χ-tree : A New Class Hierarchy Access Method for Object-Oriented Databases

Abstract

The semantic richness of the object-oriented data model has introduced many new issues on the design and implementation of OODBMSs. An important performance issue concerns the indexing support for evaluation of object-oriented queries. In particular, the class hierarchy concept has an impact on the *class scope* of queries: an object-oriented query on a class C can be interpreted as to retrieve all objects that satisfy the query belonging to either only class C itself, or all classes in the class hierarchy rooted at class C. To support the evaluation of such queries, a *class-hierarchy index* is required to facilitate retrieval of objects both from a single class, as well as from a class hierarchy. In this paper, we propose a new class-hierarchy indexing method called the χ-tree . Unlike existing class-hierarchy access methods which are based on the B^+-tree , the χ-tree is a two-dimensional access method that dynamically partitions the data space using both the class and indexed attribute dimensions. However, unlike conventional multi-dimensional access methods which assume that each indexed dimension is linearly ordered, the χ-tree uses a novel node splitting heuristic which takes into account the non-linearity of the class dimension. The χ-tree is a dynamic structure that uses a probabilistic approach to cluster index records which is amenable to fine tuning given information on the access frequency pattern. Experimental results show that although the χ-tree requires more storage space than the CH-tree [15], the search performance of the χ-tree is better than the CH-tree is majority of the cases.

Keywords: Access methods, access proximity, class hierarchy, multi–dimensional, object-oriented databases, performance evaluation.

1 Introduction

With the popularity of object-oriented technology, object-oriented database systems (OODBSs) in particular, have emerged as a promising approach towards supporting non-traditional applications such as CAD/CAD systems, CASE systems, and knowledge database systems. Such applications usually require support for modeling and representing richly structured and complex entities, which are inadequately supported by the fixed data types and limited conceptual structures available in conventional database systems.

The advent of OODBSs has introduced new indexing issues due to the semantic richness of the object-oriented model; in particular, the following three object-oriented concepts have an impact on the evaluation of object-oriented queries, as well as the indexing support required.

- **Class Hierarchy**
  A key concept in the object-oriented data model (OODM) is the notion of a class hierarchy (also known as an inheritance hierarchy). In the OODM, each entity is modeled by an object; objects are classified into classes based on their properties and behaviour. A class can be

specialized into a number of subclasses, giving rise to a hierarchy of classes which captures the IS-A relationship between classes: an object of a class is also an object of its ancestor classes (superclasses), and the properties of a class are inherited by all its descendant classes (subclasses). Figure 1 (modified from [15]) shows a database schema, where each box represents a class with its attributes listed within the box. There is a class hierarchy rooted at class *Vehicle* which is specialized into subclasses *Automobile* and *Truck*; class *Automobile* is further specialized into classes *DomesticAutomobile* and *ForeignAutomobile*. The attribute *color* of class *Vehicle* is inherited by all its subclasses *Automobile*, *Truck*, *DomesticAutomobile*, and *ForeignAutomobile*. An implication of the inheritance hierarchy semantics for the evaluation of an object-oriented query is the *class scope* of the query; that is, the classes of the objects that satisfy the query. Unlike in the relational model, where a query on a relation R retrieves tuples from only R itself, an object-oriented query on a class C has two possible interpretations: it can mean to retrieve objects either from only the queried class C itself (a **single-class query**), or from all the classes in the class hierarchy rooted at C (a **class-hierarchy query**) since any object of a subclass of C is also an object of C. Consider the following query on the schema in Figure 1: *"Retrieve all autombiles with color = red"*. If the query is interpreted as a single-class query, then only objects of class *Automobile* that satisfy the constraint on *color* need to be retrieved. However, if it is interpreted as a class-hierarchy query, then the class scope of the query is the set $\{Automobile, DomesticAutomobile, ForeignAutomobile\}$ and all objects of this set of classes that satisfy the constraint on *color* need to be retrieved. The interpretation of the query type (single-class or class-hierarchy) is specified by the user. To facilitate the evaluation of such types of queries, an index method needs to support efficient retrieval of objects from a single class, as well as from all the classes in the class hierarchy. We refer to such an index as a **class-hierarchy index**.

- **Aggregation Hierarchy**
  Besides the class-subclass relationship, another important type of relationship between classes is the attribute-domain relationship. In the OODM, the domain of an attribute of a class can be either a primitive type (e.g. integer, real, string) or a complex domain (i.e. a class domain). A class can thus be defined as a nested structure of classes, giving rise to an aggregation hierarchy (or class composition hierarchy). Note that the concepts of a class hierarchy and an aggregation hierarchy are orthogonal. An example of an aggregation hierarchy is illustrated in Figure 1 where *Vehicle* is the root class of the aggregation hierarchy. *Company* is the domain of attribute *manufacturer* in class *Vehicle*, and class *Employee* is the domain of attribute *president* in class *Company*. The attribute *age* in class *Employee* is said to be a *nested attribute* of class *Company* as well as class *Vehicle*. An example of an OO query with nested attributes is: *"Retrieve all vehicles manufactured by companies whose president is less than 30 years old"*. Without efficient index support, the evaluation of such queries can be slow as it requires access to multiple classes.

- **Methods**
  While the structural properties of objects in the OODM are described by attributes, their behavourial properties are defined by methods which are procedures to either access the values of the attributes, or operate on the attribute values. A method can be used in object-oriented queries as a derived attribute that returns a value (or an object), or as a predicate that returns a boolean value. To speed up the evaluation of OO query predicates that involve methods, efficient index support is necessary.

In this paper, our focus is on index support for the evaluation of single-class and class-hierarchy queries (i.e. class-hierarchy indexing). Among the three indexing issues, aggregation-hierarchy indexing has been the most extensively studied [20, 25, 8, 23, 14, 3, 7]. Work on method indexing includes [12, 13, 4, 9]. For a comprehensive survey on indexing issues in OODBMSs, the reader may refer to [5].

A class-hierarchy index is characterized by two parameters:

(1) the hierarchy of classes to be indexed, referred to as the *indexed class hierarchy*, and

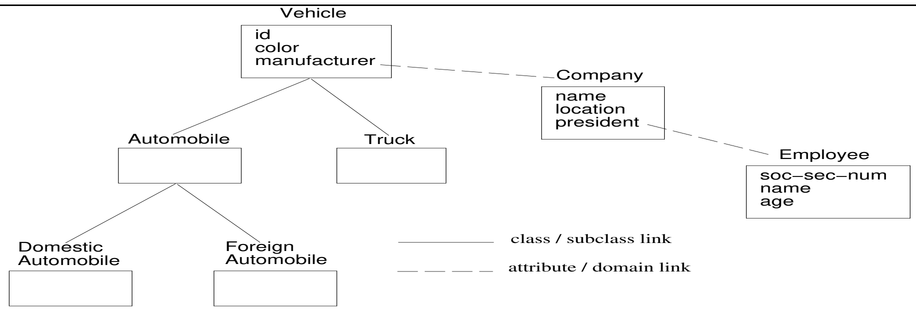

Figure 1: A Database Schema

(2) an attribute of the root class of the indexed hierarchy, referred to as the *indexed attribute*.

Existing class-hierarchy indexing methods, which are based on the B^+-tree , can be classified, based on the partitioning the data space, into two categories:

- *Class-dimension based approach* [16, 19] which partitions the data space primarily on the class of an object.
- *Attribute-dimension based approach* [16] which partitions the data space primarily on the indexed attribute of an object.

While the class-dimension based approach supports single-class queries efficiently, its support for class-hierarchy queries is unsatisfactory due to the need to traverse multiple single-class indexes. On the other hand, the attribute-dimension based approach generally provides efficient support for class-hierarchy queries on the root class (i.e., retrieving objects of all the indexed classes), but is inefficient for single-class queries or class-hierarchy queries on a sub-hierarchy of the indexed class hierarchy, as it may need to access many irrelevant leaf nodes of the single index structure. A more recent class-hierarchy access method, the hcC-tree [24], can be regarded as both a class-dimension based and an attribute-dimension based method as it duplicates the data space to support both ways of partitioning; such an approach provides efficient search performance at the expense of high storage and update cost.

We propose a new class-hierarchy index called the χ-tree (class-hierarchy index tree) which attempts to combine the complementary strengths of both approaches to provide good search performance for all query types. The χ-tree is essentially a two-dimensional index method that considers both the class and attribute dimensions together in the partitioning of the data space, thereby overcoming the inherent biasness of the class-dimension and attribute-dimension based approaches. Unlike existing approaches, the χ-tree exploits the semantics of class hierarchy to partition the data space. The basis of partitioning for the χ-tree is based on the notion of *access proximity* which can incorporate information on the query distribution to improve the partitioning. Experimental results show that although the χ-tree has higher space requirements compared to Kim's class-hierarchy index method [16], it provides better overall search performance.

The main contributions of this paper are:

(1) The proposal of a novel approach to class-hierarchy indexing which organizes the data as a two-dimensional space, in contrast to existing approaches which treats the data space as one-dimensional.

(2) The design of a new splitting strategy to handle the class dimension which is non-linear. This is a major difference from existing multi-dimensional access methods which assume that each of the indexed dimensions is linearly ordered.

(3) An empirical performance comparison which shows that the new approach gives good search performance for various types of queries.

The rest of this paper is organized as follows. Section 2 presents a model for object-oriented queries. Section 3 surveys existing access methods for class-hierarchy indexing. The motivation for our new approach to class-hierarchy indexing, the χ-tree , is presented in Section 4. Section 5 presents the design of the χ-tree . The node splitting criterion and heuristics for the χ-tree is discussed in Section 6. Section 7 presents the update algorithms for the χ-tree . In Section 8, we present and discuss the experimental results conducted to compare both the space requirement and search performance of the χ-tree with Kim's CH-tree [16]. Finally we summarize our results in Section 9.

2 Query Model

A **class hierarchy** (with only single inheritance) can be represented by a rooted directed tree $H = (V, E)$ where V represents the set of all classes in the hierarchy, and $E \subseteq V \times V$ represents the set of class-subclass relationships such that $(c, c') \in E$ iff c' is an immediate subclass of c, and $\exists c_r \in V$ such that $\forall c \in V, (c, c_r) \notin E$; i.e., c_r is the root of the class hierarchy. A class C in a class hierarchy H is termed as an **internal class** if C has at least one subclass in H; otherwise C is termed as a **leaf class**.

We shall use H_{c_r} to denote a class hierarchy with root class c_r, and $Class(H_{c_r})$ to denote the set of all classes in the class hierarchy rooted at C_r; i.e., $C \in Class(H_{c_r})$ iff either $C = C_r$ or there is a directed path from C_r to C. Given a class hierarchy H, $S \subseteq V$ is said to be a **class scope** of H iff S is a singleton set or $S = Class(H_{c_i})$ where C_i is an internal class in H. Thus, a class hierarchy H with n_i internal classes and n_l leaf classes has a total of $(2n_i + n_l)$ distinct class scopes.

To illustrate the definitions, consider the class hierarchy H_A shown in Figure 2. $Class(H_A) = \{A, B, C, D, E, F, G\}$ and all of these classes, with the exception of classes A and B, are leaf classes. There are a total of 9 distinct class scopes in H: $\{A\}, \{B\}, \{C\}, \{D\}, \{E\}, \{F\}, \{G\}, Class(H_A)$ and $Class(H_B)$.

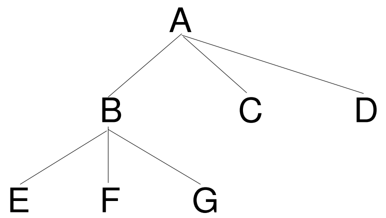

Figure 2: A Class Hierarchy

For the rest of this paper, we shall use H (with the root class subscript omitted) to denote the indexed class hierarchy, and A to denote the indexed attribute (i.e. A is an attribute of the root class of H). The domain of A is assumed to be bounded by $[A_{min}, A_{max}]$.

A **simple query** defined on a class hierarchy H and an attribute A is represented by a triple

$$q = (qc, qr_1, qr_2)$$

where qc, referred to as the **query class scope**, is a class scope of H, and $[qr_1, qr_2]$, referred to as the **query range**, is a range of the attribute domain such that $A_{min} \leq qr_1 \leq qr_2 \leq A_{max}$. A query is termed a **single-class query (SC-query)** if the query class scope is a singleton set; otherwise, it is termed a **class-hierarchy query (CH-query)**. A CH-query is further classified as a **complete CH-query (CCH-query)** if the query class scope is equal to the entire class hierarchy; i.e., $qc = Class(H)$; otherwise, the CH-query is referred to as a **partial CH-query (PCH-query)**.

3 Existing Class-Hierarchy Access Methods

This section presents a survey of existing class-hierarchy access methods. For each method, we briefly describe the key ideas involved and evaluate its performance in terms of its storage space overhead, update cost, and retrieval cost for various types of queries (SC-queries, CCH-queries, and PCH-queries).

3.1 SC-tree and CH-tree

The issue of class-hierarchy indexing was first addressed in a seminal paper by Kim *et al* [16] where two methods, based on the B^+-tree [10], were proposed.

In the first approach, known as the *single-class index* (SC-tree), a separate B^+-tree index is maintained on the indexed attribute for each class in the class hierarchy. Thus the SC-tree is physically implemented by N separate indexes, where N is the total number of classes in the indexed class hierarchy. This approach is very efficient for SC-queries since only the B^+-tree index corresponding to the query class needs to be searched, with all the qualified index leaf node entries clustered together. However, for a CH-query, the SC-tree requires searching multiple B^+-tree indexes, one for each class in the class scope of the CH-query. In this case, the number of leaf nodes scanned is likely to be much more than actual number of leaf nodes required to contain all the qualified leaf node entries. This is due to the "declustering" effect caused by the partitioning of all the leaf node entries into N separate indexes.

In the second approach, known as the *class hierarchy index* (CH-tree), a single B^+-tree index is maintained on the indexed attribute for all the classes in the class hierarchy. Clearly, for SC-queries, the performance of the CH-tree cannot be better than that of the SC-tree since a multiple-classes B^+-tree is much larger than a single-class B^+-tree , and the clustering of index leaf node entries with respect to a single class in the CH-tree is certainly weaker than that in the SC-tree . For CH-queries, it has been shown that the CH-tree outperforms the SC-tree , especially for CCH-queries and to a lesser extent for PCH-queries. Comparing the update cost of the SC-tree and CH-tree , the latter is expected to be more expensive since its size is likely to be larger than any one of the single-class B^+-trees in a SC-tree .

3.2 H-tree

The H-tree [19] is a variant of the SC-tree which aims to improve on the retrieval performance of the SC-tree for CH-queries. As in the SC-tree , a separate B^+-tree index is maintained on the indexed attribute for each class in the class hierarchy. However, unlike the SC-tree where the individual B^+-trees are independent of one another, the B^+-trees in the H-tree are linked up based on their class-subclass relationships via additional pointers (known as nested pointers) in the internal nodes of the B^+-trees . For each class-subclass relationship in the indexed class hierarchy; i.e., $(C, C') \in E$, a set of nested pointers are maintained in the internal nodes in the B^+-tree corresponding to class C which point to various internal nodes in the B^+-tree corresponding to class C'.

For CH-queries, the H-tree needs to perform a full search only on the B^+-tree index corresponding to the query class, followed by a partial search (via nested pointers) on each of the B^+-tree indexes corresponding to the other classes in the class hierarchy rooted at the query class. Although this is an improvement over the SC-tree , the H-tree still suffers from the "declustering" effect as in the SC-tree ; i.e., the H-tree may access less internal nodes compared to the SC-tree , but the number of leaf nodes accessed is still the same. For SC-queries, the performance of the H-tree is no better than the SC-tree since the H-tree has a lower fanout compared to the SC-tree due to the additional nested pointers (and associated information) in the H-tree.

The retrieval improvement of the H-tree, however, is achieved at the expense of the additional storage overhead for the nested pointers and the update cost incurred for their maintenance. For example, consider two B^+-tree indexes I and I' corresponding to the classes C and C', respectively, where C' is an immediate subclass of C. The number of nested pointers from I to I' is at least one (linking to just the root node of I') and at most L where L is the number of parent-of-leaf nodes in I' (linking to each parent-of-leaf node of I').

3.3 hcC-tree

The hcC-tree [24] is a variant of the CH-tree which tries to combine the strengths of both the SC-tree and CH-tree . As in a CH-tree , it uses a single B^+-tree -like structure to index the entire class-hierarchy. The hcC-tree is organized into three types of index nodes: internal, leaf, and oid nodes. The first two types of nodes are somewhat similar to the internal and leaf nodes of a B^+-tree , respectively. The oid nodes are at one level below the leaf nodes.

Given a class hierarchy with n classes, the hcC-tree maintains $n+1$ chains of oid nodes with n class-chains (one each for each class) and one hierarchy-chain. The class-chain for a class C clusters, based on the indexed key, the oids belonging to only class C; while the hierarchy-chain clusters all the oids. Thus, the class-chain for class C is similar to the chain of leaf nodes in the B^+-tree for class C in a SC-tree ; while the hierarchy-chain is similar to the chain of leaf nodes in a CH-tree . The oid nodes are referenced by entries in the leaf nodes. Each leaf node entry consists of a key value K, a n-bit bitmap, and a set P of $(n+1)$ pointers. Each bit corresponds to a class in the indexed class hierarchy such that if the i^{th} bit is set, then the i^{th} pointer in P points to the first node in the class-chain for class C containing oids with key values K. Each internal node entry consists of a key value, a node pointer and a n-bit bitmap.

For SC-queries, the performance of the hcC-tree is comparable to the SC-tree as it searches only one class-chain; for CCH-queries which are range queries, the performance of the hcC-tree is comparable to the CH-tree as it searches only the hierarchy-chain. However, for PCH-queries which are range queries, the hcC-tree has to search a number of class-chains corresponding to each of the classes in the class scope of the queries. This has a similar "declustering" problem as in a SC-tree . Furthermore, the hcC-tree incurs quite a high storage overhead and update cost as each oid is stored twice in the oid nodes: once each in a class-chain and in the hierarchy-chain.

4 Motivation for a Multi-Dimensional Approach to Class-Hierarchy Indexing

A major weakness of existing class-hierarchy access methods is that it clusters the index records based on only one of the two dimensions (class and attribute); its performance is therefore biased towards certain types of queries.

Figure 3 shows the partitioning of the data space (i.e. the entries in the leaf nodes of the index structures) for both the class-dimension and attribute-dimension based methods. Consider the following two queries:

- *Query 1*, represented by a horizontal search area in Figure 3, is a SC-query with a wide attribute value range, and
- *Query 2*, represented by the vertical search area in Figure 3, is a CH-query with a narrow attribute value range.

Query 1 is efficiently supported by the class-dimension based approach (Figure 3(a)) as it requires only a partial sequential scan of the leaf nodes of a single class index. However, this query is not well supported by the attribute-dimension based approach (Figure 3(b)) because it involves a long sequential scan of a single large index which is likely to access many irrelevant leaf nodes. This scenario can happen, for example, when only very few of the objects of the single class being queried have attribute values in the query range but many objects of the other classes in the class hierarchy have attribute values that fall within the query range. This makes it necessary to sequentially scan many leaf pages but with only a small portion of the scanned pages contributing to the query results. On the other hand, although *Query 2* is not efficiently supported by the class-dimension based approach (Figure 3(a)) due to the need to traverse multiple B^+-tree indexes, it is efficiently supported by the attribute-dimension based approach (Figure 3(b)) because it requires a partial sequential scan of just one B^+-tree index.

Hence the search space for the class-hierarchy indexing is actually 2-dimensional. This is evident from the query model presented in Section 2: a query specifies both a class scope as well as an

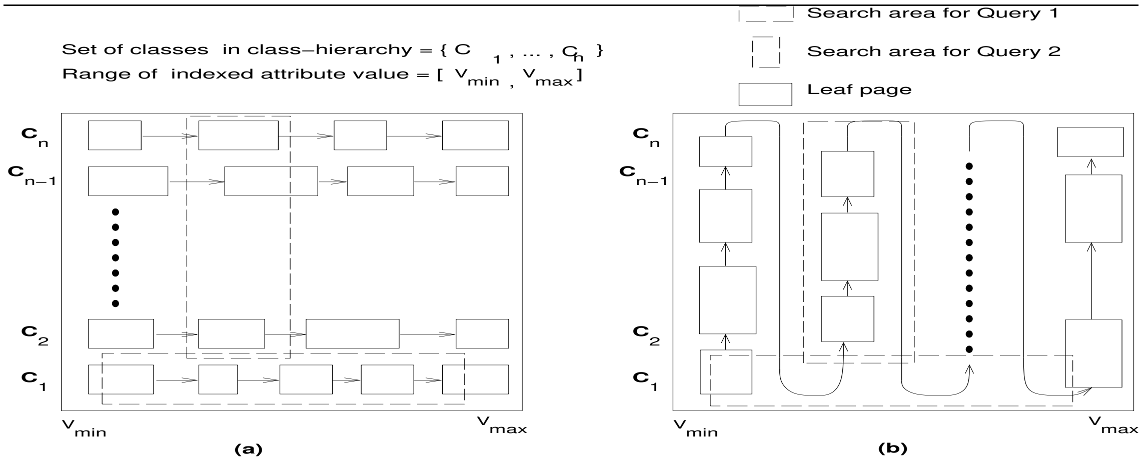

Figure 3: Partitioning of data space for (a) class-dimension based approach, and (b) attribute-dimension based approach.

attribute range. The class-hierarchy indexing problem can therefore be treated as a special case of the multi-dimensional indexing problem. The main advantage of using such an approach is that by clustering the index records based on both dimensions, the overall search performance for different query types should be better. Furthermore, a multi-dimensional indexing approach naturally lends itself to support indexing of attributes which are multi-dimensional in nature.

5 Design of the χ-tree

To avoid the inherent weaknesses of existing class-hierarchy index methods, which are based on a single-dimensional access method, we propose the χ-tree , a new class-hierarchy access method which is essentially a 2-dimensional point access method. The data records is organized as a 2-dimensional space with each indexed object represented by a 2-dimensional point (x, y) where x represents the class of the object and y represents the indexed attribute value of the object.

5.1 Differences from Conventional Multi-Dimensional Access Methods

Research on multi-dimensional access methods has been an active area driven by the need to support access to large collections of multi-dimensional geometric objects on secondary storage. Many such access methods have been proposed for multi-dimensional point and/or spatial data: grid file [21], R-tree [11], hB-tree [18], and Buddy-tree [22], etc.

All the existing multi-dimensional access methods are designed for indexing data where there exists a natural total order on each dimension of the data. Examples include geographic data such as locations of cities in terms of their longitudes and latitudes, and geometric data such as design objects in 3-dimensional euclidean space. However, for class-hierarchy indexing, the class dimension of the indexed data is non-linear as classes are organized into a hierarchical structure based on their class-subclass relationships. Although one can impose an arbitrary ordering on the classes in the indexed class-hierarchy (e.g. based on some traversal order), such an ordering is not likely to lead to an effective clustering of the index data. This means that node splitting heuristics designed for multi-dimensional access methods cannot be used to split nodes in the χ-tree . We shall present a new node splitting strategy in Section 6 after we have described the structure of the χ-tree . The new node splitting strategy, which takes into account of the non-linearity of the class dimension, is an important distinction from conventional multi-dimensional access methods.

5.2 Index Structure and Search Operation

The χ-tree is a dynamic, height-balanced, tree structured index method similar to the R-tree [11] used in spatial searching. Data are stored in the leaf nodes which appear at the same level. When a leaf node overflows, it is split into two nodes and a new entry is inserted into the parent node of the split node, which may cause the splitting to propagate all the way up to the root node. Each child node has an entry in its parent which records the data region defined by the entries in the child node.

The structure of the χ-tree leaf and internal nodes are shown in Figure 4. Each leaf node entry represents an object by a triple

$$(\text{cid}, \text{oid}, key)$$

where *cid* is the class identifier of the object, *oid* is the object's unique identifier, and *key* is the object's value of the indexed attribute A. The leaf node entries are organized into variable-length records such that all the node entries with the same class identifier are grouped into the same record.

Internal nodes contain entries of the form

$$(cidSet, key_{min}, key_{max}, P)$$

where $cidSet$ is a subset of the classes in the indexed class hierarchy H (i.e. $cidSet \subseteq Class(H)$). $[key_{min}, key_{max}]$ is a subrange of the attribute domain, and P is a child pointer pointing to a child node on the next level (i.e. the address of the corresponding descendant node). The internal node entries are organized into variable-length records such that the all node entries with the same set of classes are grouped into the same record.

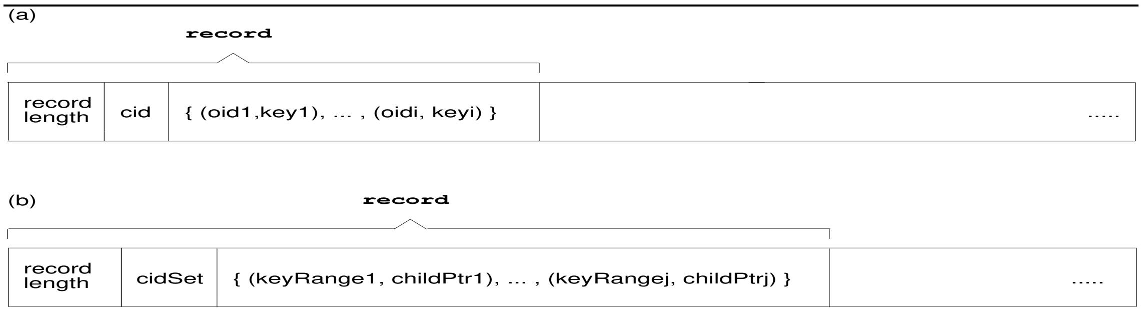

Figure 4: Structure of χ-tree (a) Leaf Node (b) Internal Node

Let M_{int} and M_{leaf} be the maximum number of entries that can fit into an internal and a leaf node, respectively. Let f be the **utilization factor** of the χ-tree where $0 < f \leq 0.5$. A χ-tree satisfies the following properties.

(I1) Each leaf node has between $\lceil f \times M_{leaf} \rceil$ and M_{leaf} entries unless it is also the root node.

(I2) Each internal node has between $\lceil f \times M_{int} \rceil$ and M_{int} entries unless it is also the root node.

(I3) The root node has at least two entries unless it is also a leaf node.

(I4) Let E be an internal node entry and N be the node pointed to by $E.P$.

If N is a leaf node, then

$$E.cidSet = \{e.\text{cid} \mid e \text{ is an entry in } N\},$$
$$E.key_{min} = min\{e.key \mid e \text{ is an entry in } N\}, \text{ and}$$
$$E.key_{max} = max\{e.key \mid e \text{ is an entry in } N\}.$$

If N is an internal node, then

$$E.cidSet = \bigcup_{\forall e_i \in N} e_i.cidSet \text{ where } e_i \text{ is an internal node entry in } N,$$

$$E.key_{min} = min\{e.key_{min} \mid e \text{ is an entry in } N\}, \text{ and}$$

$$E.key_{max} = max\{e.key_{max} \mid e \text{ is an entry in } N\}.$$

(I5) All leaf nodes are on the same level.

Given a query q, the *Search* algorithm searches the χ-tree index and returns a list of leaf node entries that satisfies the query in R, which is first initialized to be empty. Each entry in R is of the form (cid , oid). The search operation traverses the χ-tree recursively in a top-down manner (T is first initialized to point to the root node); if an internal node entry satisfies the query (i.e., the subspace represented by the node entry intersects with the query space) the algorithm proceeds to search its child node and so on until the leaf node level is reached. As in R–trees, searching in a χ-tree may involve traversal of more than one path down to the leaf level as the subspaces represented by the internal node entries may overlap.

Algorithm Search

Search (T, q, R)

Input: T is a pointer to a node of a χ-tree .
q is a simple query.
Output: R is the result list containing leaf node entries that satisfy the query q.

let N be the node pointed to by T;
if (N is an internal node) **then**
begin
  for each entry E in N **do**
    if $(E.cidSet \cap qc \neq \emptyset)$ **and** $([E.key_{min}, E.key_{max}] \cap [qr_1, qr_2] \neq \emptyset)$ **then**
      invoke **Search** $(E.P, q, R)$
end else begin
  for each entry E in N **do**
    if $(E.\mathrm{cid} \in qc)$ **and** $(E.key \in [qr_1, qr_2])$ **then**
      add E to R;
end

---

6 Node Splitting Strategy

As in single-dimensional access methods such as the B^+-tree , an insertion of a new entry into a full node will cause the node to overflow; the node is then split into two nodes by distributing the entries in the overflowed node between the original node and a new node. In multi-dimensional access methods, node splitting is more complicated due to the multi-dimensional space. The node splitting strategy essentially splits the k-dimensional space defined by the entries in the original full node into two smaller subspaces defined by the entries that have been distributed to the two nodes.

The node splitting strategy is crucial to the performance of the access method as it affects both its search efficiency and space utilization. The following are two desirable characteristics of a node split:

(C1) the split should minimize the likelihood that both the split nodes need to be examined during searching, and

(C2) the split should maintain a reasonable space utilization factor.

Note that maintaining a high space utilization is also important to search efficiency since a low space utilization will likely lead to a larger index and therefore a more costly search.

However, to obtain a good split usually involves a tradeoff between these two requirements. On the one hand, to satisfy (C1), it is desirable to have a *disjoint split*; i.e., a split such that the subspaces defined by the resulting split nodes are disjoint. On the other hand, to satisfy (C2), it is desirable to have an *even split*; i.e., a split such that the number of entries in each of the split nodes are almost equal. Unfortunately, it is not always possible to achieve a split that is both disjoint and even. Consider an overflow node with 100 k-dimensional data points which differ only in their k^{th} dimension such that for the k^{th} dimension, 80 of the entries have the value k_1 while the remaining 20 entries have the value k_2 ($k_1 < k_2$). Clearly, a disjoint split in this case has a poor space utilization.

The search for an optimal node split (based on some cost function) is an intractable problem due to the large search space. Hence, for node splitting to be efficient, heuristics will have to be employed to prune the search space. The node splitting heuristics adopted in all the existing multi-dimensional access methods are based on the assumption that each of the dimensions of the data is totally ordered. In the R-tree [11], for example, the node splitting criterion is to minimize the total covering area of the two regions corresponding to the split nodes, assuming that a smaller total area makes it less likely for the query search window to intersect the regions of both nodes. Another criterion considered in a R-tree variant, the R^*-tree [2], is to minimize the amount of overlap area between the two regions of the corresponding split nodes, assuming that a larger overlap area increases the likelihood that the query search window intersects both the regions. Both these heuristics implicitly assume that all the indexed dimensions are linearly ordered to define areas of subspaces. For class-hierarchy indexing, however, the non-linearity of the class dimension makes it difficult to define a satisfactory notion of area for the data space regions corresponding to the index nodes. To alleviate this difficulty, we propose a new node splitting criterion that not only incorporates the non-linear class dimension, but also takes into account the semantics of the class hierarchy.

Our node splitting strategy is to generate, using some heuristics, a small set of candidate node splits, from which the best node split is selected based on some cost metric. We first present some definitions relating to node splits in Section 6.1. Next, we present our cost metric for evaluating the "goodness" of node splits in Section 6.2, followed by our heuristics for generating candidate node splits in Section 6.3.

6.1 Definitions

Let S be the set of entries in an overflowed node N. A node split distributes the entries in S between N itself and a new node N'; we shall refer to N and N' as **split nodes**. A **node split of** S is denoted by $S_{X,Y}$ where X and Y represent the set of entries in each of the split nodes, respectively; i.e., $X, Y \subseteq S$, $X \cup Y = S$, and $X \cap Y = \emptyset$. A node split $S_{X,Y}$ is said to be an **admissible node split** iff $|X|, |Y| \geq f \times M$, where f is the node utilization factor, and M is the number of entries in a full node.

The 2-dimensional space embedded by a set of entries S can be represented by a triple

$$(Class(S)\ , MinKey(S)\ , MaxKey(S)\ ),$$

where $Class(S)$ denotes the set of classes of all the entries in S, and $MinKey(S)$ and $MaxKey(S)$ denote the lower and upper bounds on the attribute values of the entries in S, respectively. In particular, if S is a set of leaf node entries, then $Class(S) = \{s.\mathrm{cid}\,|s \in S\}$, $MinKey(S) = \min\{s.key\,|s \in S\}$, and $MaxKey(S) = \max\{s.key\,|s \in S\}$; otherwise, if S is a set of internal node entries, then $Class(S) = \bigcup_{s \in S} s.cidSet$, $MinKey(S) = \min\{s.key_{min}\,|s \in S\}$, and $MaxKey(S) = \max\{s.key_{max}\,|s \in S\}$. We shall use $KeyRange(S)$ to denote $[MinKey(S)\ , MaxKey(S)\ ]$.

Let $S_{X,Y}$ be a node split of a set of entries S. A class C is a **shared class** of $S_{X,Y}$ iff $C \in Class(X) \cap Class(Y)$. A class scope CS is a **shared class scope** of $S_{X,Y}$ iff $\exists C', C'' \in CS$

such that $C' \in Class(X)$ and $C'' \in Class(Y)$. Clearly, $Class(H)$ is a shared class scope in any node split, so there is at least one shared class scope in every node split. By definitions of a shared class and shared class scope, the following two properties hold:

(P1) If C is a shared class of $S_{X,Y}$, then both $\{C\}$ and $Class(H_C)$ are shared class scopes of $S_{X,Y}$ [1].

(P2) If $Class(H_C)$ is a shared class scope of $S_{X,Y}$ and C' is a superclass of C, then $Class(H_{C'})$ is also a shared class scope of $S_{X,Y}$.

For example, consider the class hierarchy H shown in Figure 2 of Section 2. Suppose $S_{X,Y}$ is a node split of H such that $Class(X) = \{A, C\}$ and $Class(Y) = \{C, F, G\}$. Then C is the only shared class of $S_{X,Y}$ and among the 9 possible class scopes in H, only two of them are shared class scopes: $\{C\}$ and $Class(H_A)$.

A key range $[r_1, r_2] \subseteq [A_{min}, A_{max}]$ is said to be a **shared key range** of a node split $S_{X,Y}$ iff (1) $[r_1, r_2] \cap KeyRange(X) \neq \emptyset$ and (2) $[r_1, r_2] \cap KeyRange(Y) \neq \emptyset$. There are two cases to consider:

(1) The key ranges of X and Y are disjoint; i.e., $KeyRange(X) \cap KeyRange(Y) = \emptyset$.

(2) The key ranges of X and Y overlap; i.e., $KeyRange(X) \cap KeyRange(Y) \neq \emptyset$.

Let $v_{min} = \min\{MaxKey(X), MaxKey(Y)\}$ and $v_{max} = \max\{MinKey(X), MinKey(Y)\}$. For case (1), $[r_1, r_2]$ is a shared key range iff $[r_1, r_2]$ contains the interval between the two disjoint key ranges which is $[v_{min}, v_{max}]$. Thus, $[r_1, r_2]$ is a shared key range iff $(A_{min} \leq r_1 \leq v_{min})$ and $(v_{max} \leq r_2 \leq A_{max})$. For case (2), $[r_1, r_2]$ is a shared key range iff $[r_1, r_2]$ overlaps with the the interval where the two key ranges overlap; i.e., $KeyRange(X) \cap KeyRange(Y)$ which is $[v_{max}, v_{min}]$. Thus, $[r_1, r_2]$ is a shared key range iff either (1) $(A_{min} \leq r_1 < v_{max} \leq r_2 \leq A_{max})$, or (2) $(v_{max} \leq r_1 \leq v_{min})$ and $(r_1 \leq r_2 \leq A_{max})$.

We shall use $SharedScope(S_{X,Y})$ to denote the set of class scopes of H each of which is a shared class scope of $S_{X,Y}$, and $SharedRange(S_{X,Y})$ to denote the set of key ranges of A each of which is a shared key range of $S_{X,Y}$. That is,

$$SharedScope(S_{X,Y}) = \{CS | CS \text{ is a class scope of } H, CS \cap Class(X) \neq \emptyset, CS \cap Class(Y) \neq \emptyset\},$$

and

$$SharedRange(S_{X,Y}) = \begin{cases} \{[r_1, r_2] | A_{min} \leq r_1 \leq v_{min}, v_{max} \leq r_2 \leq A_{max}\} & \text{if case (1),} \\ \{[r_1, r_2] | A_{min} \leq r_1 < v_{max} \leq r_2 \leq A_{max}\} \cup & \\ \{[r_1, r_2] | v_{max} \leq r_1 \leq v_{min}, r_1 \leq r_2 \leq A_{max}\} & \text{if case (2)} \end{cases}$$

6.2 A Metric to Evaluate the "Goodness" of Node Splits

We introduce the notion of *access proximity* as our metric to measure the "goodness" of a node split. The **access proximity** of a node split $S_{X,Y}$, denoted by $Proximity(S_{X,Y})$, is defined as the probability that both the split nodes of $S_{X,Y}$ need to be accessed. Thus, given two node splits $S_{X,Y}$ and $S_{X',Y'}$, $S_{X,Y}$ is considered to be a better node split than $S_{X',Y'}$ iff $Proximity(S_{X,Y}) < Proximity(S_{X',Y'})$.

Let Q be the set of all possible queries on a class hierarchy H and on an attribute A. Let p be the query distribution such that p_i is the probability that query $q_i \in Q$ is asked. Since a split node, say X, of a node split $S_{X,Y}$, needs to be accessed by a query q only when the subspace enclosed by X (i.e., $Class(X)$ and $KeyRange(X)$) intersects with the search space specified by q (i.e., its query class scope and query range), therefore,

$$\begin{aligned} Proximity(S_{X,Y}) &= \text{Prob(Both the split nodes of } S_{X,Y} \text{ need to be accessed)} \\ &= \text{Prob}(Class(X) \cap qc \neq \emptyset, KeyRange(X) \cap [qr_1, qr_2] \neq \emptyset, \end{aligned}$$

---

[1] Note that $Class(H_C) = \{C\}$ if C is a leaf class of the indexed class hierarchy.

$$
\begin{aligned}
& \quad Class(Y) \cap qc \neq \emptyset, KeyRange(Y) \cap [qr_1, qr_2] \neq \emptyset) \\
&= \text{Prob}(qc \in SharedScope(S_{X,Y}) \text{ and } [qr_1, qr_2] \in SharedRange(S_{X,Y})) \\
&= \sum_{q_i \in Q'} p_i \quad \text{where} \qquad (1) \\
& \quad Q' = \{q \in Q | qc \in SharedScope(S_{X,Y}), [qr_1, qr_2] \in SharedRange(S_{X,Y})\}.
\end{aligned}
$$

However, since information about the query distribution is usually not available, we shall assume that the query distribution is uniformly distributed. In fact, the uniform query distribution assumption forms the basis of the node splitting heuristics of multi-dimensional access methods such as the R-tree [11] and R^*-tree [2]. However, the definition of our access proximity metric can exploit information about the query distribution in the evaluation of a node split.

We shall now elaborate on the computation of the metric for the case when the query distribution is uniformly distributed as this is the most likely situation due to the difficulty of obtaining the query distribution. We shall assume the domain of the indexed attribute A to be discrete. Let Q_c denote the set of all distinct class scopes of H, and Q_r denote the set of all distinct key ranges of A. So $Q = \{q | qc \in Q_c, [qr_1, qr_2] \in Q_r\}$ and $|Q| = |Q_c| \times |Q_r|$. From Section 2, $|Q_c| = 2n_i + n_l$ where n_i and n_l are the total number of internal and leaf classes in H, respectively. Since the query distribution is uniformly distributed, therefore $p_i = \frac{1}{|Q|}, \forall q_i \in Q$. From Equation (1),

$$
Proximity\,(S_{X,Y}) = \frac{|Q'|}{|Q|} = \frac{|SharedScope(S_{X,Y})| \times |SharedRange(S_{X,Y})|}{|Q|}. \qquad (2)
$$

Since the domain of the indexed attribute A is discrete, $|SharedRange(S_{X,Y})|$ is finite and can be easily computed. $|SharedScope(S_{X,Y})|$ is bounded by $[1, 2n_i + n_l]$ can be efficiently computed by a bottom-up traversal of the indexed class hierarchy.

6.3 Heuristics to Generate Node Splits

As the search space for an optimal node split is exponentially large, efficient heuristics are necessary to prune the search space. This section describes the heuristics that we developed to generate a small set of candidate node splits from which the "best" node split is selected based on the access proximity metric presented in the previous section.

To obtain a near optimal node split, the heuristics used should generate candidate node splits that have a low access proximity. From Equation (1), it can be inferred that the access proximity of a node split is likely to be low if its set of shared class scopes, $SharedScope(S_{X,Y})$, and/or its set of shared key ranges, $SharedRange(S_{X,Y})$, is small. We shall refer to a node split that has a small set of shared class scopes as a **class-split** and a node split that has a small set of shared key ranges as a **key-split**. To keep the heuristics simple and efficient, our strategy to generate good candidate node splits is divided into two steps: the first step generates a set of class-splits, and the second step generates a set of key-splits. The best node split, based on the access proximity metric, is then selected from these two sets of node splits. The overall node splitting strategy is shown in the **splitNode** algorithm. The following two subsections describe the heuristics to generate class-splits and key-splits, respectively.

6.3.1 Heuristics to Generate Class-Splits

A class-split is a node split with a small set of shared class scopes $SharedScope(S_{X,Y})$. To generate class-splits from a set of entries S, we need to partition the entries in S based on the classes of the entries. Let $S[C]$ denote the set of entries in S that are of class C. For a set of leaf node entries S, $S[C] = \{s \in S | s.\text{cid} = C\}$. Clearly, $\{S[C] | S[C] \neq \emptyset\}$ defines a partition on S. This definition of $S[C]$, however, cannot be directly applied to the case where S is a set of internal node entries because each internal node entry is associated with a set of classes, and not a single unique class. Instead, we shall associate a set of classes to its youngest superclass which is unique.

Given a set of classes T, let $YSC(T)$ denote the youngest superclass of T; i.e., the class subhierarchy rooted at the class $YSC(T)$ is the smallest class hierarchy that contains all the

Algorithm SplitNode

SplitNode (T, N, N_{new})

Input: T is a pointer to the root node of the χ-tree .
N is the overflowed node to be split.
Output: A pair of nodes (N, N_{new}) representing the split nodes.

$R = \{\}$; /* initialize the set of candidate node splits */
Invoke **GenerateClassSplits** (N, R) to a generate a set of admissible class-splits.
Invoke **GenerateKeySplits** (N, R) to a generate a set of admissible key-splits.
From among R, let (N_1, N_2) be the node split with the minimum access proximity;
$N = N_1$; $N_{new} = N_2$; /* return the split nodes */

---

classes in T. Formally, $YSC(T) \in Class(H)$ such that $T \subseteq Class(H_{YSC(T)})$ and $\not\exists C' \in Class(H_{YSC(T)})$ such that $C' \neq YSC(T)$ and $T \subseteq Class(H_{C'})$. The notion of a youngest superclass of a set of classes provides a unique representation of a set of classes. So for a set of internal node entries S, let $S[C] = \{s \in S | YSC(s.cidSet) = C\}$. It is evident that $\{S[C] | S[C] \neq \emptyset\}$ defines a partition on S.

The definition of $S[C]$ can be generalized to a class subhierarchy as follows. Let $S[H_C]$ denote the set of entries in S that belong to any one of the classes in H_C; i.e., if S is a set of leaf node entries, then $S[H_C] = \{s \in S | s.\text{cid} \in Class(H_C)\}$; otherwise, if S is a set of leaf node entries, then $S[H_C] = \{s \in S | YSC(s.cidSet) \in Class(H_C)\}$. Note that $S[H_C]$ does not define a partition on S.

Our heuristics to generate class-splits are based on the two properties presented in Section 6.1:

(P1) If C is a shared class of $S_{X,Y}$, then both $\{C\}$ and $Class(H_C)$ are shared class scopes of $S_{X,Y}$.

(P2) If $Class(H_C)$ is a shared class scope of $S_{X,Y}$ and C' is a superclass of C, then $Class(H_{C'})$ is also a shared class scope of $S_{X,Y}$.

Recall that it is desirable for a node split to have a small set of shared class scopes as having a large set of shared class scopes increases the likelihood that a query needs to access both split nodes of the split $S_{X,Y}$. Therefore by (P1), it is desirable to minimize the number of shared classes of $S_{X,Y}$ since having more shared classes increases the number of shared class scopes. By (P2), it is desirable for a node split to avoid shared class scope $Class(H_C)$ where C is deeply nested in the indexed class hierarchy since it induces more shared class scopes than one with C being less deeply nested. Based on these observations, we propose the following two heuristics for generating candidate class-splits.

Heuristic 1: Minimize number of shared classes.

By (P1), a good class-split should have a small number of shared classes. However, it is not always possible to generate node splits that have no shared classes, as a shared class is sometimes necessary in order for the node split to be admissible. Furthermore, even if such an admissible node split without shared classes exists, the search for it is a subset-sum problem which is NP-complete. The following proposition shows that it is always possible to refine a node split that has at least two shared classes into another node split with at least one less shared class and a smaller set of shared class scopes. This implies that it is always possible to refine an arbitrary admissible node split into one with at most one shared class.

Proposition 1 *Let $S_{X,Y}$ be an admissible node split such that $S_{X,Y}$ has at least two shared classes. Then there exists another admissible node split $S_{X',Y'}$ such that (1) $S_{X',Y'}$ has at least one less shared class than $S_{X,Y}$, and (2) $SharedScope(S_{X',Y'}) \subset SharedScope(S_{X,Y})$.*

Proof: Let C_1 and C_2 be two distinct shared classes of $S_{X,Y}$. So $\{C_1, C_2\} \subseteq SharedScope(S_{X,Y})$. It must be the case that either $|X[C_1]| \leq |Y[C_2]|$ or $|X[C_1]| > |Y[C_2]|$. Consider the first case where $|X[C_1]| \leq |Y[C_2]|$. Let $T \subseteq Y[C_2]$ such that $|T| = |X[C_1]|$. By swapping $X[C_1]$ and T, we obtain a new admissible node split where C_1 is not a shared class of the new node split. Formally, define a new node split $S_{X',Y'}$ such that (1) $X' = X - X[C_1] \cup T$, and (2) $Y' = Y - T \cup X[C_1]$. Since $|X'| = |X|$, $|Y'| = |Y|$, and $S_{X,Y}$ is an admissible node split, it follows that $S_{X',Y'}$ is also an admissible node split. Clearly, by definition of $S_{X',Y'}$, $SharedScope(S_{X',Y'}) \subseteq SharedScope(S_{X,Y})$. Since C_1 is a shared class of $S_{X,Y}$ but not of $S_{X',Y'}$, therefore $SharedScope(S_{X',Y'}) \subset SharedScope(S_{X,Y})$. By a similar argument, we can show the claim also holds for the second case where $|X[C_1]| > |Y[C_2]|$. $\Box$

Heuristic 2: Avoid shared class scope $Class(H_C)$ with deeply nested class C

By (P2), a good class-split should avoid shared class scopes H_C with C being deeply nested in the indexed class hierarchy so as to reduce the number of shared class scopes. For example, suppose the indexed class hierarchy is the class hierarchy shown in Figure 2 of Section 2. Consider an overflowed node with S as the set of entries in the overflowed node such that $Class(S) = \{C, D, E, G\}$. Consider the following two node splits, $S_{X,Y}$ and $S_{X',Y'}$ such that:

- $Class(X) = \{C, D, E\}$ and $Class(Y) = \{G\}$
- $Class(X') = \{C, E, G, \}$ and $Class(Y') = \{D\}$

Note that both node splits, $S_{X,Y}$ and $S_{X',Y'}$, do not have any shared classes. However, $S_{X,Y}$ has two shared class scopes ($Class(H_A)$ and $Class(H_B)$) while $S_{X',Y'}$ has only one shared class scope ($Class(H_A)$). Hence $S_{X',Y'}$ is considered to be a better class-split than $S_{X,Y}$.

Based on these two properties, the **GenerateClassSplits** algorithm uses heuristics to generate admissible class-splits with at most one shared class and with a minimal number of shared class scopes H_C that have deeply nested classes C.

Algorithm GenerateClassSplits

GenerateClassSplits (N,R)

Input: N is the overflowed node to be split.
Output: R is the set of candidate node splits.

$n_1 = \lceil f \times (|N| - 1) \rceil$; /* f is the node utilization factor */
$n_2 = |N| - 1 - n_1$; /* $[n_1, n_2]$ is a bound on the number of entries to be moved from N */
$S = \{\}$; /* initialize the set of classes in new split node */
$L = \{\}$; /* initialize the set of candidate class-splits */
invoke **SplitHierarchy** (H, N, n_1, n_2, S, L) where H is the indexed class hierarchy;
for each class-split (C_y, C_s) in L **do**
begin
    invoke **RefineSplit** $(H, N, n_1, n_2, C_y, C_s, X, Y)$;
    insert (X, Y) into R;
end

The **GenerateClassSplits** algorithm generates candidate class-splits in two phases. The first phase (**SplitHierarchy** algorithm), which is the main step of the algorithm, generates the candidate class-splits using the heuristics discussed. However, the candidate class-splits generated by the first phase are represented at a "coarser" level (representing distribution of classes of the node entries), which are then refined by the second phase (**RefineSplit** algorithm) into the actual class-splits representation (representing distribution of the node entries).

In the first phase, each of the candidate class-splits generated by the **SplitHierarchy** algorithm is represented by (C_y, C_s), where C_y represents the set of classes that are distributed to the new split node, and C_s represents the shared class of the node split, if any. That is, $Class(X) = Class(S) - C_y$ and $Class(Y) = C_y \cup \{C_s\}$. Thus, each of the generated candidate class-splits has at most one shared class. In the second phase, each (C_y, C_s) is refined by the **RefineSplit** algorithm into an actual node split (X, Y), where X and Y represent the set of node entries distributed to the two split nodes, respectively.

Algorithm SplitHierarchy

SplitHierarchy(H_R, N, n_1, n_2, S, L)

Input: H_R is the class hierarchy rooted at class R to be split.
N is the set of entries in the overflowed node.
$[n_1, n_2]$ is the bound on the number of entries to be moved to the new split node.
S is a set of classes to be moved to the new split node.
L is a set of candidate class-splits; each class-split is represented by (C_y, C_s) where
C_y is the set of classes in the new split node, and C_s is the shared class (if any).
Output: L is a set of class-splits.

if $(|N[H_R]| < n_1)$ **then**
  return ; /* terminate splitting */
else begin
  if $(|N[R]| \leq n_2)$ **then**
    add $(S \cup \{R\}, nil)$ to L; /* create a new class-split with R moved to the new split node */
  else
    add (S, R) to L; /* create a new class-split with R as a shared class */
end
for each immediate subclass C of R **do**
begin
  if $(|N[H_C]| > n_2)$ **then**
    invoke **splitHierarchy** (H_C, n_1, n_2, S, L); /* split class hierarchy H_C */
  else if $(|N[H_C]| \geq n_1)$ **then**
    add $(S \cup Class(H_C)\ , nil)$ to L; /* create a new class-split with H_C moved to new split node */
  else if $(|N[H_C]| > 0)$ **then**
  begin
    $S = S \cup Class(H_C)$; /* move H_C to new split node */
    let $H_R =$ **cutHier**(H_R, H_C); /* remove H_C from H_R */
    invoke **splitHierarchy** $(H_R, n_1 - |N[H_C]|, n_2 - |N[H_C]|, S, L)$;
    /* split modified class hierarchy H_R */
  end
end

The **SplitHierarchy** algorithm traverses the indexed class hierarchy top-down and tries to generate candidate node splits with a minimum number of shared class scopes. The algorithm first checks the number of node entries belonging to R, the root class of the class hierarchy H. If $|N[R]|$ falls within the bound $[n_1, n_2]$, a class-split is generated with class R distributed to the new split node; otherwise, if the size exceeds n_2, a class-split is generated with R as a shared class. The algorithm then examines the sub-hierarchy rooted at each immediate subclass to check for other potential class-splits. The function **cutHier**(H_1, H_2) accepts two class hierarchies where H_2 is a sub-hierarchy of H_1 and returns a class hierarchy which is equivalent to H_1 with the sub-hierarchy H_2 removed.

Algorithm RefineSplit

RefineSplit $(H, N, n_1, n_2, C_y, C_s, X, Y)$

Input: H is the indexed class hierarchy.
N is the overflowed node to be split.
$[n_1, n_2]$ is the bound on the number of entries in N to be moved over to the new split node.
(C_y, C_s) is a class-split where C_y is a set of classes distributed to the new split node, and C_s refers to a shared class (if any) or *nil*.
Output: (X, Y) is a node split of N.

$X = Y = \{\}$;
for each class $C \in Class(H)$ **do**
begin
  if $C \in C_y$ **then**
    $Y = Y \cup N[C]$; $N = N - N[C]$;
  else if $C \neq C_s$ **then**
    $X = X \cup N[C]$; $N = N - N[C]$;
end
if $C_s \neq nil$ **then**
  move $(n_1 - |X|)$ entries in N into X, and the remaining entries in N into Y;

6.3.2 Heuristics to Generate Key-Splits

In comparison to the splitting of the class dimension discussed in the previous section, the splitting of the key dimension is easier as the key dimension is linearly ordered. A node split is considered a key-split if it has a small set of shared key ranges. From the discussion of a shared key range in Section 6.1, a node split $S_{X,Y}$ is a key-split if either

(1) the key ranges of X and Y are disjoint and separated by a large interval, or

(2) the key ranges of X and Y overlap with a small overlap interval.

To partition the entries in S such that one of the above properties hold, the following ordering on the entries based on their key ranges is used. $\forall e_1, e_2 \in S$, if S is a set of leaf node entries, then e_1 precedes e_2 iff $e_1.key \leq e_2.key$; otherwise, if S is a set of internal node entries, then e_1 precedes e_2 iff either (a) $e_1.key_{min} < e_2.key_{min}$ or (b) ($e_1.key_{min} = e_2.key_{min}$) and ($e_1.key_{max} \leq e_2.key_{max}$).

Let $[n_1, n_2]$ be the bound on the number of node entries to be distributed to the new split node; this bound is computed from two parameters: the node utilization factor and the total number of entries in a full node. A set of $(n_2 - n_1 + 1)$ candidate key-splits can be generated such that for the i^{th} candidate key-split S_{X_i,Y_i}, X_i contains exactly $(n_1 + i - 1)$ node entries, and each entry in X_i precedes each entry in Y_i by the ordering defined. This set of key-splits can be computed efficiently using the algorithm for computing order statistics [1].

7 Update Algorithms

7.1 Insert Operation

To insert a new entry, the algorithm first selects a leaf node. If the selected leaf node overflows after the insertion, the overflowed node is split into two nodes by distributing the entries in it between a new node and itself. Finally, the ancestor nodes of the selected leaf node are updated to preserve property $I4$.

Algorithm Insert

Insert (T, E)

Input: T is a pointer to the root node of a χ-tree .
E is a new leaf node entry to be inserted into the χ-tree .

invoke **ChooseLeaf**(T, E, L) to select a leaf node (pointed by L) from the χ-tree ;
let N_L be the leaf node pointed to by L;
insert E into N_L;
if (N_L overflows) **then**
  invoke **SplitNode**(T, L, L_{new}) to split the entries in N_L
  into N_L and a new node pointed to by L_{new};
invoke **AdjustTree**(T, E, L, L_{new}) to update the internal node entries of the χ-tree ;

Algorithm ChooseLeaf

ChooseLeaf (T, E, L)

Input: T is a pointer to a node of a χ-tree .
E is a new leaf node entry to be inserted into χ-tree .
Output: L is a pointer to the selected leaf node in χ-tree to insert E.

let N be the node pointed to by T;
if (N is a leaf node) **then**
  let $L = T$;
else begin
  Among all the internal entries in N, find the entry E_N
    that has the maximum *access proximity* with E;
  invoke **ChooseLeaf**($E_N.P$,E,L);
end

The **ChooseLeaf** algorithm searches the χ-tree top-down, from the root to the leaf level to look for the leaf node to insert a new entry (T is first initialized to point to the root node). At each level, one node is visited and selection of the next node to visit is based on the *access proximity* criterion, which will be introduced in Section 6 together with the **SplitNode** algorithm. Note that the node splitting criterion has a great impact on the performance of the index method.

The **AdjustTree** algorithm traverses the χ-tree bottom-up from the leaf node (with a newly inserted entry), along the path formed by its ancestor nodes, up to the root node. Each internal node visited along the path is updated to account for the changes in its child node. If the child node has split, a new internal node entry is created and added to its parent node, which may in turn cause the parent node to overflow. A new root node is created if the node split propagates up to the root level.

7.2 Delete Operation

To remove a leaf node entry from a χ-tree , the deletion algorithm first traverses the χ-tree top-down to search for the leaf node containing the to-be-deleted leaf node entry. The leaf node entry is then removed from the leaf node. If the leaf node underflows after the deletion, then all the entries in the underflowed node are relocated and the leaf node is eliminated. The node elimination is propagated upwards as necessary with the internal nodes along the path to the root being adjusted to preserve property $I4$. Due to space constraint, the details of the deletion algorithm are not shown

Algorithm AdjustTree

AdjustTree (T, L, L_{new})

Input: T is a pointer to the root node of the χ-tree .
L is a pointer to a leaf node which has been inserted with a new entry.
L_{new} is a pointer to a new split node if the insertion into the node pointed to by L has caused a node split; otherwise, L_{new} is *nil*.
Output: T is updated to point to a new root node if the current root node overflows.

while $(L \neq T)$ **do**
begin
  let N_L be the node pointed to by L;
  let N_P be the parent node of N_L, and P be the pointer to N_P;
  let E_N be the internal entry in N_P such that $E_N.P = L$;
  update E_N to account for the new entry that has been inserted into N_L;
  if $(L_{new} \neq nil)$ **then**
  begin
    let N_{new} be the node pointed to by L_{new};
    create a new internal node entry $E_{N_{new}}$, corresponding to N_{new};
    insert $E_{N_{new}}$ into N_P;
    if (N_P overflows) **then**
      invoke **SplitNode** (T, P, P_{new}) to split the entries in the parent node N_P into N_P and a new node pointed to by P_{new};
  end else
    $P_{new} = nil$;
  let $L = P$ and $L_{new} = P_{new}$;
end
if $(L_{new} \neq nil)$ **then** /* L is now pointing to the root node */
begin
  create a new internal node R and two new internal node entries, E_N and $E_{N_{new}}$, corresponding to the nodes pointed to by L and L_{new}, respectively;
  insert E_N and $E_{N_{new}}$ into R;
  update T to point to the new root node R;
end

here. However, as the deletion algorithm is similar to that for the R-tree, the reader may refer to [11].

8 Performance Study

This section compares the performance of the χ-tree with Kim's CH-tree [16]. The implementation details of the two index methods are presented in Section 8.1. Section 8.2 describes the methodology of the performance study and Section 8.3 presents the experimental results.

8.1 Index Implementation

To compare the storage requirements of the CH-tree and χ-tree , we first elaborate on the structure of the CH-tree which is shown in Figure 5; the structure of the χ-tree is explained in Section 5.2. In the CH-tree , the leaf node entries are organized by the indexed attribute, entries with the same key value are grouped into the same record. Within each record, the *key-directory* groups the

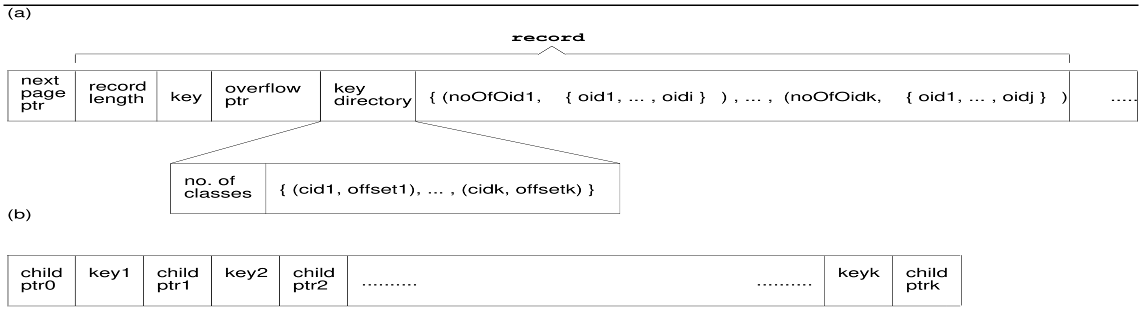

Figure 5: Structure of CH-tree (a) Leaf Node (b) Internal Node

entries by class. Like in the B^+-tree , the leaf nodes in the CH-tree are sequentially linked using the *next page pointer*. When a leaf node record grows beyond the size of the leaf node, some of the entries in the overflowed record are distributed into a new node which is linked from the overflowed node by the *overflow pointer*.

For both index methods, the node utilization factor is set to 0.5. The sizes of the various parameters of both index methods are shown in Table 1. Note that for the *cidSize* parameter, only one byte is allocated. Although the number of classes in an application domain can be large, they are partitioned into a number of class hierarchies, and for the majority of the applications, we expect that 256 classes is a reasonably large number of classes for each of the indexed class hierarchies; the number of bytes allocated can be increased for larger indexed class hierarchies. In order to map the one-byte internal class identifiers used in the index records of an indexed class hierarchy to their actual class identifiers defined in the application domain, each indexed class hierarchy is associated with a main memory resident lookup table. The compression of the class identifier codes in the index records improves not only the space utilization of the index but also its search performance.

Table 1: Parameter Values Used in Index Implementations.

Parameter	**Size (in bytes)**	**Meaning**
	Common Parameters	
$pageSize$	2048 or 4096	size of disk page
$recLenSize$	2	size of record-length field
$cidSize$	1	size of cid field
$oidSize$	8	size of oid field
$keySize$	4	size of key field
$ptrSize$	4	size of pointer field (includes child-ptr, overflow-ptr & next-page-ptr)
	χ-tree Parameters	
$cidSetSize$	1	size of cid-set field
$keyRangeSize$	8	size of key-range field
	CH-tree Parameters	
$numClassSize$	1	size of number-of-classes field
$offsetSize$	2	size of offset field
$numObjSize$	2	size of number-of-objects field

For the *classSetSize* parameter used in the χ-tree , only one byte is allocated; the *classSet* field is used as a bit vector to represent a subset of the classes in the indexed class hierarchy, with each bit corresponding to a particular class in the hierarchy. This representation suffices for indexed class hierarchies which do not exceed 8 classes. For larger indexed class hierarchies, there are two ways to handle the *classSet* field representation. The first method is to increase the storage allocation for the field. However, due to the splitting heuristic, it is likely that only a small subset of the bits are used for each internal node entry, resulting in poor utilization of the storage space. A better alternative approach is to partition the indexed class hierarchy into a number of smaller sub-hierarchies such that each sub-hierarchy is being indexed by a χ-tree with an appropriate size for the the *classSet* field, thereby improving the storage utilization.

To perform well, the second alternative requires a judicious partitioning of the indexed class hierarchy so as to reduce the need to traverse more than one sub-hierarchy index. This can be achieved using the same principles as for generating class-splits (Section 6.3.1) by minimizing the number of shared class scopes. Furthermore, it is also desirable for each of the sub-hierarchies to be of about the same size (in terms of the number of sizes). For example, consider the class hierarchy

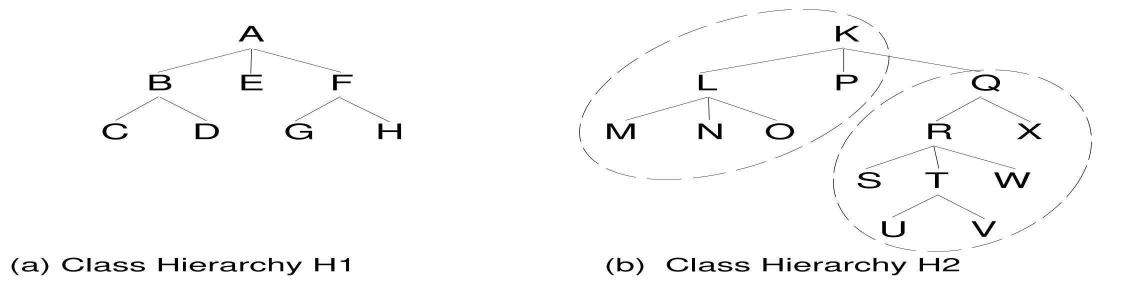

Figure 6: Class Hierarchies Used in Experiments

H2 shown in Figure 6(b) consisting of 14 classes. This can be partitioned into two smaller sub-hierarchies, each with no more than 8 classes, as indicated by the two dotted regions. Note that for this partition, there is only one shared class scope $Class(K)$ which means that only CCH-queries need to traverse both the sub-hierarchy indexes.

For the performance study, we have assumed that the domain of the indexed attribute is discrete and that the query distribution is not known. So for the χ-tree , the computation of the access proximity is based on a uniform query distribution and is computed as derived in Equation (2).

8.2 Methodology

Table 2 shows the parameters used in generating the index records and queries, and their values. For all the experiments, both the number of objects *numObj* and attribute domain *keyDom* parameters remain fixed. By varying four of the variable parameters as well as the *pageSize* parameter, a total of 248 experiments are performed. Each experiment corresponds to a different set of index records for a given page size, which is controlled by the *pageSize*, *classHier*, θ_1, and θ_2 parameters. The *pageSize* parameter controls the size of each index page. The *classHier* parameter determines the complexity of the class hierarchy, in terms of its size (number of classes) and structure (degree and number of levels). Figure 6 shows the two class hierarchies used in this study. The more complex class hierarchy, $H2$, is partitioned into two smaller sub-hierarchies as shown in Figure 6(b), with 6 and 8 classes respectively, so that the number of classes in each sub-hierarchy is no more than 8 classes; two χ-tree indexes are created for experiments using the class hierarchy $H2$.

The distribution of the objects in each class and the distribution of the key values are both varied using the Zipf's distribution [17] by varying their respective skew factors, θ_1 and θ_2. For the number of index records per class distribution, the Zipf's distribution is used such that a class gets more records than its superclass. For the number of index records per key value distribution, the key domain is partitioned into ten disjoint ranges, and the index records are then distributed among the ranges using the Zipf's distribution. Note that in the Zipf's distribution, the skew factor

Table 2: Parameter Values Used in Data and Query Generation.

Parameter	**Value**	**Meaning**
	`Fixed Parameters`	
numObj	$200,000$	`number of index records`
keyDom	`[1,1000]`	`domain of indexed attribute (key)`
	`Variable Parameters`	
classHier	$\{H1, H2\}$	`type of class hierarchy`
θ_1	$\{0, 0.5, 1\}$	`skew factor for object distribution`
θ_2	$\{0, 0.5, 1\}$	`skew factor for key distribution`
queryRange	$\{10\%, 20\%, 30\%\}$	`query range as a percentage of the key domain`
sizeClassScope	`[1, size of class hierarchy]`	`size of query's class scope`

$\theta \leq 1$ is such that a smaller θ value indicates a more skewed distribution. The Zipf's distribution approximates the uniform distribution when $\theta = 1$.

The set of queries are generated using the parameters *queryRange* and *sizeClassScope* which correspond to the two parameters that characterize each query: (1) query range and (2) class scope. The *queryRange* parameter controls the percentage of the key domain covered by the query. The *sizeClassScope* parameter determines the number of classes in the query's class scope and is dependent on the *classHier* parameter. Each unique pair of *queryRange* and *sizeClassScope* parameter values forms a unique *query category* and 20 queries are randomly generated for each category.

For each experiment, a χ-tree and a CH-tree are constructed using the same set of index records (i.e. for a given class hierarchy and skew factors). The size of each index structure is recorded. The set of search queries are then run against both index structures and the average number of page access for each query category are recorded for the χ-tree and CH-tree .

8.3 Experimental Results

This section presents the experimental results comparing both the space requirement and search performance of the χ-tree and the CH-tree .

8.3.1 Comparison of Space Requirement

Table 3 compares the space requirement of both indexes in terms of the ratio of the total number of index pages of the χ-tree to that of the CH-tree (denoted by δ) for different values of *pageSize* and *classHier*.

As expected, the χ-tree incurs a higher space requirement than the CH-tree . This is partly due to the lower fanout of the internal node in the χ-tree : the *key-range* field in a χ-tree internal node record is double the size of the *key* field in a CH-tree internal node record. Furthermore, while the CH-tree 's internal node records are fixed-length records, the χ-tree 's internal node records are variable length records each with an additional *record-length* field.

The values of δ are generally smaller for class hierarchy $H2$ compared to class hierarchy $H1$. This is probably due to the relatively greater increase in the storage overhead incurred by the CH-tree internal node records when it is used to index a more complex class hierarchy (e.g. the size of the key directory increases with the number of classes).

Table 3: Comparison of Space Requirement.

(a) $pageSize = 2048$, $classHier = H1$

θ_1	θ_2	χ-tree			CH-tree			
		No. of Internal Pages	**No. of Leaf Pages**	**Total No. of Pages** (T_{xtree})	**No. of Internal Pages**	**No. of Leaf Pages**	**Total No. of Pages** (T_{chtree})	$\delta = \frac{T_{xtree}}{T_{chtree}}$
0.0	0.0	16	1802	1818	5	1074	1079	1.69
	0.5	16	1721	1737	5	1189	1194	1.46
	1.0	17	1702	1719	6	1000	1006	1.71
0.5	0.0	17	1719	1736	5	1077	1082	1.61
	0.5	16	1720	1736	5	1190	1195	1.46
	1.0	15	1707	1722	6	1000	1006	1.72
1.0	0.0	16	1725	1741	5	1077	1082	1.61
	0.5	16	1704	1720	5	1191	1196	1.44
	1.0	14	1717	1731	6	1000	1006	1.73

(b) $pageSize = 2048$, $classHier = H2$

θ_1	θ_2	χ-tree			CH-tree			
		No. of Internal Pages	**No. of Leaf Pages**	**Total No. of Pages** (T_{xtree})	**No. of Internal Pages**	**No. of Leaf Pages**	**Total No. of Pages** (T_{chtree})	$\delta = \frac{T_{xtree}}{T_{chtree}}$
0.0	0.0	19	1768	1787	5	1098	1103	1.63
	0.5	17	1715	1732	6	1213	1219	1.43
	1.0	17	1737	1754	7	1000	1007	1.75
0.5	0.0	18	1730	1748	5	1088	1093	1.60
	0.5	18	1719	1737	7	1215	1222	1.43
	1.0	18	1713	1731	7	1001	1008	1.72
1.0	0.0	16	1706	1722	5	1088	1093	1.58
	0.5	17	1723	1740	6	1206	1212	1.44
	1.0	19	1704	1723	7	1000	1007	1.72

8.3.2 Comparison of Search Performance

Table 4 compares the search performance of the χ-tree and CH-tree in terms of the ratio of the number of disk page accessed using the CH-tree to that using the χ-tree , given by $\frac{T_{chtree}}{T_{xtree}}$ where T_{chtree} and T_{xtree} denote the total number of pages accessed for the CH-tree and χ-tree , respectively. The performance comparison clearly shows that the χ-tree outperforms the CH-tree for majority of the cases, with the performance margin getting smaller as the size of the query class scope increases. For CCH-queries, the CH-tree outperforms the χ-tree , as indicated by $\frac{T_{chtree}}{T_{xtree}} < 1$ for the last column of values in Table 4. This is expected since when the query class scope includes every class in the indexed class hierarchy, the CH-tree , which partitions the data space solely on the indexed attribute, provides very good clustering as the search space becomes effectively single dimensional.

The effect of the complexity of the class hierarchy on the search performance is that the performance gain is greater for a more complex class hierarchy: the ratio $\frac{T_{chtree}}{T_{xtree}}$ is higher for class

Table 3: Comparison of Space Requirement (cont.)

(c) $pageSize = 4096$, $classHier = H1$

θ_1	θ_2	χ-tree No. of Internal Pages	χ-tree No. of Leaf Pages	χ-tree Total No. of Pages (T_{xtree})	CH-tree No. of Internal Pages	CH-tree No. of Leaf Pages	CH-tree Total No. of Pages (T_{chtree})	$\delta = \frac{T_{xtree}}{T_{chtree}}$
0.0	0.0	5	870	875	1	589	590	1.49
	0.5	5	871	876	3	595	598	1.47
	1.0	4	838	842	3	543	546	1.55
0.5	0.0	5	878	883	1	588	589	1.50
	0.5	5	853	858	3	592	595	1.45
	1.0	4	851	855	3	543	546	1.57
1.0	0.0	5	872	877	1	588	589	1.49
	0.5	5	850	855	3	593	596	1.44
	1.0	5	896	901	3	543	546	1.66

(d) $pageSize = 4096$, $classHier = H2$

θ_1	θ_2	χ-tree No. of Internal Pages	χ-tree No. of Leaf Pages	χ-tree Total No. of Pages (T_{xtree})	CH-tree No. of Internal Pages	CH-tree No. of Leaf Pages	CH-tree Total No. of Pages (T_{chtree})	$\delta = \frac{T_{xtree}}{T_{chtree}}$
0.0	0.0	7	903	910	1	590	591	1.54
	0.5	6	889	895	3	602	605	1.48
	1.0	6	867	873	3	543	546	1.60
0.5	0.0	6	871	877	1	592	593	1.48
	0.5	6	850	856	3	602	605	1.42
	1.0	6	864	870	3	543	546	1.60
1.0	0.0	6	876	882	1	593	594	1.49
	0.5	6	870	876	3	602	605	1.45
	1.0	6	871	877	3	543	546	1.61

hierarchy $H2$ than $H1$. This is due to the combined effect of the clustering property of both the index methods: for the χ-tree , the increased complexity of the class hierarchy, in terms of the number of classes and their relationships, enables a finer-grained clustering of the data; on the other hand, since the CH-tree clusters data based only on the attribute dimension, the clustering effect in the CH-tree becomes weaker with more classes in the complex class hierarchy H2.

In general, the performance margin increases as the percentage of the query range increases. As more data has to be retrieved for a larger percentage of the query range, the effect of the poor clustering of the data in the CH-tree is amplified resulting in more pages being accessed.

The impact of the skewness of the object distribution θ_1 on the performance is that the performance margin decreases as the distribution becomes less skewed; i.e., as the value of θ_1 becomes larger. For the impact of the skewness of the key distribution θ_2, the performance gain is generally the highest when $\theta_2 = 0.5$ and lowest when $\theta_2 = 1.0$. The page size does not seem to have a significant impact on the performance margin.

Table 4: Comparison of Search Performance, $\frac{t_{chtree}}{t_{xtree}}$.

(a) $pageSize$ = 2048, $classHier = H1$

		Query	Size of Query Class Scope		
θ_1	θ_2	Range (%)	1	3	8
0.0	0.0	10	6.83	1.88	0.55
		20	6.47	2.10	0.62
		30	7.15	1.86	0.65
	0.5	10	7.27	2.33	0.61
		20	7.57	2.42	0.64
		30	8.39	2.08	0.67
	1.0	10	6.32	1.89	0.53
		20	6.81	1.97	0.56
		30	7.11	2.08	0.57
0.5	0.0	10	4.85	1.69	0.56
		20	5.14	1.80	0.62
		30	5.24	1.78	0.65
	0.5	10	5.84	1.92	0.60
		20	5.69	1.98	0.63
		30	6.04	1.87	0.67
	1.0	10	4.93	1.60	0.53
		20	5.03	1.66	0.56
		30	5.37	1.72	0.57
1.0	0.0	10	4.33	1.62	0.56
		20	4.53	1.62	0.62
		30	4.58	1.79	0.65
	0.5	10	4.75	1.78	0.61
		20	4.89	1.77	0.64
		30	4.96	1.84	0.68
	1.0	10	3.91	1.43	0.53
		20	4.04	1.48	0.55
		30	4.19	1.50	0.57

9 Conclusions

In this paper, we have proposed a new class-hierarchy indexing method, the χ-tree , and evaluated its storage requirement and search performance by comparing it with Kim's CH-tree . The χ-tree is a novel approach to class-hierarchy indexing which organizes the data as a two-dimensional space, in contrast to existing approaches which treat the data space as one-dimensional. As the class-hierarchy indexing problem is inherently two-dimensional, this new approach should provide better clustering of the index records over single-dimensional approaches and therefore provide better overall search performance for various types of queries with different class scopes and key ranges. Furthermore, being a multi-dimensional access method, the χ-tree is more adaptable than existing approaches to support class-hierarchy indexing for attributes which are multi-dimensional.

The χ-tree differs from existing multi-dimensional access methods in the linearity assumption made by the latter methods. Existing multi-dimensional access methods are designed for applications where each of the indexed dimensions is linearly ordered which simplifies node splitting. However, for the class-hierarchy indexing problem, the class dimension is hierarchically structured. As an arbitrary linearization of the class dimension will not provide effective clustering of the indexed records, a new node splitting strategy, which is based on the notion of proximity access, is developed for the χ-tree . This strategy can exploit knowledge about the query distribution to fine

Table 4: Comparison of Search Performance, $\frac{t_{chtree}}{t_{xtree}}$ (cont.)

(b) $pageSize = 2048$, $classHier = H2$

θ_1	θ_2	Query Range (%)	Size of Query Class Scope 1	3	4	6	8	14
0.0	0.0	10	8.98	5.13	2.69	2.09	1.18	0.56
		20	9.84	6.39	2.63	2.76	1.37	0.60
		30	11.88	4.55	2.51	2.32	1.38	0.65
	0.5	10	10.50	5.88	3.01	2.54	1.39	0.64
		20	11.30	6.86	3.07	3.12	1.55	0.67
		30	14.31	5.96	3.00	2.82	1.54	0.65
	1.0	10	9.24	4.78	2.40	2.24	1.23	0.52
		20	10.01	5.50	2.63	2.49	1.32	0.54
		30	12.06	5.77	2.81	2.63	1.36	0.55
0.5	0.0	10	7.38	3.19	2.05	1.46	1.01	0.56
		20	8.51	3.59	2.16	1.66	1.15	0.60
		30	8.75	3.04	2.36	1.59	1.18	0.65
	0.5	10	8.22	3.75	2.34	1.70	1.16	0.63
		20	8.92	4.26	2.35	2.05	1.27	0.67
		30	10.14	3.58	2.29	1.84	1.30	0.65
	1.0	10	7.12	3.02	1.99	1.52	1.02	0.52
		20	7.95	3.40	2.12	1.69	1.09	0.55
		30	8.92	3.50	2.13	1.73	1.13	0.56
1.0	0.0	10	6.71	2.51	1.95	1.27	0.97	0.56
		20	7.06	2.65	2.16	1.33	1.06	0.60
		30	7.75	2.92	2.33	1.46	1.11	0.65
	0.5	10	7.00	2.67	2.05	1.36	1.07	0.62
		20	7.58	2.80	2.11	1.45	1.15	0.66
		30	8.21	2.95	2.19	1.50	1.19	0.64
	1.0	10	6.11	2.26	1.78	1.18	0.90	0.52
		20	6.53	2.46	1.87	1.25	0.96	0.55
		30	7.06	2.58	1.92	1.31	0.99	0.56

tune its splitting, thereby improving its clustering and search performance. Performance experimental results show the although the χ-tree requires more storage requirement than the CH-tree , its search performance is better than that of the CH-tree in majority of the cases.

An interesting direction for future work is to extend the χ-tree to become an integrated index [3, 6] that supports both class-hierarchy as well as aggregation-hierarchy indexing.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference*, pages 322–331, Atlantic City, NJ, May 1990.

[3] E. Bertino. An indexing technique for object-oriented databases. In *Proceedings of the 7th International Conference on Data Engineering*, pages 160–170, 1991.

[4] E. Bertino. Method precomputation in object-oriented databases. *SIGOIS Bulletin*, 12(2):199–212, 1991. Proceedings of the ACM-SIGOIS and IEEE-TC-OA International Conference on Organizational Computing Systems, Atlanta, Georgia, November, 1991.

Table 4: Comparison of Search Performance, $\frac{t_{chtree}}{t_{xtree}}$ (cont.)

(c) $pageSize = 4096$, $classHier = H1$

θ_1	θ_2	**Query Range (%)**	**Size of Query Class Scope** 1	3	8
0.0	0.0	10	5.21	1.69	0.58
		20	5.84	2.02	0.62
		30	6.83	1.73	0.65
	0.5	10	5.76	1.94	0.59
		20	6.26	2.21	0.61
		30	7.53	1.93	0.65
	1.0	10	5.48	1.76	0.57
		20	6.19	1.98	0.61
		30	7.06	2.15	0.62
0.5	0.0	10	4.13	1.57	0.57
		20	4.64	1.70	0.61
		30	5.08	1.63	0.63
	0.5	10	4.86	1.84	0.61
		20	5.11	1.97	0.65
		30	5.64	1.79	0.66
	1.0	10	4.60	1.65	0.57
		20	5.06	1.76	0.60
		30	5.58	1.86	0.62
1.0	0.0	10	3.72	1.53	0.57
		20	4.12	1.59	0.61
		30	4.50	1.63	0.63
	0.5	10	4.05	1.72	0.61
		20	4.48	1.75	0.65
		30	4.80	1.76	0.67
	1.0	10	3.60	1.42	0.54
		20	3.99	1.47	0.57
		30	4.20	1.54	0.59

[5] E. Bertino. A survey of indexing techniques for object-oriented databases. In J.C. Freytag, D. Maier, and G. Vossen, editors, *Query Processing for Advanced Database Systems*, chapter 13, pages 384–418. Morgan Kaufmann Publishers Inc., 1994.

[6] E. Bertino and P. Foscoli. Index organizations for object oriented database systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(2):193–209, April 1995.

[7] E. Bertino and C. Guglielmina. Path-index: An approach to the efficient execution of object-oriented queries. *Data and Knowledge Engineering*, 10:1–27, 1993.

[8] 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.

[9] E. Bertino and A. Quarati. An approach to support method invocations in object-oriented queries. In *Proceedings of the International Workshop on Research Issues in Transactions and Query Processing (RIDE-TQP)*, Phoenix, AZ, February 1992.

[10] D. Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2):121–138, 1979.

[11] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference*, pages 47–57, Boston, MA, 1984.

[12] A. Jhingran. Precomputation in a complex object environment. In *Proceedings of the 7th International Conference on Data Engineering*, pages 652–659, 1991.

Table 4: Comparison of Search Performance, $\frac{t_{chtree}}{t_{xtree}}$ (cont.)

(d) $pageSize = 4096$, $classHier = H2$

θ_1	θ_2	**Query Range (%)**	**Size of Query Class Scope** 1	3	4	6	8	14
0.0	0.0	10	6.50	3.76	2.06	1.83	1.00	0.53
		20	8.72	5.07	2.05	2.59	1.32	0.59
		30	10.08	3.71	2.04	2.20	1.32	0.60
	0.5	10	7.47	4.41	2.26	2.11	1.19	0.55
		20	10.02	5.54	2.48	2.86	1.44	0.62
		30	11.75	5.29	2.57	2.63	1.45	0.62
	1.0	10	6.97	3.92	2.01	2.09	1.21	0.51
		20	9.95	4.80	2.35	2.46	1.37	0.57
		30	10.99	5.58	2.61	2.67	1.43	0.58
0.5	0.0	10	5.71	2.53	1.72	1.32	0.86	0.53
		20	7.72	3.08	1.81	1.60	1.12	0.60
		30	8.09	2.59	2.06	1.50	1.12	0.60
	0.5	10	6.43	3.10	1.87	1.58	1.09	0.57
		20	8.37	3.64	2.07	1.96	1.28	0.64
		30	8.87	3.37	2.13	1.82	1.29	0.64
	1.0	10	5.89	2.76	1.72	1.54	1.05	0.51
		20	8.04	3.21	1.98	1.71	1.17	0.57
		30	8.40	3.58	2.14	1.84	1.20	0.59
1.0	0.0	10	5.36	2.02	1.62	1.16	0.83	0.53
		20	6.65	2.36	1.79	1.29	1.04	0.59
		30	7.24	2.48	2.02	1.38	1.06	0.60
	0.5	10	5.67	2.36	1.64	1.27	0.99	0.55
		20	7.04	2.58	1.86	1.39	1.12	0.62
		30	7.41	2.69	2.05	1.44	1.14	0.62
	1.0	10	5.21	2.13	1.59	1.18	0.92	0.51
		20	6.72	2.37	1.79	1.27	0.99	0.56
		30	6.91	2.61	1.91	1.36	1.02	0.58

[13] A. Kemper, C. Kilger, and G. Moerkotte. Function materialization in object bases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 258–267, Denver, Colorado, 1991.

[14] A. Kemper and G. Moerkotte. Access support in object bases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 364–374, Atlantic City, N.J., 1990.

[15] W. Kim. *Introduction to Object-Oriented Databases.* MIT Press Series on Computer Systems. The MIT Press, 1990.

[16] W. Kim, K.-C. Kim, and A. Dale. Indexing techniques for object-oriented databases. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, ACM Press Frontier Series, chapter 15, pages 371–394. Addison-Wesley, New York, 1989.

[17] D.E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, 2 edition, 1973.

[18] B. Lomet, D. abd Salzberg. The hb-tree: A multi-attribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, December 1990.

[19] C.C. Low, B.C. Ooi, and H. Lu. H-tree – a dynamic associative search index for oodb. In *Proceedings of the ACM SIGMOD International Conference*, pages 134–143, San Diego, California, 1992.

[20] D. Maier and J. Stein. Indexing in an object-oriented dbms. In *Proceedings of International Workshop on Object-Oriented Database Systems*, pages 171–182, Asilomar, California, September 1986. Also in

On Object-Oriented Database Systems, edited by Dittrich, K.R., Dayal, U. and Buchmann, A.P., Chapter 20, Springer-Verlag, 1991.

[21] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[22] B. Seeger and H.-P. Kriegel. The buddy-tree: An efficient and robust access method for spatial data base systems. In *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[23] E. Shekita and M. Carey. Performance enhancement through replication in an object-oriented dbms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 325–336, Portland, Oregon, 1989.

[24] B. Sreenath and S. Seshadri. The hcc-tree: An efficient index structure for object oriented databases. In *Proceedings of the* 20^{th} *International Conference on Very Large Data Bases*, pages 203–213, Santiago, Chile, September 1994.

[25] P. Valduriez, S. Khoshafian, and G. Copeland. Implementation techniques for complex objects. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 101–110, Kyoto, Japan, 1986.