. The preliminary results show that a class-hierarchy index tends to be more efficient than a single-class index in terms of the number of index pages that need to be fetched for a given query, as long as there are at least two classes in a class hierarchy. As for the index size, a class-hierarchy index may be smaller or larger than the sum of the corresponding single-class indexes. For the index structure that we modeled, the index size depends on how many overflow nodes are created and how efficiently each leaf-level node is utilized.

ACKNOWLEDGMENTS

We are grateful to Jorge Garza for a discussion of the implementation details of an index in a database system, and to Hong-Tai Chou for comments on the assumptions we have made for our cost formulation and simulation experiments. We thank Linda Lamphear for preparing the Figures and Tables for us.

REFERENCES

[AFSA86] Afsarmanesh, H., D. Knapp, D. McLeod, and A. Parker. "An Object-Oriented Approach to VLSI/CAD," in *Proc. Intl Conf. on Very Large Data Bases*, August 1985, Stockholm, Sweden.

[AHLS84] Ahlsen M., A. Bjornerstedt, S. Britts, C. Hulten, and L. Soderlund. "An Architecture for Object Management in OIS," *ACM Trans. on Office Information Systems*, vol. 2, no. 3, July 1984, pp. 173-196.

[BANE87a] Banerjee, J., et al. "Data Model Issues for Object-Oriented Applications," *ACM Trans. on Office Information Systems*, Jan. 1987.

[BANE87b] Banerjee, J., W. Kim, H.J. Kim, and H.F. Korth. "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," to appear in *Proc. ACM SIGMOD Conf. on the Management of Data*, San Francisco, Calif., May 1987.

[BLAS77] Blasgen, M.W. & Eswaran, K.P., "Storage and access in relational data bases," *IBM Syst. Journal*, No.4, 1977.

[BOBR83] Bobrow, D.G., and M. Stefik. *The LOOPS Manual*, Xerox PARC, Palo Alto, CA., 1983.

[BOBR85] Bobrow, D.G., K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. *CommonLoops: Merging Common Lisp and Object-Oriented Programming*, Intelligent Systems Laboratory Series ISL-85-8, Xerox PARC, Palo Alto, CA., 1985.

[CHAN82] Chan, A. et. al., "Storage and access structures to support a semantic data model", in *Proc. Intl Conference on VLDB*, Sept. 1982.

[COME79] Comer, D. "The Ubiquitous B-Tree," *Computing Surveys*, Vol. 11 No. 2, June 1979.

[CURR84] Curry, G.A. and R.M. Ayers. "Experience with Traits in the Xerox Star Workstation," *IEEE Trans. on Software Engineering*, vol. SE-10, no. 5, September 1984, pp. 519-527.

[DATE83] Date, C.J. *An Introduction to Database Systems, Vol. 2*, Addison-Wesley, 1983.

[GOLD81] Goldberg, A. "Introducing the Smalltalk-80 System," *Byte*, vol. 6, no. 8, August 1981, pp. 14-26.

[GOLD83] Goldberg, A., and D. Robson. *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983

[IBM81] SQL/Data System: Concepts and Facilities. GH24-5013-0, File No. S370-50, IBM Corporation, Jan. 1981.

[IEEE85] *Database Engineering*, IEEE Computer Society, vol. 8, no. 4, December 1985 special issue on Object-Oriented Systems (edited by F. Lochovsky).

[KNUT73] Knuth, D. *The Art of Computer Programming, Vol. 3*, Addison-Wesley, Reading, Mass. 1973.

[LMI85] *ObjectLISP User Manual*, LMI, Cambridge, MA, 1985.

[MAIE86] Maier, D., and J. Stein. "Indexing in an Object-Oriented DBMS," Oregon Graduate Center, Technical Report: CS/E-86-006, May 1986.

[SELI79] Selinger, P.G. et. al. "Access Path Selection in a Relational Database Management System," in *Proc. ACM SIGMOD Conf.*, Boston, Mass. pp 23-34, 1979.

[STEE84] Steele, G.L. *Common Lisp: The Language*, Digital Press, 1984.

[STEF86] Stefik, M., and D.G. Bobrow. "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, January 1986, pp. 40-62.

[STON76] Stonebraker, M., E. Wong, P. Kreps, G. Held. "The Design and Implementation of INGRES," *ACM Trans. on Database Systems*, vol. 1, no. 3, Sept. 1976, pp. 189-222.

[SYMB84] *FLAV Objects, Message Passing, and Flavors*, Symbolics, Inc., Cambridge, MA, 1984.

[ULLM82] Ullman, J. *Principles of Database System*, Computer Science Press, Inc. 1982.

[WOEL86] Woelk, D., W. Kim, and W. Luther. "An Object-Oriented Approach to Multimedia Databases," in *Proc. ACM SIGMOD Conf. on the Management of Data*, Washington D.C., May 1986.

[WOEL87] Woelk, D., and W. Kim. "Multimedia Information Management in an Object-Oriented Database System," to appear in *Proc. Very Large Data Bases*, Brighton, England, Sept. 1987.

APPENDIX

Each of the 9 tables included here describes the organization of an experiment we have conducted, along with the results of each of the 20 simulation runs within the experiment. The number of classes in the class hierarchy was varied between 2 and 6 for some fixed values of the parameters Ki and NRQ. Each row in the table shows the performance measure for a specified number of classes in the class hierarchy. It also shows the values of Ni and Di used for each class in the class hierarchy. The columns indicate the key distributions assumed, for both single-class indexing and class-hierarchy indexing.

Concerning the Ni and Di in the table, we note the following. Ni and Di are specified for each class in the class-hierarchy for single-class indexing, while only one pair of Ni and Di values are specified for each distribution of key values for a class-hierarchy index. This is because an index is maintained for each of the classes in the class hierarchy in single-class indexing, while a single index is maintained for all classes in the class hierarchy in class-hierarchy indexing. For a class-hierarchy containing more than 2 classes, we have only specified one pair of Di and Ni values for single-class indexing. The assumption is that it also includes all pairs of Di and Ni values from the previous category.

The category range-key (1) denotes range query evaluation where all range values are in one class, while range-key (2) denotes range query evaluation where range values are scattered in two different classes.

TABLE 1 ⊗ K_i = 200 - 500; NRQ = 1/10 of mixed dist. keys

CLASSES	KEY DISTRIBUTION	SINGLE-CLASS DISJOINT	SINGLE-CLASS INCLUSIVE	DISJOINT CLASS-HIERARCHY	INCLUSIVE CLASS-HIERARCHY	MIXED CLASS-HIERARCHY
2	DKi ; Ni	DK_1 = 100; N_1=20000 DK_2 = 120; N_2=30000		DK_c = 220; N_c= 50000	DK_c = 120; N_c= 50000	DK_c = 170; N_c= 50000
	INDEX SIZE	112		111	121	171
	SINGLE-KEY QUERY	3	4	2	2	2
	RANGE-KEY (1)	11	20	10	18	18
	RANGE-KEY (2)	12		10		18
3	DKi ; Ni	DK_3 = 200; N_3=60000		DK_c = 420; N_c=110000	DK_c = 200; N_c=110000	DK_c = 310; N_c=110000
	INDEX SIZE	313		421	401	311
	SINGLE-KEY QUERY	4	6	2	3	2
	RANGE-KEY (1)	34	66	32	63	32
	RANGE-KEY (2)	27		32		32
4	DKi ; Ni	DK_4 = 250; N_4=100000		DK_c = 670; N_c=210000	DK_c = 250; N_c=210000	DK_c = 460; N_c=210000
	INDEX SIZE	564		674	501	463
	SINGLE-KEY QUERY	5	8	3	3	3
	RANGE-KEY (1)	50	144	49	93	49
	RANGE-KEY (2)	50		49		49
5	DKi ; Ni	DK_5=340; N_5=150000		DK_c=1010; N_c=360000	DK_c = 340; N_c=360000	DK_c = 675; N_c=360000
	INDEX SIZE	905		1015	1021	1354
	SINGLE-KEY QUERY	6	10	3	4	4
	RANGE-KEY (1)	73	279	71	205	139
	RANGE-KEY (2)	73		71		139
6	DKi ; Ni	DK_6=400; N_6=200000		DK_c=1410; N_c=560000	DK_c = 400; N_c=560000	DK_c = 905; N_c=560000
	INDEX SIZE	1306		1417	1201	1815
	SINGLE-KEY QUERY	7	12	3	4	4
	RANGE-KEY (1)	96	458	93	271	183
	RANGE-KEY (2)	96		93		183

TABLE 2.8 K_i 500 - 200; NRQ = 1/50 of inclusive dist. keys

CLASSES	KEY DISTRIBUTION	SINGLE-CLASS DISJOINT	SINGLE-CLASS INCLUSIVE	DISJOINT CLASS-HIERARCHY	INCLUSIVE CLASS-HIERARCHY	MIXED CLASS-HIERARCHY
2	DKi ; Ni	DK_1=40; N_1=2000 DK_2=70; N_2=30000		DK_c= 110; N_c= 50000	DK_c= 70; N_c= 50000	DK_c= 90; N_c= 50000
	INDEX SIZE	112		111	141	181
	SINGLE-KEY QUERY	3	4	2	3	3
	RANGE-KEY (1)	4	6	3	5	5
	RANGE-KEY (2)	4		3		5
3	DKi ; Ni	DK_3= 150; N_3=60000		DK_c= 260; N_c=110000	DK_c= 150; N_c=110000	DK_c= 205; N_c=110000
	INDEX SIZE	263		261	301	411
	SINGLE-KEY QUERY	4	6	2	3	3
	RANGE-KEY (1)	6	12	4	7	7
	RANGE-KEY (2)	6		4		7
4	DKi ; Ni	DK_4=335; N_4=100000		DK_c= 595; N_c=210000	DK_c= 335; N_c=210000	DK_c= 465; N_c=210000
	INDEX SIZE	599		598	671	466
	SINGLE-KEY QUERY	5	8	3	3	2
	RANGE-KEY (1)	11	32	10	15	8
	RANGE-KEY (2)	11		10		8
5	DKi ; Ni	DK_5=600; N_5=150000		DK_c=1195; N_c=360000	DK_c= 600; N_c=360000	DK_c= 897; N_c=360000
	INDEX SIZE	900		1200	1203	901
	SINGLE-KEY QUERY	6	10	3	4	3
	RANGE-KEY (1)	17	60	15	27	15
	RANGE-KEY (2)	17		15		15
6	DKi ; Ni	DK_6=1000; N_6=200000		DK_c=2195; N_c=560000	DK_c=1000; N_c=560000	DK_c=1597; N_c=560000
	INDEX SIZE	1403		2206	2005	1605
	SINGLE-KEY QUERY	7	12	3	4	3
	RANGE-KEY (1)	26	110	23	43	23
	RANGE-KEY (2)	26		23		23

TABLE 3 ⊠ $K_j \simeq 200$; NRQ = 1/20 of inclusive dist. keys

CLASSES \ KEY DISTRIBUTION		SINGLE-CLASS		DISJOINT CLASS-HIERARCHY	INCLUSIVE CLASS-HIERARCHY	MIXED CLASS-HIERARCHY
		DISJOINT	INCLUSIVE			
2	DKi ; Ni	DK_1 = 100; N_1=2000 DK_2 = 150; N_2=30000		DK_c= 250; N_c= 50000	DK_c= 150; N_c= 50000	DK_c= 200; N_c= 50000
	INDEX SIZE	127		126	151	101
	SINGLE-KEY QUERY	3	4	2	2	2
	RANGE-KEY (1)	7	12	6	9	6
	RANGE-KEY (2)	8		6		6
3	DKi ; Ni	DK_3= 300; N_3=60000		DK_c= 550; N_c=110000	DK_c= 300; N_c=110000	DK_c= 425; N_c=110000
	INDEX SIZE	278		276	301	426
	SINGLE-KEY QUERY	4	6	2	2	2
	RANGE-KEY (1)	11	27	9	16	16
	RANGE-KEY (2)	12		9		16
4	DKi ; Ni	DK_4= 500; N_4=100000		DK_c=1050; N_c=210000	DK_c= 500; N_c=210000	DK_c= 775; N_c=210000
	INDEX SIZE	529		528	501	779
	SINGLE-KEY QUERY	5	8	3	2	3
	RANGE-KEY (1)	17	56	16	26	28
	RANGE-KEY (2)	18		16		28
5	DKi ; Ni	DK_5= 750; N_5=150000		DK_c=1800; N_c=360000	DK_c= 750; N_c=360000	DK_c=1275; N_c=360000
	INDEX SIZE	905		905	754	1280
	SINGLE-KEY QUERY	6	10	3	3	3
	RANGE-KEY (1)	25	105	23	41	41
	RANGE-KEY (2)	26		23		41
6	DKi ; Ni	DK_6=1000; N_6=200000		DK_c=2800; N_c=560000	DK_c=1000; N_c=560000	DK_c=1900; N_c=560000
	INDEX SIZE	1406		1407	2005	1909
	SINGLE-KEY QUERY	7	12	3	4	3
	RANGE-KEY (1)	32	162	29	103	53
	RANGE-KEY (2)	32		29		53

TABLE 4 8 $K_i \propto 500$; NRQ = 1/100 of disjoint dist. keys

CLASSES	KEY DISTRIBUTION	SINGLE-CLASS DISJOINT	SINGLE-CLASS INCLUSIVE	DISJOINT CLASS-HIERARCHY	INCLUSIVE CLASS-HIERARCHY	MIXED CLASS-HIERARCHY
2	DKi ; Ni	DK_1 = 40; N_1=2000 DK_2 = 60; N_2=30000		DK_c = 100; N_c = 50000	DK_c = 60; N_c = 50000	DK_c = 80; N_c = 50000
	INDEX SIZE	102		101	121	161
	SINGLE-KEY QUERY	3	4	2	3	3
	RANGE-KEY (1)	3	4	2	3	3
	RANGE-KEY (2)	3		2		3
3	DKi ; Ni	DK_3=120; N_3=60000		DK_c = 220; N_c=110000	DK_c = 120; N_c=110000	DK_c = 170; N_c=110000
	INDEX SIZE	223		221	241	341
	SINGLE-KEY QUERY	4	6	2	3	3
	RANGE-KEY (1)	6	12	4	7	7
	RANGE-KEY (2)	6		4		7
4	DKi ; Ni	DK_4 = 200; N_4=100000		DK_c = 420; N_c=210000	DK_c = 200; N_c=210000	DK_c = 310; N_c=210000
	INDEX SIZE	424		421	601	621
	SINGLE-KEY QUERY	5	8	2	4	3
	RANGE-KEY (1)	9	24	6	16	11
	RANGE-KEY (2)	9		6		11
5	DKi ; Ni	DK_5 = 300; N_5=150000		DK_c=720; N_c=360000	DK_c = 300; N_c=360000	DK_c = 510; N_c=360000
	INDEX SIZE	725		724	901	1023
	SINGLE-KEY QUERY	6	10	3	4	3
	RANGE-KEY (1)	13	45	11	25	17
	RANGE-KEY (2)	13		11		17
6	DKi ; Ni	DK_6 = 400; N_6=200000		DK_c=1120; N_c=560000	DK_c=400; N_c=560000	DK_c=760; N_c=560000
	INDEX SIZE	1126		1126	1201	1524
	SINGLE-KEY QUERY	7	12	3	4	4
	RANGE-KEY (1)	18	78	15	37	27
	RANGE-KEY (2)	18		15		27

TABLE 5 8 $K_i \simeq 350$; NRQ = 1/20 of mixed dist. keys

CLASSES	KEY DISTRIBUTION	SINGLE-CLASS DISJOINT	SINGLE-CLASS INCLUSIVE	DISJOINT CLASS-HIERARCHY	INCLUSIVE CLASS-HIERARCHY	MIXED CLASS-HIERARCHY
2	DKi ; Ni	$DK_1 = 60$; $N_1=2000$ $DK_2 = 85$; $N_2=30000$		$DK_c = 145$; $N_c= 50000$	$DK_c = 85$; $N_c= 50000$	$DK_c = 115$; $N_c= 50000$
	INDEX SIZE	147		146	171	116
	SINGLE-KEY QUERY	3	4	2	3	2
	RANGE-KEY (1)	8	14	7	13	7
	RANGE-KEY (2)	8		7		7
3	DKi ; Ni	$DK_3 = 170$; $N_3=60000$		$DK_c = 315$; $N_c=110000$	$DK_c = 170$; $N_c=110000$	$DK_c = 243$; $N_c=110000$
	INDEX SIZE	318		316	341	244
	SINGLE-KEY QUERY	4	6	2	3	2
	RANGE-KEY (1)	15	39	13	25	13
	RANGE-KEY (2)	15		13		13
4	DKi ; Ni	$DK_4 = 300$; $N_4=100000$		$DK_c = 615$; $N_c=210000$	$DK_c = 300$; $N_c=210000$	$DK_c = 458$; $N_c=210000$
	INDEX SIZE	619		618	601	459
	SINGLE-KEY QUERY	5	8	3	3	3
	RANGE-KEY (1)	27	96	26	47	24
	RANGE-KEY (2)	27		26		24
5	DKi ; Ni	$DK_5 = 430$; $N_5=150000$		$DK_c =1045$; $N_c=360000$	$DK_c = 430$; $N_c=360000$	$DK_c = 738$; $N_c=360000$
	INDEX SIZE	1050		1050	861	742
	SINGLE-KEY QUERY	6	10	3	3	3
	RANGE-KEY (1)	42	190	40	75	40
	RANGE-KEY (2)	42		40		40
6	DKi ; Ni	$DK_6 = 575$; $N_6=200000$		$DK_c =1620$; $N_c=560000$	$DK_c =575$; $N_c=560000$	$DK_c =1098$; $N_c=560000$
	INDEX SIZE	1628		1628	1153	1103
	SINGLE-KEY QUERY	7	12	3	4	3
	RANGE-KEY (1)	63	338	58	113	58
	RANGE-KEY (2)	62		58		58

TABLE 6 8 K_i = 150 - 500; NRQ = 1/100 of mixed dist. keys

CLASSES	KEY DISTRIBUTION	SINGLE-CLASS DISJOINT	SINGLE-CLASS INCLUSIVE	DISJOINT CLASS-HIERARCHY	INCLUSIVE CLASS-HIERARCHY	MIXED CLASS-HIERARCHY
2	DKi ; Ni	DK_1= 350; N_1=50000 DK_2= 600; N_2=120000		DK_c= 950; N_c=170000	DK_c= 600; N_c=170000	DK_c= 775; N_c=170000
	INDEX SIZE	419		476	603	389
	SINGLE-KEY QUERY	3	4	2	3	2
	RANGE-KEY (1)	7	11	6	11	6
	RANGE-KEY (2)	7		6		6
3	DKi ; Ni	DK_3=600; N_3=180000		DK_c=1550; N_c=350000	DK_c= 600; N_c=350000	DK_c=1075; N_c=350000
	INDEX SIZE	1022		779	1203	1080
	SINGLE-KEY QUERY	5	7	3	4	3
	RANGE-KEY (1)	16	27	9	25	14
	RANGE-KEY (2)	13		9		14
4	DKi ; Ni	DK_4=715; N_4=250000		DK_c=2265; N_c=600000	DK_c= 715; N_c=600000	DK_c=1490; N_c=600000
	INDEX SIZE	1740		2274	1433	1496
	SINGLE-KEY QUERY	6	10	3	4	4
	RANGE-KEY (1)	21	52	18	33	18
	RANGE-KEY (2)	21		18		18
5	DKi ; Ni	DK_5=700; N_5=280000		DK_c=2965; N_c=880000	DK_c= 715; N_c=880000	DK_c=1840; N_c=880000
	INDEX SIZE	2443		2979	2149	1848
	SINGLE-KEY QUERY	7	13	3	5	3
	RANGE-KEY (1)	25	82	21	57	21
	RANGE-KEY (2)	25		21		21
6	DKi ; Ni	DK_6=600; N_6=300000		DK_c=3565; N_c=1210000	DK_c=715; N_c=1210000	DK_c=2140; N_c=1210000
	INDEX SIZE	3046		3582	2864	4285
	SINGLE-KEY QUERY	8	16	3	6	4
	RANGE-KEY (1)	29	117	24	87	45
	RANGE-KEY (2)	29		24		45

TABLE 7 8 $K_i \simeq 200$; NRQ = 20 keys

CLASSES	KEY DISTRIBUTION	SINGLE-CLASS DISJOINT	SINGLE-CLASS INCLUSIVE	DISJOINT CLASS-HIERARCHY	INCLUSIVE CLASS-HIERARCHY	MIXED CLASS-HIERARCHY
2	DKi ; Ni	$DK_1 = 500$; $N_1 = 50000$ $DK_2 = 1000$; $N_2 = 150000$		$DK_c = 1500$; $N_c = 200000$	$DK_c = 1000$; $N_c = 200000$	$DK_c = 1250$; $N_c = 200000$
	INDEX SIZE	436		503	503	418
	SINGLE-KEY QUERY	3	4	3	3	3
	RANGE-KEY (1)	10	15	11	14	9
	RANGE-KEY (2)	9		11		9
3	DKi ; Ni	$DK_3 = 1000$; $N_3 = 160000$		$DK_c = 2500$; $N_c = 360000$	$DK_c = 1000$; $N_c = 360000$	$DK_c = 1750$; $N_c = 360000$
	INDEX SIZE	771		838	1005	880
	SINGLE-KEY QUERY	4	6	3	3	3
	RANGE-KEY (1)	11	24	11	23	14
	RANGE-KEY (2)	11		11		14
4	DKi ; Ni	$DK_4 = 1150$; $N_4 = 200000$		$DK_c = 3650$; $N_c = 560000$	$DK_c = 1150$; $N_c = 560000$	$DK_c = 2400$; $N_c = 560000$
	INDEX SIZE	1349		1223	1156	1206
	SINGLE-KEY QUERY	6	9	3	3	3
	RANGE-KEY (1)	17	38	11	23	14
	RANGE-KEY (2)	15		11		14
5	DKi ; Ni	$DK_5 = 1250$; $N_5 = 250000$		$DK_c = 4900$; $N_c = 810000$	$DK_c = 1250$; $N_c = 810000$	$DK_c = 3075$; $N_c = 810000$
	INDEX SIZE	1977		1642	2506	3090
	SINGLE-KEY QUERY	7	12	3	4	3
	RANGE-KEY (1)	18	52	11	43	23
	RANGE-KEY (2)	19		11		23
6	DKi ; Ni	$DK_6 = 1300$; $N_6 = 255000$		$DK_c = 6200$; $N_c = 1065000$	$DK_c = 1300$; $N_c = 1065000$	$DK_c = 3750$; $N_c = 1065000$
	INDEX SIZE	2630		3115	2606	3768
	SINGLE-KEY QUERY	8	15	3	4	3
	RANGE-KEY (1)	19	66	14	43	23
	RANGE-KEY (2)	20		14		23

TABLE 8 ⊠ $K_i \simeq 500$; NRQ = 20 keys

CLASSES	KEY DISTRIBUTION	SINGLE-CLASS DISJOINT	SINGLE-CLASS INCLUSIVE	DISJOINT CLASS-HIERARCHY	INCLUSIVE CLASS-HIERARCHY	MIXED CLASS-HIERARCHY
2	DKi ; Ni	DK_1= 100; N_1=50000 DK_2= 300; N_2=150000		DK_c= 400; N_c=200000	DK_c= 300; N_c=200000	DK_c= 350; N_c=200000
	INDEX SIZE	402		401	601	701
	SINGLE-KEY QUERY	3	4	2	3	3
	RANGE-KEY (1)	22	42	21	41	41
	RANGE-KEY (2)	22		21		41
3	DKi ; Ni	DK_3=350; N_3=160000		DK_c= 750; N_c=360000	DK_c=350; N_c=360000	DK_c= 550; N_c=360000
	INDEX SIZE	753		753	1051	1103
	SINGLE-KEY QUERY	4	6	3	4	4
	RANGE-KEY (1)	23	63	23	61	43
	RANGE-KEY (2)	23		23		43
4	DKi ; Ni	DK_4= 420; N_4=200000		DK_c=1170; N_c=560000	DK_c= 420; N_c=560000	DK_c= 795; N_c=560000
	INDEX SIZE	1174		1176	1261	1594
	SINGLE-KEY QUERY	5	8	3	4	4
	RANGE-KEY (1)	24	84	23	61	43
	RANGE-KEY (2)	24		23		43
5	DKi ; Ni	DK_5= 550; N_5=250000		DK_c=1720; N_c=810000	DK_c= 550; N_c=810000	DK_c=1135; N_c=810000
	INDEX SIZE	1727		1728	1653	1140
	SINGLE-KEY QUERY	7	11	3	5	4
	RANGE-KEY (1)	27	107	23	63	43
	RANGE-KEY (2)	26		23		43
6	DKi ; Ni	DK_6= 600; N_6=255000		DK_c=2320; N_c=1065000	DK_c= 600; N_c=1065000	DK_c=1460; N_c=1065000
	INDEX SIZE	2330		2331	2403	1466
	SINGLE-KEY QUERY	8	14	3	6	4
	RANGE-KEY (1)	28	130	23	83	43
	RANGE-KEY (2)	28		23		43

TABLE 9 ⊗ $K_i \simeq 350$; NRQ = 20 keys

CLASSES	KEY DISTRIBUTION	SINGLE-CLASS		DISJOINT CLASS-HIERARCHY	INCLUSIVE CLASS-HIERARCHY	MIXED CLASS-HIERARCHY
		DISJOINT	INCLUSIVE			
2	DKi ; Ni	$DK_1 = 150$; $N_1 = 50000$ $DK_2 = 400$; $N_2 = 150000$		$DK_c = 550$; $N_c = 200000$	$DK_c = 400$; $N_c = 200000$	$DK_c = 475$; $N_c = 200000$
	INDEX SIZE	552		553	401	476
	SINGLE-KEY QUERY	3	4	3	2	2
	RANGE-KEY (1)	22	42	23	21	21
	RANGE-KEY (2)	22		23		21
3	DKi ; Ni	$DK_3 = 480$; $N_3 = 160000$		$DK_c = 1030$; $N_c = 360000$	$DK_c = 480$; $N_c = 360000$	$DK_c = 755$; $N_c = 360000$
	INDEX SIZE	1033		1035	961	758
	SINGLE-KEY QUERY	4	6	3	3	3
	RANGE-KEY (1)	23	63	23	41	23
	RANGE-KEY (2)	23		23		23
4	DKi ; Ni	$DK_4 = 600$; $N_4 = 200000$		$DK_c = 1630$; $N_c = 560000$	$DK_c = 600$; $N_c = 560000$	$DK_c = 1115$; $N_c = 560000$
	INDEX SIZE	1636		1638	1203	1120
	SINGLE-KEY QUERY	6	9	3	4	3
	RANGE-KEY (1)	26	86	23	43	23
	RANGE-KEY (2)	25		23		23
5	DKi ; Ni	$DK_5 = 700$; $N_5 = 250000$		$DK_c = 2330$; $N_c = 810000$	$DK_c = 700$; $N_c = 810000$	$DK_c = 1515$; $N_c = 810000$
	INDEX SIZE	2339		2341	2104	3037
	SINGLE-KEY QUERY	7	12	3	5	4
	RANGE-KEY (1)	27	111	23	63	43
	RANGE-KEY (2)	27		23		43
6	DKi ; Ni	$DK_6 = 750$; $N_6 = 255000$		$DK_c = 3080$; $N_c = 1065000$	$DK_c = 750$; $N_c = 1065000$	$DK_c = 1915$; $N_c = 1065000$
	INDEX SIZE	3092		3095	2254	3834
	SINGLE-KEY QUERY	8	15	3	5	4
	RANGE-KEY (1)	28	134	23	63	43
	RANGE-KEY (2)	28		23		43