Triple-Node Hierarchies for Object-Oriented Database Indexing

Frank Hing-Wah Luk

and

Ada Waichee Fu [1]

Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, Hong Kong
Email: adafu@cse.cuhk.edu.hk

April 29, 1998

Abstract

An indexing structure called *triple-node hierarchy* is proposed for enhancing query processing in object-oriented database systems. The proposed structure provides efficient support for object references along an aggregation hierarchy by maintaining direct mapping between objects of interested pairs of classes. The intermediate classes along the object path are maintained separately for update purpose. We show that the proposed structure can achieve better performance compared to the previously known methods. The superior performance is also demonstrated by a set of simulations based on a cost model that we have developed. With some modification, the proposed structure can also provide fast support for object navigation in an integration of aggregation and inheritance hierarchies. Our results show that the extended triple-node hierarchy performs better than the best previous method known to us.

keywords: object-oriented databases, indexing, aggregation, inheritance, performance analysis, query processing.

[1] This author was partially supported by the RGC (the Hong Kong Research Grants Council) grant UGC REF.CUHK 495/95E.

1 Introduction

Object-oriented data model is a data model that provides a richer semantics than the traditional relational model for database systems. It can directly model nested relationships among data types, which can be organized as aggregation or inheritance hierarchies. With such a model, the query language can be more declarative and hence the application development can be simplified.

One major issue of any database system is the performance of query and update processing. Indexing techniques have been found to be essential for a guarantee of performance. Object-oriented database systems require a different set of indexing methods because of the complex structures of aggregation and inheritance. Known indexing techniques for object-oriented databases include the following types:

1. Structural
   (a) For aggregation hierarchy:
   The value of an attribute of an object is an object or a set of objects. The access scope of a query may include the nested attributes, which is essentially object accesses along references chains from one object instance to another. Methods for this support of nested predicates can be found in [20, 6, 13, 14, 24, 10, 11, 8].
   (b) For inheritance hierarchy:
   The access scope of a query against a class may include instances of the class and those of its subclasses. An index structure can support both instances in the same search index. Previous work includes [17, 1, 18, 19, 16, 22, 21],
2. Behavioral
   For OODB, queries may contain method invocation. The behavioral indexing technique is based on pre-computation of method results and storing them into an index. Previous work on behavioral indexing includes [15, 12, 13, 14, 2, 7].

The effects of two indexes could be entangled, that is, the inclusion of one index could affect the benefit achievable by another index. This phenomenon is known as *index interaction* [9]. Index interaction has a larger performance impact when the global effect of indexing multiple query paths is considered. Some work that deals with this effect includes [5, 13].

Following the study of access support relation [13, 14] and join index hierarchies [24], a new indexing structure, called the **triple-node hierarchy** is proposed in this paper. The triple-node hierarchy supports object references by maintaining direct association between two objects along a path expression of arbitrary path length while the intermediate objects are

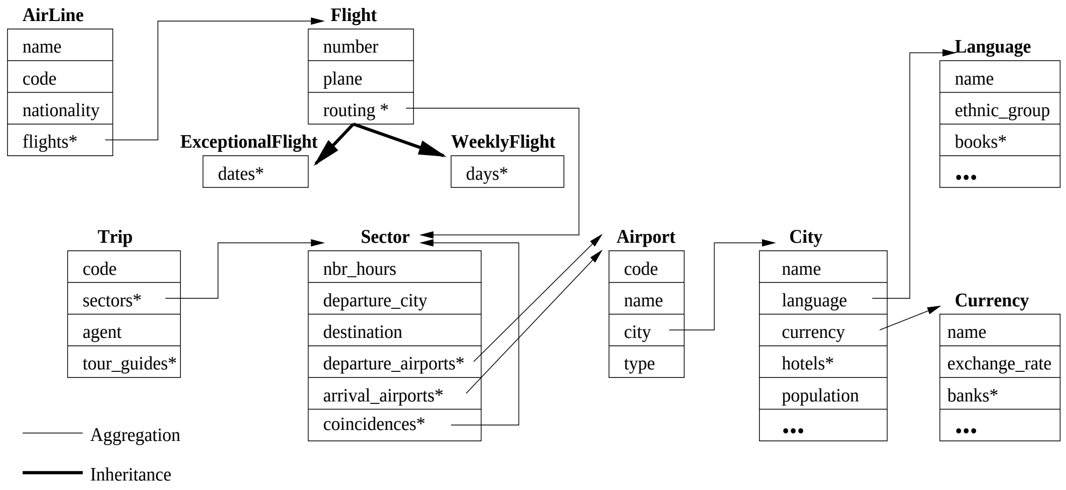


Figure 1: Object-oriented database schema example.

stored separately. It had been shown in [24] that only frequently referenced object pairs should be maintained. The triple-node hierarchies can also handle index interactions. We show that the proposed method out-perform the best previous method in terms of update and storage costs while maintaining similar navigation costs. The triple-node hierarchy supports multiple object pairs along multiple query paths of an aggregation hierarchy, minimizing the costs on overlapping path expressions. We also extended the triple-node hierarchy to handle integration of aggregation and inheritance hierarchies in OODB, and the performance is shown to better than the best previous [4] method known to us.

This paper is organized as follows. Section 2 is an overview of the join index structures. Section 3 presents the proposed triple-node hierarchy. Section 4 gives a performance evaluation and comparison with previous methods. Section 5 presents the problem of integrating the aggregation and inheritance hierarchies and how the triple-node hierarchy can be extended for this problem. Section 6 is a performance analysis and comparison with previous method. Section 7 is a conclusion.

2 Join index structures

Join index structure was first proposed by Valduriez [23] for optimizing join and semijoin operations in relational databases. The method of join index structure is extended by [24] for object-oriented database. We shall extend the method further and propose a new structure called the *triple-node hierarchy*, which we show to have better overall performance for navigation and updates. In this section, we give an overview of join index hierarchy.

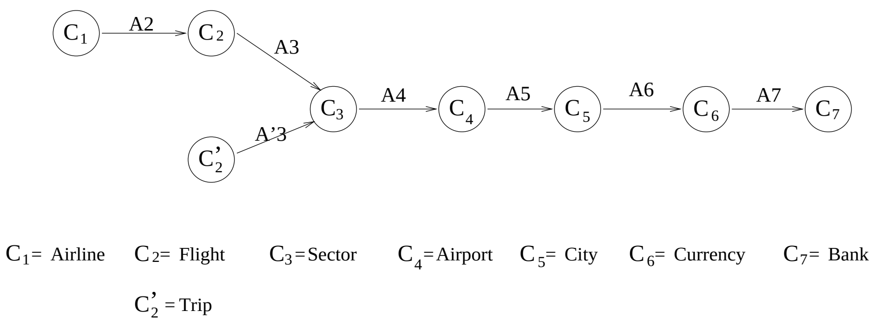


Figure 2: Overlapping Schema Paths for the example

A **database schema** or a **schema graph** is a directed graph in which the nodes correspond to classes, and edges to relationships between classes. An example of an object-oriented database schema is shown in Figure 1. In Figure 1, a class is represented by a box, and the attributes of the class are inside the box. Those attributes labeled with * denote multi-valued attributes. Suppose one is interested to find all the currencies that are used in cities that will be visited during a trip. This may help a traveler in planning for a trip. Then one may navigate the path ⟨ (**Trip**), sectors (**Sector**), arrival_airports (**Airport**), city (**City**), currency (**Currency**) ⟩.

In general, suppose A_k is an attribute of class C_i, and A_k ranges over class C_j, then there exists a directed edge from C_i to C_j in the schema graph, we say that the edge is **labeled** with A_k. In a schema graph, if for $i = 1, ..., n-1$, there is a directed edge from C_i to C_{i+1}, labeled with A_{i+1}, then $P(1,n) = \langle (C_1), A_2(C_2), A_3(C_3), ..., A_n(C_n) \rangle$ or $\langle (C_1), A_2(C_2), A_3(C_3), ..., A_n \rangle$ [2] is a **schema path** or simply a **path**. In the more general case, class C_i represents the inheritance sub-hierarchy whose root classes is C_i, which means that user is interested in classes within this sub-hierarchy.

Suppose we have two paths,
⟨ (**Trip**), sectors (**Sector**), arrival_airports (**Airport**), city (**City**), currency (**Currency**), banks (**Bank**) ⟩, and
⟨ (**Airline**), flights (**Flight**), routing (**Sector**), departure_airports (**Airport**), city (**City**)⟩.

Figure 2 represents the combined diagram for the two paths with an overlapping sub-path. These paths can be used to answer queries of the form:

Find the banks that provide money exchange for the currencies that is related to trip Z930.

Find all airlines that have flights departing from the city of Venice.

[2] C_n may not exist if the last attribute is **primitive** meaning that it has values which are not the object ID's of a class. In the rest of the paper we consider only paths that include C_n, since the cases without C_n will be similar.

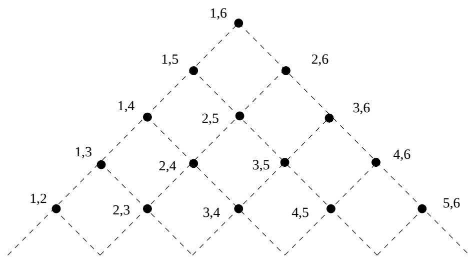


Figure 3: Complete join index hierarchy for a path length of 5

Note that the second query requires a backward navigation of the corresponding path.

2.1 Join Index Hierarchies

Our proposed structure is based on the join index hierarchy [24]. In this section, we shall give an overview of join index hierarchies, and point out the main observation that leads to our proposal. Suppose we are given a schema path $\langle (C_1), A_2(C_2), A_3(C_3), ..., A_n(C_n)\rangle$. We first consider the simpler case where each C_i is a single class [3]. A **join index file** or **join index node** $JI(i,j)(1 \leq i < j \leq n)$ consists of a set of tuples $(OID(O_i), OID(O_j), m)$, where O_i and O_j are objects of classes C_i and C_j respectively, and there exists an object path $< O_i, O_{i+1}, ..., O_{j-1}, O_j >$ such that for $k = 0, 1, ..., j-i-1$, O_{i+k+1} is referenced by O_{i+k} via the attribute A_{i+k+1}, and m is the number of the above distinct object paths that connect the objects O_i and O_j.

We assume that a set of join index nodes must be supported due to frequent references. Such nodes are called the **target join indices**. Join index nodes connecting different object classes along a schema path form a **join index hierarchy**, denoted as $JIH(C_1, A_2, C_2, A_3, ...A_n, C_n)$, or simply $JIH(1,n)$. The join index node for the longest join index path, $JI(1,n)$, is the root of the hierarchy. Each node $JI(i,j)$ where $j-i > 1$ has two direct children $JI(i, j-k)$ and $JI(i+l, j)$ where $0 < k < j-i$ and $0 < l < j-i$. The join index nodes $JI(i, i+1)$, for $i = 1, ...n-1$, are at the bottom of the hierarchy, and are called **base join index nodes**. The remaining join index nodes are called **derived join index nodes**. The non-target derived join index nodes are called the **auxiliary join index nodes**.

A **complete join index hierarchy** consists of all the possible base and the derived join indices. Figure 3 shows a complete join index hierarchy for a schema path of length 5. A **base join index hierarchy** consists of only the base join index nodes. A **partial join index hierarchy** consists of a subset of the derived join indices in a complete join index hierarchy, which include all the target join indices. For each vertex $JI(i,j)$ with $i+1 < j$ in a partial join

[3] The integration of aggregation and inheritance hierarchies is considered in Section 5.

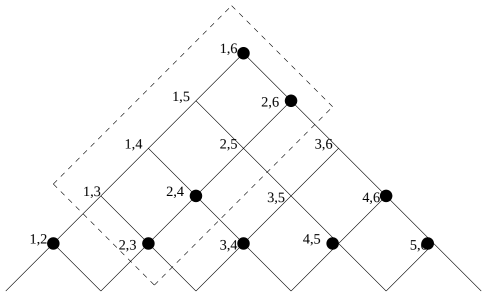


Figure 4: Update on a partial join index hierarchy

index hierarchy, there must also exists a node pair for $JI(i,k)$ and $JI(k,j)$, where $i < k < j$. For example, Figure 4 shows a partial join index hierarchy for a path of length 5, with target nodes $JI(1,6)$ and $JI(2,6)$. It means that queries are likely to navigate the forward or backward paths between classes $\{1,6\}$ or $\{2,6\}$. In a partial join index hierarchy, the non-target join index nodes are mainly used for update propagation. For example, $JI(2,4)$ in Figure 4 is such an index node. For each join index node $JI(i,j)$ there will be a B^+-tree built on the index i and another B^+-tree built on the index j, which are used in both navigation and updates operations.

It is shown in [24] that among the three types of index hierarchies, the partial join index hierarchy is the most efficient in supporting navigation for a pre-defined set of access, and requires the smallest amount of storage and update costs. In [24], an algorithm is given which computes an *optimal* join index hierarchy. The optimality is defined in [24] to depend on both the minimal number of join index nodes, and the minimal update cost.

2.2 Update maintenance

An update in one class or in the relationship of one class with another may cause the update of a base join index. Such an update may affect some corresponding join indices at higher levels. For an update on $JI(k,k+1)$ exactly the $JI(i,j)$ with $i \leq k$, $j \geq k$ will need to be updated accordingly. That is, exactly the join indices in the rectangle with corners at $JI(k,k+1)$ and $JI(1,n)$ need to be updated.

For example in the partial join index hierarchy highlighted by the black dots in Figure 4, if the node $JI(2,3)$ is modified, then the indices within the region of the dotted rectangle will be modified. They are nodes $JI(2,4), JI(2,6)$, and $JI(1,6)$. Node $JI(2,4)$ is first modified by $JI(2,3) \bowtie JI(3,4)$, then $JI(2,6)$ is modified by $JI(2,4) \bowtie JI(4,6)$, and finally $JI(1,6)$ is modified by $JI(1,2) \bowtie JI(2,6)$. Therefore the total number of $\bowtie$ operations performed is given by 3.

Important Observation: Note that in the above example, the B^{+}-tree T_1 indexed on class 3 for $JI(2,3)$ and the B^{+}-tree T_2 indexed on class 3 for $JI(3,4)$ will be traversed in the first $\bowtie$ in the above update. [4] One observe that it may be possible to combine T_1 and T_2 since they are built on the same index. The result is a B^{+}-tree that supports both $JI(2,3)$ and $JI(3,4)$, we may say that it supports a combined index node labeled by $JI(2,3,4)$. It implies that the first two B^{+}-tree search can be replaced by a single search, and that the join operation $JI(2,3) \bowtie JI(3,4)$ is no longer necessary. This can result in significant savings in the update cost. In the next section, we shall show that this idea turns out to be feasible.

3 Triple-Node Hierarchy

Given a path expression $P = \langle (C_1), A_2(C_2), A_3(C_3), ..., A_n(C_n) \rangle$, a **triple-node**, denoted by $Tri(i,j,k)$ $(1 \le i < j < k \le n)$ consists of a set of tuples $< OID(O_j), OID(O_t), TYPE, m >$ where O_j is an object which belongs to class C_j and O_t is an object which belongs to either class C_i or class C_k, distinguished by the flag $TYPE$, and there exist $m \ge 1$ distinct object paths connecting the two objects O_j and O_t. A B^{+}-tree is constructed for the triple-node, indexed on the OIDs of objects of class C_j.

If there exist tuples $< OID(O_j), OID(O_i), TYPE = i, m_0 >$ and $< OID(O_j), OID(O_k), TYPE = k, m_1 >$ in $Tri(i,j,k)$, then there exist $m_0 \times m_1$ distinct object paths connecting O_i and O_k via O_j. If there exists a tuple $< OID(O_j), OID(O_i), TYPE = i, m_0 >$, but there does not exist any tuple $< OID(O_j), OID(O_k), TYPE = k, m_1 >$ in $Tri(i,j,k)$, then there exist object paths connecting O_i and O_j but the paths do not emanate to any object in C_k.

The triple-node $Tri(\perp, j, k)$ contains only tuples $< OID(O_j), OID(O_k), m >$, where O_k belongs to class C_k, and the triple-node $Tri(j,k,\top)$ contains only tuples $< OID(O_k), OID(O_j), m >$, where O_j belongs to class C_j. In other words, $\perp$ and $\top$ denote non-exist classes.

The domain class of an attribute is **primitive** if the attribute consists of values other than object ID's. It is noted that all attributes in the path except A_n have as domain non-primitive classes. The domain of the last attribute A_n may be either primitive or non-primitive. If it is primitive, the B^{+}-tree that is associated with the triple-node $Tri(i,n,\top)$, $i < n$, will be indexed on the attribute values. In this case, queries may be targeted on instances with a particular value or values that fall in a certain range for attribute A_n.

The **triple-node hierarchy** is a hierarchy connecting triple-nodes together. The triple-

[4] Similarly the B^{+}-tree T_3 indexed on class 4 for $JI(2,4)$ and the B^{+}-tree T_4 indexed on class 4 for $JI(4,6)$ are traversed in the second $\bowtie$ operation. the B^{+}-tree T_5 indexed on class 2 for $JI(1,2)$ and the B^{+}-tree T_6 indexed on class 2 for $JI(2,6)$ are traversed in the third $\bowtie$ operation. Therefore, T_3 and T_4 can be combined, and likewise for T_5 and T_6.

node $Tri(\perp, j, k)$, $j < k$, is used to support forward object references from a set of objects of class C_j to objects of class C_k, while the triple-node $Tri(j, k, \top)$, $j < k$, is used to support backward object references from a set of objects of class C_k to objects of class C_j. Suppose the triple-node $Tri(\perp, i, k)$ is constructed to support the forward object references between objects of classes C_i and C_k on the path expression $P(i, k) = \langle (C_i), A_{i+1}(C_{i+1}), ..., A_k(C_k) \rangle$ (or $Tri(i, k, \top)$ for backward object references), in order to support update operations of the triple-node $Tri(\perp, i, k)$ (or $Tri(i, k, \top)$), a triple-node $Tri(i, j, k)$, $i < j < k$, is constructed. Whenever there is an update on $Tri(i, j, k)$, $Tri(\perp, i, k)$ (or $Tri(i, k, \top)$) is also updated. If $i+1 < j$ and $j+1 < k$, then two triple-nodes $Tri(i, p, j)$ and $Tri(j, q, k)$, where $i < p < j$, and $j < q < k$ are in turn constructed to support updates on $Tri(i, j, k)$. This kind of relationship related to updates connects a set of triple-nodes together to form a triple-node hierarchy.

3.1 Construction of the Triple-Node Hierarchy

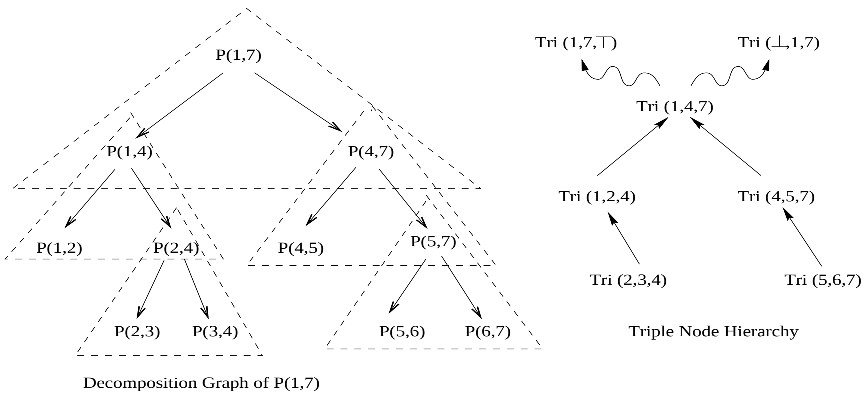


Figure 5: Partial join index hierarchy for $P(1, 7)$ and triple-node hierarchy

A partial join index hierarchy describes the sequence of decomposing a long path expression into shorter ones. For example, Figure 5 shows a *decomposition graph* that corresponds to the partial join index hierarchy of the path expression $P(1, 7)$, where $P(1, 7)$ is decomposed at 4 into $P(1, 4)$ and $P(4, 7)$, $P(1, 4)$ is decomposed at 2 into $P(1, 2)$ and $P(2, 4)$, etc. The label $P(i, j)$ in the decomposition graph corresponds to $JI(i, j)$ in the partial join index hierarchy. Given a partial join index hierarchy, a set of triple-nodes can be identified by the decomposition points in the graph. Suppose the path expression $P(i, k)$ is decomposed at j into two subpath expressions, a triple-node $Tri(i, j, k)$ is constructed.

For example, in Figure 5, $P(1, 7)$ is decomposed at 4 into $P(1, 4)$ and $P(4, 7)$. Thus, the triple-node $Tri(1, 4, 7)$ is included in the triple-node hierarchy. $P(1, 4)$ is decomposed at 2 into $P(1, 2)$ and $P(2, 4)$ and thus $Tri(1, 2, 4)$ is also included. Similarly, the triple-nodes $Tri(2, 3, 4)$, $Tri(4, 5, 7)$ and $Tri(5, 6, 7)$ are also included.

After identifying the set of triple-nodes in the triple-node hierarchy, the **level** of a triple-nodes $Tri(i, j, k)$ is defined as the maximum value of $j - i$ and $k - j$. For example, $Tri(2, 3, 4)$ is at level 1, $Tri(1, 2, 4)$ is level 2 and $Tri(1, 4, 7)$ is level 3.

If the system does not store information between an object and the objects referencing it, then we need to store extra triple-nodes $Tri(i, i+1, \top)$ $(1 \leq i \leq n-1)$, for every class C_{i+1} in the path expression $P = \langle (C_1), A_2(C_2), A_3(C_3), ..., A_n(C_n) \rangle$ except if there is already a triple-node $Tri(i, i+1, Y)$ $(Y > i+1)$ being stored in the triple-node hierarchy.

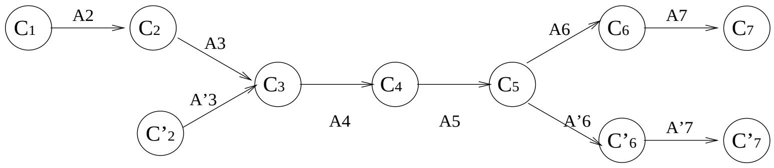


Figure 6: Overlapping Path Expressions

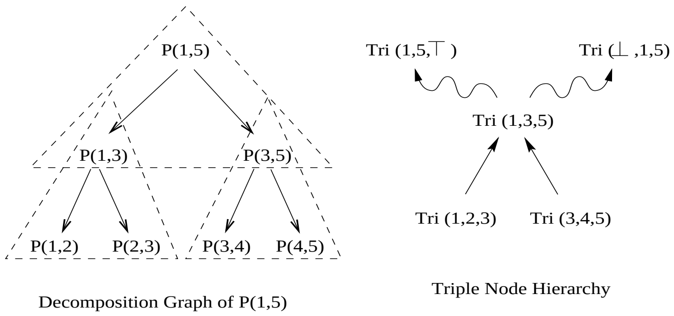


Figure 7: Triple-Node Hierarchy supporting $Tri(1, 5, \top)$ and $Tri(\perp, 1, 5)$

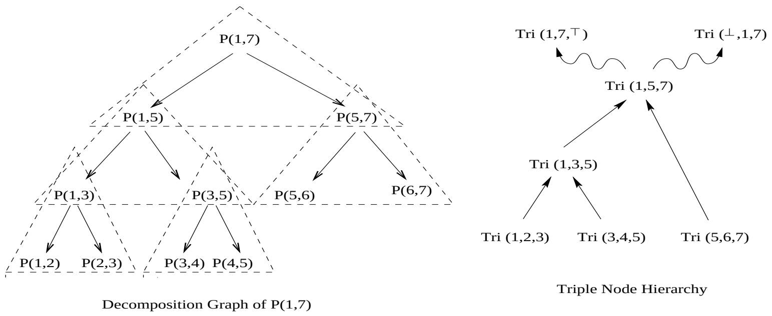


Figure 8: Triple-Node Hierarchy supporting $Tri(1, 7, \top)$ and $Tri(\perp, 1, 7)$

The triple-node hierarchy can be used to support a set of frequently referenced object pairs. These object pairs lie on a set of path expressions in the aggregation hierarchy which may be overlapped. Consider the case shown in Figure 6. Suppose we have to support three frequently referenced object pairs on $P(1, 5)$, $P(1, 7)$ and $P(2'7')$, the partial join index hierarchies of the

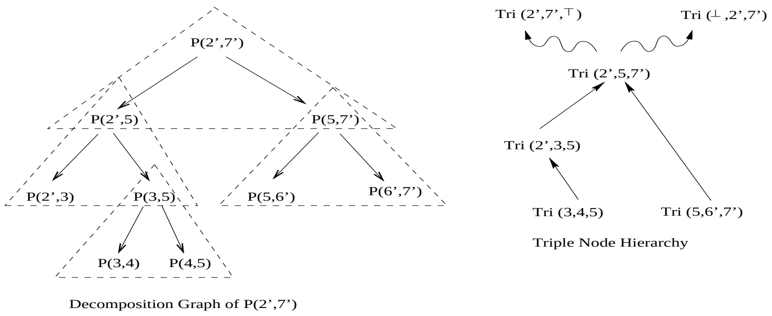


Figure 9: Triple-Node Hierarchy supporting $Tri(2', 7', \top)$ and $Tri(\perp, 2', 7')$

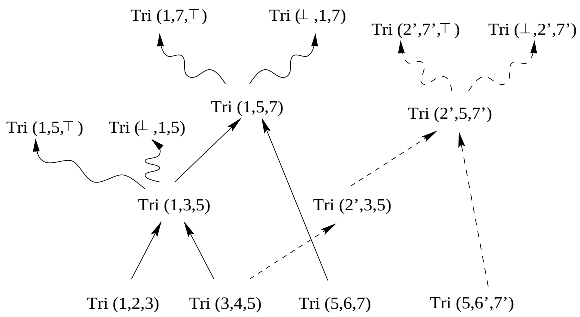


Figure 10: Combined Triple-Node Hierarchy

three path expressions and the corresponding triple-node hierarchies are shown in Figure 7, Figure 8 and Figure 9. Figure 10 shows the combined triple-node hierarchy.

3.2 Updates in the Triple-Node Hierarchy

Updates in object-oriented databases will be reflected in the index structure. An update in the relationship between two classes will affect the corresponding triple-nodes in the triple-node hierarchy. Here we consider three types of update operations in the attribute relationship A_{i+1} between two classes C_i and C_{i+1} :

Insert operation : insert O_{i+1} into $O_i.A_{i+1}$. This operation inserts an object O_{i+1} of class C_{i+1} into the attribute A_{i+1} of an object O_i of class C_i.

Delete operation : delete O_{i+1} from $O_i.A_{i+1}$. This operation deletes an object O_{i+1} of class C_{i+1} from the attribute A_{i+1} of an object O_i of class C_i.

Modify operation : this operation modifies the attribute value of an object O_i of class C_i from an object O_{i+1} of class C_{i+1} to another object O'_{i+1} of class C_{i+1}. This operation is performed by deleting O_{i+1} from $O_i.A_{i+1}$ and inserting O'_{i+1} into $O_i.A_{i+1}$

Update($\delta(i,j)$, $Tri(x,i,j)$)
Input: a set of update operations $\delta(i,j)$; a triple-node $Tri(x,i,j)$
Output: a set of update operations $\delta(x,j)$

```
for all tuples (OID(O_i), OID(O_j), m) in δ(i,j)
  if there exists tuple < OID(O_i), OID(O_j), TYPE = j, m_1 > in Tri(x,i,j)
    set Δm = m_1 + m
    if Δm > 0
      replace the tuple with < OID(O_i), OID(O_j), TYPE = j, Δm >
    else  remove the tuple
    endif
  else
    set Δm = m
    insert tuple < OID(O_i), OID(O_j), TYPE = j, Δm >
  endif
  for all tuples < OID(O_i), OID(O_x), TYPE = x, m_0 > in Tri(x,i,j)
    insert (OID(O_x), OID(O_j), m_0 × Δm) into δ(x,j)
  endfor
endfor
```

Figure 11: Algorithm Update

Let $\delta(i,j)$ be a set of update operations on classes C_i and C_j. $\delta(i,j)$ contains a set of tuples $(OID(O_i), OID(O_j), m)$, where m denotes the total number of object paths connecting O_i and O_j which are being updated, and a positive m implies that the operation is an insert operation while a negative m implies delete operation. For example, the insert operation above corresponds to $(OID(O_i), OID(O_{i+1}), 1)$ and the delete operation above corresponds to $(OID(O_i), OID(O_{i+1}), -1)$.

Figure 11 shows an algorithm which performs a set of update operations $\delta(i,j)$ in a triple-node $Tri(x,i,j)$ where $x < i < j$. The update operations in triple-node $Tri(i,j,Y)$ where $i < j < Y$ can be done similarly. Figure 12 shows an algorithm which performs an update in the triple-node hierarchy. Update operations are always initiated by a set of update operations δ(i,i+1), and propagated to higher level triple-nodes.

4 Performance Evaluation

We have constructed a mathematical model for the performance evaluation of the triple-node hierarchy. which is shown in Appendix A. Based on this model, we compare the performance for a few indexing techniques including the access support relations [13], the join index hierarchies

Update_Propagation($\delta(i,j)$)
Input: a set of update operations $\delta(i,j)$

```
for all triple-nodes in the triple-node hierarchy
  for all triple-nodes with the format Tri(x, i, j),
    Update(δ(i, j), Tri(x, i, j))
    if δ(x, j) ≠ φ then Update_Propagation(δ(x, j)) endif
  endfor
  for all triple-nodes with the format Tri(i, j, y),
    Update(δ(i, j),Tri(i, j, y))
    if δ(i, y) ≠ φ then Update_Propagation(δ(i, y)) endif
  endfor
endfor
```

Figure 12: Algorithm Update_Propagation

class	C_1	C_2	C_3	C_4	C_5	C_6	C_7
$\|C_i\|$	2000	1500	2000	1800	2000	1500	1800
fan_i	0.8s	0.9s	1.0s	1.1s	0.8s	1.2s	1.8s
class		C'_2				C'_6	C'_7
$\|C_i\|$		1500				1500	1800
fan_i		0.9s				1.2s	1.8s
s	selectivity factor (variable)						

Table 1: Application-specific Parameters

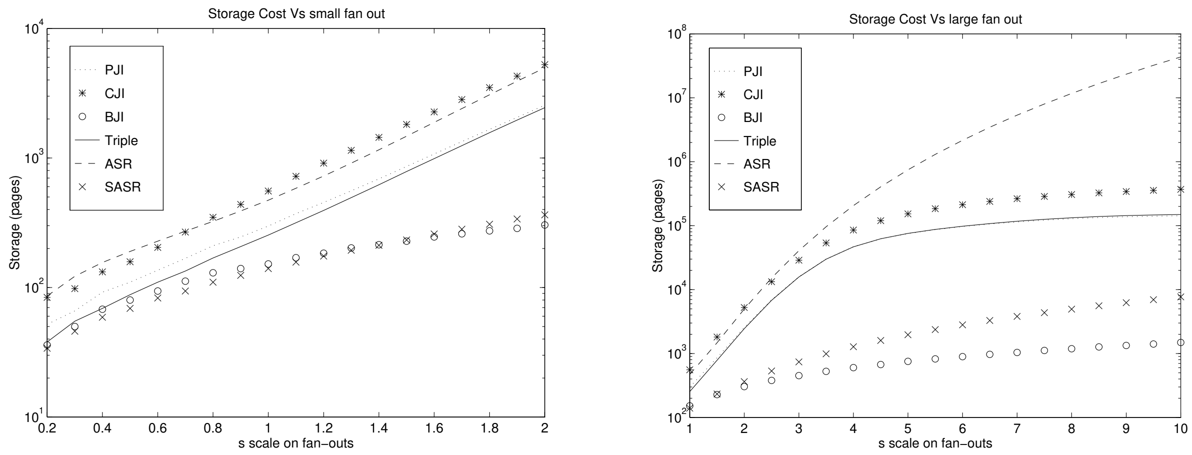
[24], and the triple-node hierarchy. The sample aggregation hierarchy is the one in Figure 6. The running example tries to support both forward and backward navigations between frequently referenced classes includes C_1 and C_7 ($Q^{1,7}$), C_1 and C_5 ($Q^{1,5}$), and $C_{2'}$ and $C_{7'}$ ($Q^{2',7'}$).

Two different types of access support relations are compared in our cost model : the full access support relations, and the decomposed access support relations. Three types of join index hierarchies are also compared : Complete, Partial and Based.[5]

Application-specific parameters are shown in Table 1. The selectivity factor determines the fan-out between classes in the schema path. Two full access support relations (labeled ASR) are constructed to support navigations. Five access support relations (labeled SASR) including $\{ASR^{1,3}$, $ASR^{3,5}$, $ASR^{5,7}$, $ASR^{2',3}$, $ASR^{5,7'}\}$ are constructed according to the rules on sharing of ASR [13]. The partial join index hierarchy structure maintains the set of target nodes $\{JI(1,5)$, $JI(1,7)$, $JI(2',7')\}$ for efficient navigation. The partial join index hierarchy

[5] see Section 2.1

consists of five auxiliary join index nodes { $JI(1,3)$, $JI(3,5)$, $JI(5,7)$, $JI(2',5)$, $JI(5,7')$ }. The triple-node hierarchy is constructed as shown in Figure 10.



(a) Storage Vs small fanout

(b) Storage Vs large fanout

Figure 13: Storage Vs fanout

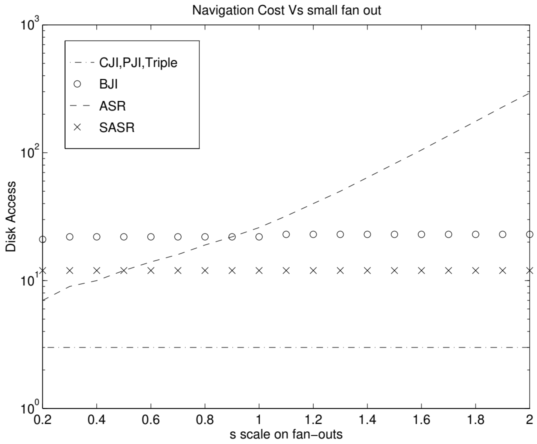
4.1 Storage Cost

The fan-out (selectivity) among classes is critical to the constructions of join index nodes, triple-nodes and access support relations. The storage costs of the six index structures against various fan-out factors are shown in Figure 13.

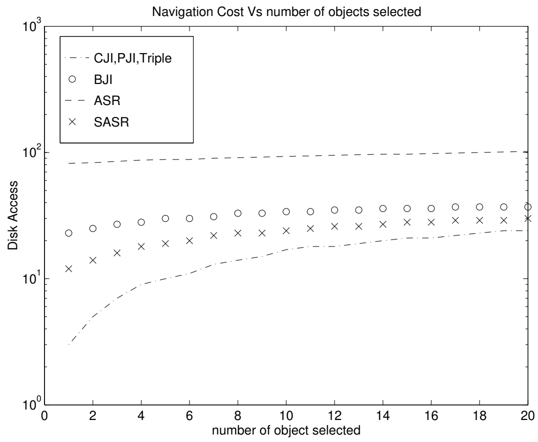
In general, the storage costs increase as the fan-out factor increases. When the fan-out factor is small, the triple-node hierarchy, SASR, and the based join index hierarchy incur smaller storage costs than the others. The costs of SASR and the based join index hierarchy increase more slowly than the others as the fan-out factor increases. This is due to their simple structures.

4.2 Navigation Cost

Figure 14(a) shows the navigation costs of the six index structures. Here, we consider only the frequently referenced navigations , i.e., $(Q^{1,7}, Q^{1,5}, Q^{2',7'})$ with the same weight. We also assume that the forward and backward navigations share the same weight. The navigation starts by only one object. As the complete join index hierarchy, the partial join index hierarchy and the triple-node hierarchy support navigations by maintaining direct mapping between frequently referenced object pairs, therefore their navigation costs are the same. In fact, the navigation cost of the complete join index is slightly lower than the other two because it provides full support



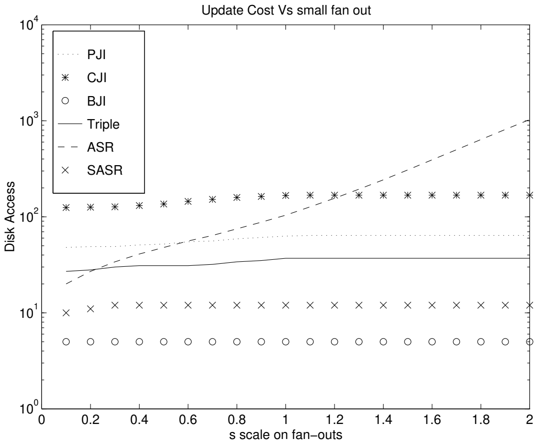
(a) Navigation Cost Vs small fanout



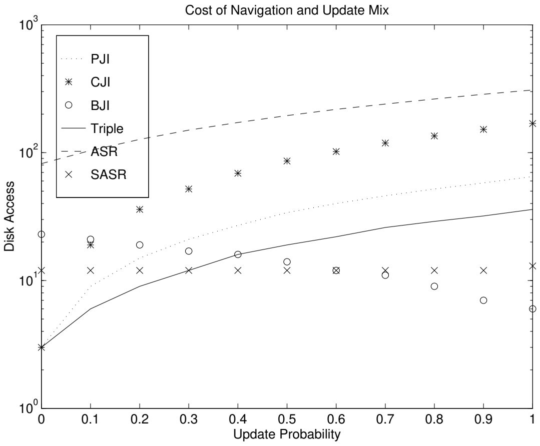
(b) Navigation Cost Vs objects selected

Figure 14: Navigation Cost Vs fanout

for non-frequent navigations, which are considered as rare cases. For Full ASR, the navigations on $Q^{1,7}$ and $Q^{2',7'}$ are effective, because the indexes are clustered. However, the backward navigation cost on $Q^{1,5}$ requires a higher cost because $Q^{1,5}$ is not indexed and a search through the entire ASR is required. For the base join index hierarchy and SASR, a higher navigation cost is required because extra join operations are required. Figure 14(b) shows the navigation costs when the navigation starts from more than one object, and the selectivity factor for fan-out is assumed to be 1.5. We can see that the triple-node hierarchy, the complete join index hierarchy and the partial join index hierarchy perform better in navigation operations.



(a) Update Cost Vs small fanout



(b) Navigation Update Mix (fanout set to 1.5)

Figure 15: Update Cost and Navigation Update Mix

4.3 Update Cost

Figure 15(a) shows the update cost of the six index structures. We assume that the update probability of each class relationship for adjacent classes on the schema paths is the same. We also assume that the update operation starts with only one relation. It is obvious that the update costs of SASR and the based join index hierarchy are lower because of their simple structures. The triple-node hierarchy has a lower update cost than the complete and partial join index hierarchies as well as Full ASR. These results match our expectation as each time when update is propagated to higher level nodes in the partial join index hierarchy, extra join operations are required. These costs, however, are saved in the triple-node hierarchy.

Figure 15(b) describes the cost of mixtures of navigation and update operations for the six index structures. The total cost is defined as $(1-p)\times$ Navigation Cost $+\ p\times$ Update Cost, where p is the update probability. The update probability =0.6 means that there are 60% updates and 40% navigations. The fan-out factor is set to be 1.5. It can be seen that the triple-node hierarchy outperforms the other methods. It requires only about half the number of page accesses that is needed for the partial join index hierarchies in most cases.

5 Aggregation and Inheritance Hierarchies

The triple-node hierarchy presented in the previous sections is designed for the support of aggregation hierarchy, i.e., navigations through a sequence of object classes via their attribute relationships. In this section, we show how we can provide efficient support for queries against the integration of the inheritance hierarchies and the aggregation hierarchies.

In a schema path which involves inheritance hierarchies, if a set of subclasses associated with the same higher-level class has similar kinds of attributes, it could be beneficial to construct one combined triple-node for all classes in the inheritance sub-hierarchy instead of a large number of small triple-nodes for each class in the inheritance sub-hierarchy. If this idea is adopted, then the tuples in the triple-node $Tri(i,j,k)$ may include OIDs of the instances from classes C_i, C_j and C_k, and all their subclasses which inherit the nested attribute from the superclasses. For the triple-nodes $Tri(i,j,\top)$ and $Tri(\perp,i,j)$ which are used to support navigations, we may either construct a combined triple-node for all classes or a large number of small triple-nodes for each class in the inheritance sub-hierarchy.

For queries targeted on all classes in an inheritance sub-hierarchy rooted at class C_i, the construction of a B^+-tree on the combined triple-nodes $Tri(i,n,\top)$ can provide efficient support. However, as the tuples in these nodes contain OIDs of instances of all classes in the inheritance sub-hierarchy, this simple B^+-tree may not be able to provide efficient support for queries that target on only one class. On the other hand, if we construct several small triple-nodes $Tri(i_1, n, \top)$, $Tri(i_2, n, \top)$,... for each class in the inheritance sub-hierarchy instead of a combined one, queries targeted on all classes in the inheritance sub-hierarchy may require navigation on several small triple-nodes, which may be costly.

We note that there have been several indexing structures proposed in the literature which are able to meet this conflicting requirement and to provide efficient support for queries against an inheritance sub-hierarchy. Examples are CH-trees, H-trees, hcC-trees, CG-trees and CD-trees. It is therefore of interest to integrate these indexing techniques with the triple-node hierarchy as a solution for indexing in an integration of both inheritance sub-hierarchies and aggregation hierarchies.

We have investigated the integration of the triple-node hierarchy with the CH-tree. We choose CH-tree because it has a simple structure. Other methods employ more complex data structures, but has been shown by experiments to have better performance than the CH-tree. We believe that the integration of the triple-node hierarchy with one of these other methods can achieve even better performance.

5.1 Class-Hierarchy Tree

The class-hierarchy tree (CH-tree) was proposed by Kim et al. [17]. It has a simple structure based on the B^+-tree. As subclasses inherit attributes from their superclass, it is possible to maintain an index on a common attribute for all classes in an inheritance sub-hierarchy. The domain of an attribute may be either a primitive class or a non-primitive class. Thus, the key values in the index can be either some primitive values or the OIDs of the instances of the domain class.

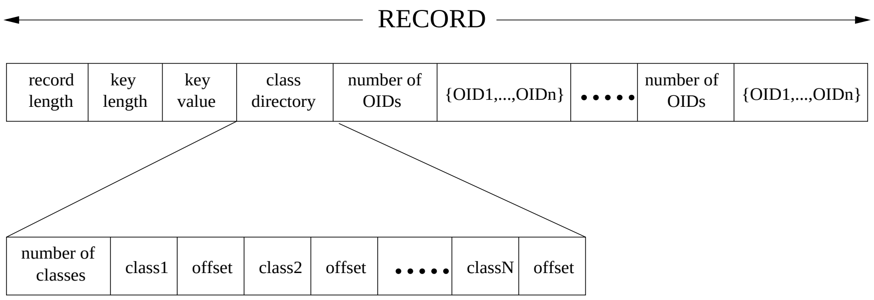
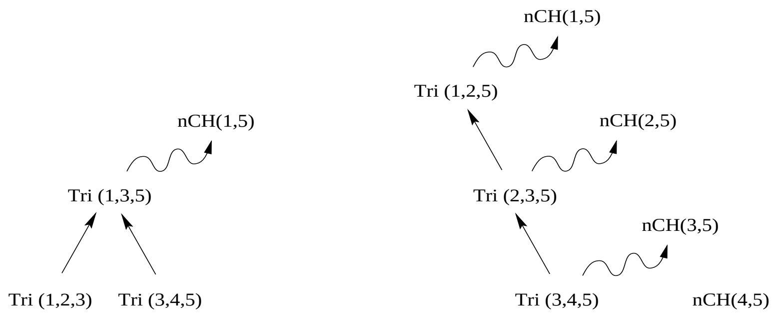


Figure 16: A leaf node record in CH-tree

The structure of a non-leaf node of a CH-tree is similar to that of a B^+-tree while the structure of leaf nodes of a CH-tree has a different structure. A leaf node of a CH-tree consists of (1) a key value and (2) a class directory. For each target class in the inheritance sub-hierarchy, a CH-tree leaf node also consists of (3) a list of OIDs of instances of the class that hold the key value in the indexed attribute and (4) the number of elements in the OIDs list. The class directory contains an entry for each class that has instances with the key value in the indexed attribute. An entry for a class consists of the class identifier and the offset in the index record at which the OIDs of the objects can be found. The leaf node of a CH-tree groups the OIDs for a key value in terms of the classes to which they belong. Figure 16 shows an example of a leaf node in a CH-tree.

5.2 The Nested CH-tree

A **Nested CH-tree (nCH-tree)** is a CH-tree which is indexed on a nested attribute. The intermediate objects are supported by a triple-node hierarchy. The nCH-tree supports a fast evaluation of queries issued against a class or all classes in the inheritance sub-hierarchy by a nested attribute, in the scope of a path expression in an aggregation hierarchy ending with that indexed attribute. Update operations do not require object traversals. Instead, they are propagated from the triple-node hierarchy. Note that a CH-tree is a special case of a nested CH-tree. In a CH-tree the indexed attribute is a non-nested attribute of the root class of an inheritance sub-hierarchy and thus the tree does not require the support of the triple-node hierarchy.



(a) nCH(1,5) supported by Triple Node Hierarchy

(b) Nested CH-trees supported by Triple Node Hierarchy

Figure 17: nCH-tree organizations

Let $C_{i,j}$ denote the j-th class in the inheritance sub-hierarchy having position i in path P. We make the assumption that classes in an inheritance sub-hierarchy are sorted in some order. We require that the root of the class hierarchy is the first class in the ordering. Thus $C_{i,1}$ denotes the root class of the inheritance sub-hierarchy having position i in the path expression. For simplicity, we shall sometimes refer to class $C_{i,1}$ as class C_i. Let $nCH(i,j)$ denote a nested CH-tree on an inheritance sub-hierarchy rooted at class $C_{i,1}$ and indexed on the nested attribute A_j. Given a path expression $P = \langle (C_{1,1}, A_2(C_{2,1}), A_3(C_{3,1}), ..., A_n(C_{n,1}) \rangle$, to construct the nCH-tree $nCH(1,n)$ is similar to constructing a triple-node hierarchy supporting the triple-node $Tri(1,n,\top)$. Figure 17(a) shows the index organization of an nCH-tree $nCH(1,5)$ supported by the corresponding triple-node hierarchy. Figure 17(b) shows the index organization of four nCH-trees, including $nCH(1,5), nCH(2,5), nCH(3,5)$ and $nCH(4,5)$, supported by the corresponding triple-node hierarchy.

A retrieval operation performed on an nCH-tree is similar to that on a CH-tree. In the nCH-tree $nCH(i,j)$, a predicate against the indexed attribute A_j on a single class is evaluated as follows. Let C be the class against which the predicate is issued. The index is scanned to find the leaf node record with the key value satisfying the predicate. Then the key directory is accessed to determine the offset in the index record where the list of OIDs of the instances of C is located. If there is no entry for class C, then there are no instances of C satisfying the predicate. If the query is against more than one class in the indexed inheritance sub-hierarchy rooted at $C_{i,1}$, the predicate is processed in the same way, except that the lookup in the key directory is executed for each class involved in the query. Therefore, this access mechanism can be used when a query is applied only to some (not necessarily all) classes in the target inheritance sub-hierarchy.

Update operation is propagated by the triple-node hierarchy to the nCH-tree. For an nCH-tree $nCH(i,j)$, an update in the attribute relationship within the scope of the nCH-tree may be reflected by an update operation $\delta(i,j)$ propagated by the triple-node hierarchy.

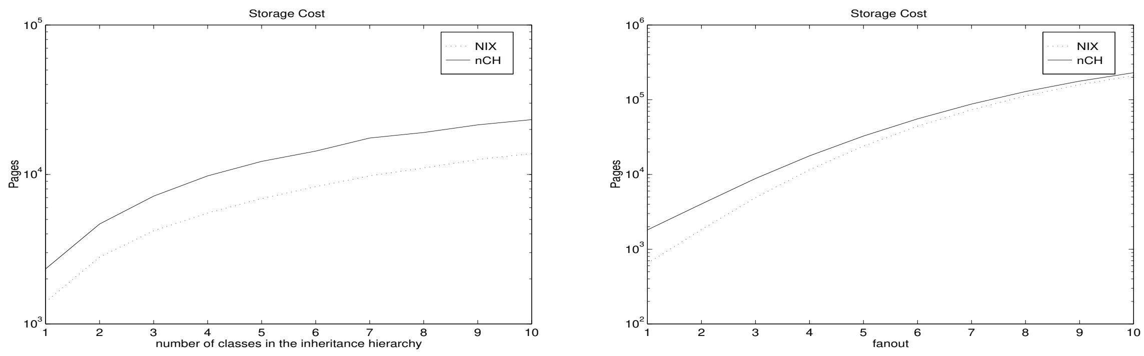
5.3 Expectations on Performance

We shall compared our method with the **nested-inherited index (NIX)** [4], which is the best previous method known to us that can provide an integrated treatment of indexing in the framework of both aggregation and inheritance hierarchies in OODBs. [6]

NIX provides an efficient evaluation of nested predicates for both queries having as a target a single class and queries having as a target any number of classes in a given inheritance graph.

The NIX organization stores the OIDs of all classes within the scope of a path expression in the leaf node records. The size of a leaf node record in the NIX can be quite large. On the other hand, the nCH-tree stores only the OIDs of the target classes in each inheritance sub-hierarchy with the indexed key value, while the intermediate objects that are irrelevant to the navigation operations are stored separately in the triple-node hierarchy. This simple structure allows the nCH-tree to have a smaller leaf node size and thus more leaf nodes may be stored in a page. This is beneficial for range queries which involves the retrieval of records for contiguous key value. Moreover, there is in general a less number of tuples stored in each nCH-tree, which lowers the height of the B^+-tree compared to the NIX structure. This is important for point queries, since the retrieval cost for point queries depends mainly on the height of the B^+-tree

[6] An Enhanced Nested-Inherited Index (ENIX) has been proposed in [8], which supports more access patterns. However, as shown in [8], the retrieval performance of ENIX is the same as that of NIX. Therefore, for our purpose, comparison with NIX is sufficient.



(a) (b)

Figure 18: Storage Cost

structures associated with the index organization. Therefore, we believe that the nCH-tree can achieve a lower navigation cost.

6 Performance Evaluation

The mathematical cost model for the performance evaluation of the nCH-tree is given in Appendix B. We have conducted simulation experiments based on the mathematical cost model. We compare the performance of the nCH-tree structure with the nested-inherited index (NIX) [4]. The cost model for the NIX is given in [4, 3]. The system parameters in Table 3 are set according to common implementations of the B^{+}-tree index. We vary the parameters in Table 4 for all classes in the path expression.

To evaluate the storage requirements of the nCH-tree and the NIX organization, we have performed a set of experiments by varying the number of classes, and hence the total number of instances, and the fanout parameter.

We consider a path expression $P = \langle (C_1), A_2(C_2), A_3(C_3), A_4(C_4), A_5(C_5)\rangle$ with length $=$ 4. We assume that there are four inheritance hierarchies associated with each class in the path expression, and each of these hierarchies carries the same number of classes and there are 10000 instances in each class. The parameters d, D, and N are set to 10000 for all classes. The key value is the nested attribute A_5, which is an attribute of class $C_{4,1}$. As the NIX organization provides support for every class in the scope of the path expression, in order to have a fair comparison, we constructed four nCH-trees, one for each class in the path expression.

6.1 Storage Cost

Figure 18(a) shows the storage cost of the nCH-tree and NIX organization against the number of classes in each inheritance sub-hierarchy. The fanout parameter is set to 1 for all classes, except for the classes rooted at $C_{4,1}$, which is set to 20. As the number of classes in each inheritance sub-hierarchy increases, the total number of object instances also increases, and thus more space is needed. This factor is reflected in the storage cost graph.

Figure 18(b) shows the storage cost of the nCH-tree and NIX organization against the fanout parameter. The number of classes in each inheritance sub-hierarchy is set to 3, and the fanout is the same for all classes. The costs of both structures increase as the fanout parameter increases.

Although the results show that the nCH-tree has a higher storage cost than the NIX organization, this may not be crucial because large capacity storage devices are widely available in the market. Therefore, it may be preferable to privilege organizations which provide good performance, even if they have a greater demand on storage requirements.

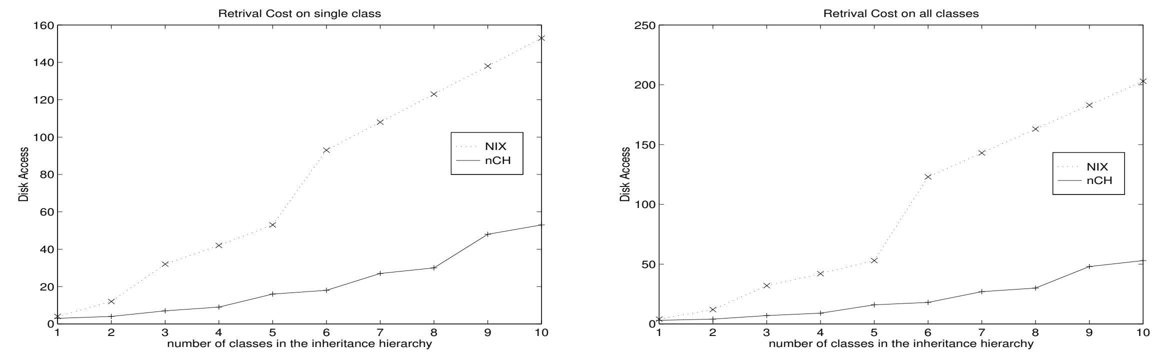
Note that we have constructed four nCH-trees, one for each class in the path expression. However, as the nCH-trees can support only the frequently referenced navigations, the storage cost of nCH-trees may be actually lower than the results reported here; this happens when some of these nCH-trees are considered as non-frequently navigations. Note also that if NIX has to support queries on relationships not indexing on class C_n, it may greatly increase the storage cost, whereas the nCH-tree structure can support such queries easily.

6.2 Range Query Cost

We have performed several simulation experiments by varying different factors that influence the performance of the indexing structures. The retrieval cost is calculated by averaging the cost on retrieval operations on each of the four classes.

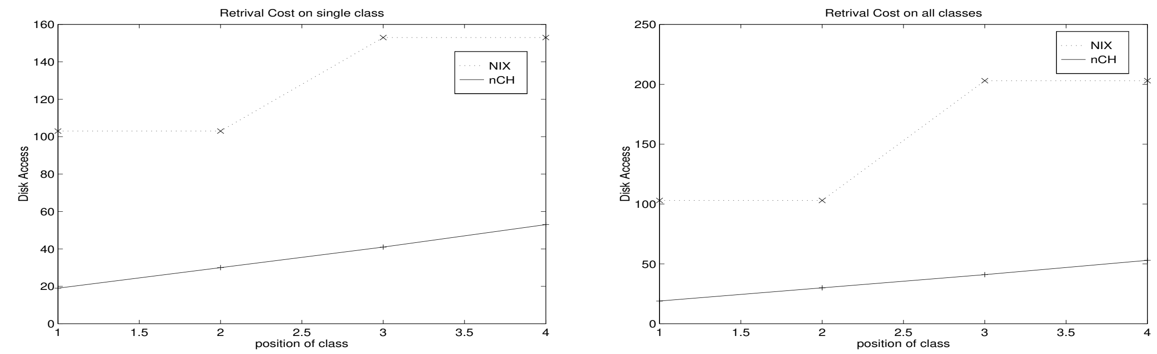
(1) In the first experiment, we measured the retrieval cost against the number of classes in the inheritance hierarchies rooted at the classes in a path. The value of the fanout parameter fan is set to 1 for all classes except those rooted at $C_{4,1}$, which is set to 20. The range queries cover 0.1% of the available key values. Figure 19 shows the average retrieval cost of the nCH-tree and NIX organization against the number of classes in the inheritance hierarchies, for queries targeted on a single class and on all classes in the inheritance sub-hierarchy. The nCH-tree has a lower retrieval cost in both cases. The retrieval cost of the NIX increases faster than that of the nCH-tree. It is because of the rapid growth of the size of the leaf node record in the NIX.

(2) In the second experiment, we measured the retrieval cost against the position of the



(a) (b)

Figure 19: Retrieval Cost Vs number of classes in the inheritance hierarchies



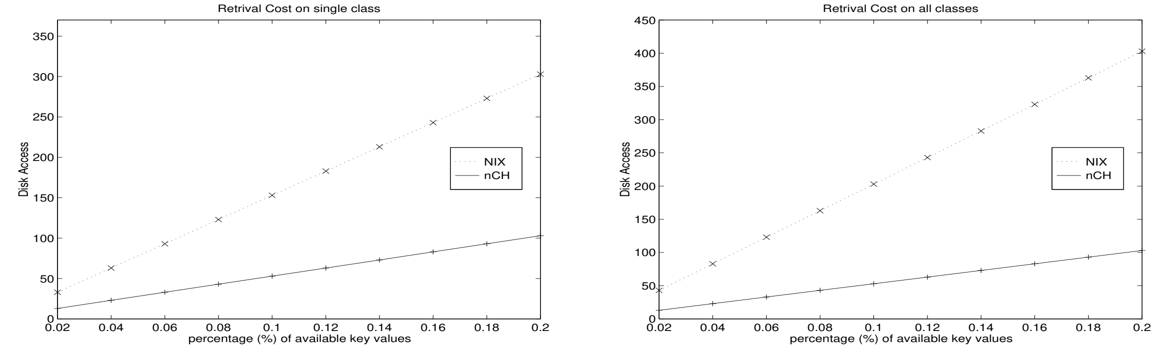
(a) (b)

Figure 20: Retrieval Cost Vs position of class having large fanout

classes with the large fanout parameter fan. The value of the fanout parameter fan is set to 1 for all classes except the classes rooted at the position being measure, which is set to 20. The number of classes in each inheritance hierarchies is set to 10. The range queries cover 0.1% of the available key values.

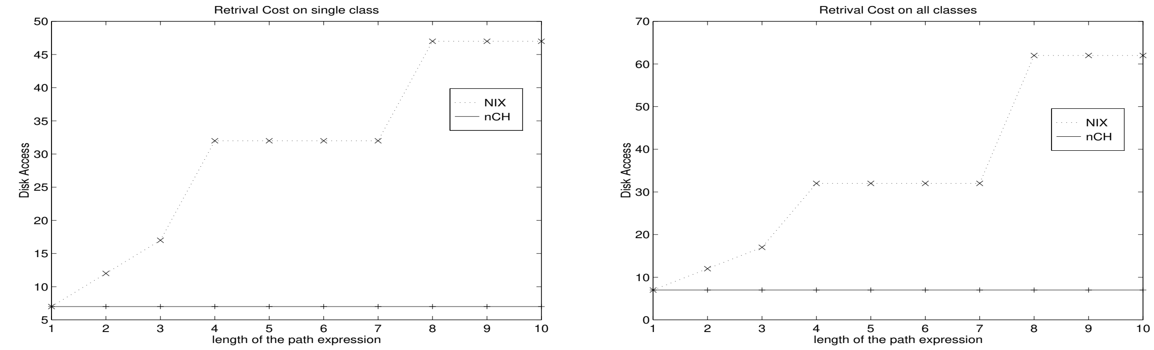
Figure 20 shows the average retrieval cost of the nCH-tree and NIX organization against the number of classes in the inheritance hierarchies, for queries targeted on a single class and on all classes in the inheritance sub-hierarchy. The nCH-tree has a lower retrieval cost in both cases. The retrieval cost of the nCH-tree and NIX increase as the position of classes having a large fanout shifts towards the indexed attribute. This is because for an object with a large fanout, the instances that reference this object will also have a large fanout. Therefore, as the position of classes having a large fanout shifts towards the indexed attribute, the number of objects having the same key value increases. The growth of the size of the leaf node record incurs more retrieval cost.

(3) In the third experiment, we measured the retrieval cost against the range of key values



(a) (b)

Figure 21: Retrieval Cost Vs range of key values covered by the queries



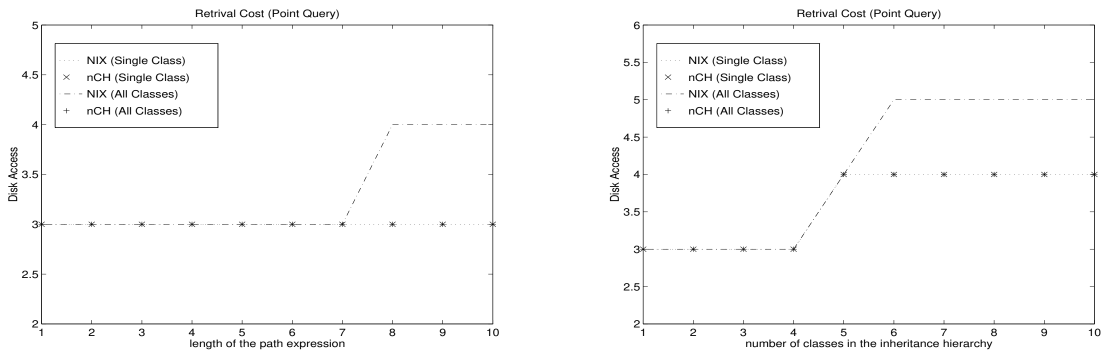
(a) (b)

Figure 22: Average Retrieval Cost Vs path length

covered by the queries. The values of the fanout parameter fan is set to 1 for all classes except the classes rooted at $C_{4,1}$, which is set to 20. The number of classes in each inheritance hierarchies is set to 10. The range of key values covered by the queries is varied to cover from 0.02% of the available key values to 0.2%.

Figure 21 shows the average retrieval cost of the nCH-tree and NIX organization against the range of queries targeted on a single class and on all classes in the inheritance sub-hierarchy. The nCH-tree has a lower retrieval cost in both cases. The retrieval cost of the NIX increases faster than the nCH-tree because the size of the leaf node records of the NIX is greater than that of the nCH-tree, which does not favor range search.

(4) In the fourth experiment, we measure the retrieval cost against the length of the path expression. The values of the fanout parameter fan is set to 1 for all classes except the classes rooted at the last class of the path expression, i.e. $C_{n,1}$, which is set to 20. The number of classes in each inheritance hierarchies is set to 3. The range of key values covered by the queries is set to 0.1% of the available key values.



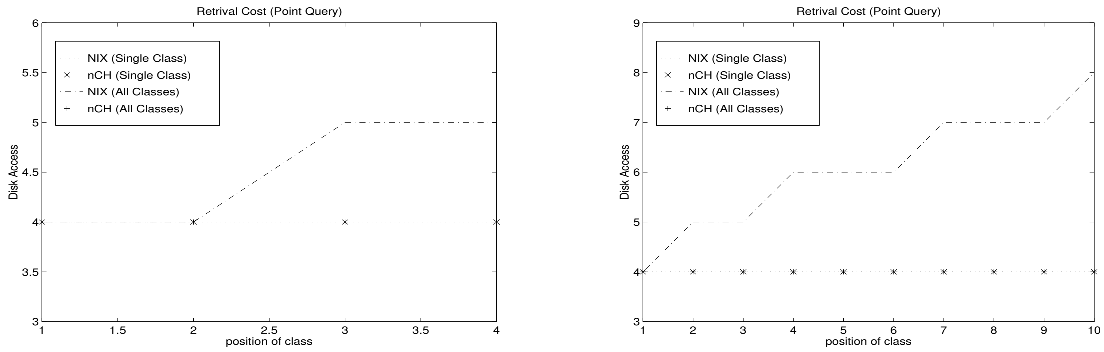
(a) (b)

Figure 23: Point Query Cost

Figure 22 shows the average retrieval cost of the nCH-tree and NIX organization against the length of the path expression. Increasing the length of the path does not affect the number of classes in each inheritance sub-hierarchy, but the number of inheritance hierarchies increases. The retrieval cost of the nCH-tree keeps very low in both cases while the retrieval cost of the NIX increases as the path length increases. This is because as the path length increase, the size of the leaf node records of the NIX also increases to store the additional information on classes. The nCH-trees, on the other hand, stores only the information on a single inheritance sub-hierarchy, and thus, it is not affected by additional classes in different positions. From the experiments that have been performed, the results match our expectation in Section 5.3, that is, navigations by the nCH-tree structures offer better performance than the NIX organization in most cases.

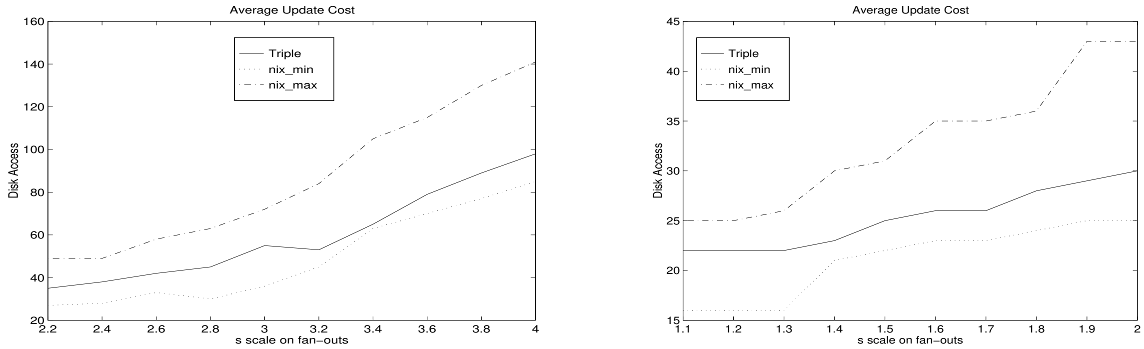
6.3 Point Query Cost

Next we compare the performance for point queries, meaning that the target answer is a single point rather than a range. In the first experiment, we vary the path length. The values of the fanout parameter fan is set to 1 for all classes except the classes rooted at the last class of the path expression, i.e. $C_{n,1}$, which is set to 20. The number of classes in each inheritance hierarchies is set to 3. The result is shown in Figure 23(a). In the second experiment, the path length is 4. The number of classes in the inheritance hierarchy is varied. The remaining set up are the same as in (1). The result is shown in Figure 23(b). In the third set of experiment measures the effect of the position of the class with a large fanout as in experiment (2) for range queries. The number of classes in each inheritance hierarchies is set to 10. The result for path lengths of 4 and 10 are shown in Figures 24(a) and 24(b), respectively. The nCH-tree structure performs better because the heights of its search trees are shorter than that for NIX.



(a) (b)

Figure 24: Point Query Cost



(a) (b)

Figure 25: Update Cost

6.4 Update Cost

Figure 25 shows the average update costs of the nCH-tree structure and the NIX index structure. We assume that the update probability of each object relationship between class i and class $i+1$, for $1 \leq i \leq 4$, is the same. We also assume that the update operation starts by only one relationship. In Figure 25, the upper bound and the lower bound of the update cost of NIX are calculated. The upper bound occurs when all the auxiliary structures have to be updated by a chain effect, the lower bound occurs when no auxiliary structure is affected. It is shown that the nCH-tree has an update cost in between the boundaries of the NIX organization. In general, the NIX has a lower update cost when the update is performed at the beginning of the path. As the updated classes shift towards the ending of the path, the update cost increases. This is because the auxiliary index need to be accessed. As the path length gets longer, it can be expected that the update cost of the NIX will be higher. The nCH-tree is seen to have a smaller increase rate.

7 Conclusion

We have presented an indexing structure for object-oriented databases, called the triple-node hierarchy. The triple-node hierarchy supports both forward and backward navigations among objects and classes along different path expressions. We argue that the performance will be better than the corresponding partial join index hierarchy. We have developed an analytical cost model and the simulation results show that the triple-node hierarchy performs better then the best previous methods known to us.

We have also addressed the need for index organizations which support both aggregation and inheritance hierarchies and have presented the nCH-tree structure which is an integration of the triple-node hierarchy and the CH-tree. The nCH-tree structure supports a fast evaluation of queries on a nested attribute issued against a class or all classes in an inheritance hierarchy along a path. We have developed an analytical cost model and our simulation results show that navigations through the nCH-tree structure offer a better retrieval performance than NIX [4] in most cases. In addition, the retrieval and update procedures are much simpler than NIX.

We have not investigated fully how to construct an optimal triple-node hierarchy for a given schema and schema paths. We note that the optimality for partial join index hierarchy is not well-defined, since the nodes in the upper levels of the hierarchy are usually much bigger than the lower level nodes, the definition of optimality in [24] whose first criterion is the number of nodes is not well justified. This would be an interesting problem for further study.

The nCH-tree structure shows the possibility of integrating the triple-node hierarchy method with other indexing methods. It would be of interest to integrate the triple-node hierarchy with, say the CD-tree, to construct a nested CD-tree organization and to compare the performance. We believe that the construction of different nested tree structures supported by the triple-node hierarchy could be a promising candidate towards the indexing support for query processing in object-oriented databases.

References

[1] E. Bertino. An indexing technique for object-oriented database. In *Proc. Int. Conf. Data Engineering*, pages 160–170, 1991.

[2] E. Bertino. Method precomputation in object-oriented database. In *Proceedings of ACM-SIGOIS and IEEE-TC-OA International Conference on Organizational Computing Systems (COCS91)*, 1991.

[3] E. Bertino and P. Foscoli. On modeling cost functions for object-oriented databases. Submitted for publication, Jan 1992.

[4] E. Bertino and P. Foscoli. Index organizations for object-oriented database systems. *IEEE Trans. Knowledge and Data Engineering*, pages 193–209, 1995.

[5] E. Bertino and C. Guglielmina. Optimization of object-oriented queries using path indices. In *Proceedings of International IEEE Workshop on Research Issues on Data Engineering: Transaction and Query Processiing (RIDE-TQP)*, pages 140–149, 1992.

[6] E. Bertino and W. Kim. Indexing Techniques For Queries On Nested Objects. *IEEE Trans. Knowledge and Data Engineering*, pages 196–214, 1989.

[7] E. Bertino and A. Quarati. An approach to support method invocations in object-oriented queries. In *Proceedings of International IEEE Workshop on Research Issues on Data Engineering: Transaction and Query Processiing (RIDE-TQP)*, pages 163–168, 1992.

[8] E. Bertino, S. Salerno, and B. Shidlovsky. Enhanced nested-inherited index for oodbms. In *Proc. International Conference on Information and Knowledge Management (CIKM)*, 1995.

[9] Sudarshan S. Chawathe, Ming-Syan Chen, and Philip S. Yu. On index selection schemes for nested object hierarchies. In *Proceedings International Conference on Very Large Database*, pages 331–341, 1994.

[10] S. Choenni, E. Bertino, and H. M. Blanken. On the selection off optimal index configuration in oo databases. In *Proc. Int. Conf. Data Engineering*, pages 526–537, 1994.

[11] K. A. Hua and C. Tripathy. Object skeletions: an efficient navigation structure for object-oriented database systems. In *Proc. Int. Conf. Data Engineering*, pages 508–517, 1994.

[12] A. Jhingran. Precomputation in a complex object environment. In *Proc. Int. Conf. Data Engineering*, 1991.

[13] A. Kemper and G. Moerkotte. Access support in object bases. In *Proceedings ACM-SIGMOD Conference on the Management of Data*, pages 364–374, 1990.

[14] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proceedings International Conference on Very Large Database*, pages 290–301, 1990.

[15] Michael Kifer, Won Kim, and Yehoshua Sagiv. Querying object-oriented databases. In *Proceedings ACM-SIGMOD Conference on the Management of Data*, pages 393–402, 1992.

[16] Christoph Kilger and Guido Moerkott. Indexing multiple sets. In *Proceeding of the 20th VLDB Conference Santiago, Chile*, pages 180–191, 1994.

[17] K. C. Kim, W. Kim, and A. Dale. Indexing techniques for object-oriented database. In W. Kim and F. H. Lockovsky, editors, *object-oriented concepts, Databases, and Applications*, pages 371–394. Addison-Wesley, 1989.

[18] C. C. Low, H. Lu, B. C. Ooi, and J. Han. Efficient access methods in deductive and object-oriented databases. In *Proceedings International Conference on Deductive and Object-Oriented Databases*, pages 68–84, 1991.

[19] C. C. Low, B. C. Ooi, and H. Lu. H-trees: A dynamic associative search index for oodb. In *Proceedings ACM-SIGMOD Conference on the Management of Data*, pages 134–143, 1992.

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

[21] Sridhar Ramaswamy and Paris C. Kanellakis. Oodb indexing by class-division. In *Proceedings ACM-SIGMOD Conference on the Management of Data*, pages 139–150, 1995.

[22] B. Sreenath and S. Seshadri. The hcc-tree: An efficient index structure for object oriented database. In *Proceeding of the 20th VLDB Conference Santiago, Chile*, pages 203–213, 1994.

[23] P. Valduriez. Join indices. *ACM Trans. Database Systems*, pages 218–246, 1987.

[24] Z. Xie and J. Han. Join index hierarchies for supporting efficient navigations in object-oriented databases. In *Proceeding of the 20th VLDB Conference Santiago, Chile*, pages 522–533, 1994.

[25] S. B. Yao. Approximating block accesses in database organizations. *Communications of the ACM*, pages 260–261, April 1977.

8 Appendix A: Cost Model for Tripel-Node Hierarchies

An analytical model is constructed to study the performance of the triple-node hierarchy. The study focuses on several major performance measurements, including storage size, navigation cost and update cost (propagated over the triple-node hierarchy). Table 2 lists some system parameters used in the cost analysis.

8.1 Storage

The probability that an object in class C_{j-1} does not reference a certain object in class C_j is

$$\frac{\binom{|C_j|-1}{f_{j-1}}}{\binom{|C_j|}{f_{j-1}}} = 1 - \frac{f_{j-1}}{|C_j|}$$

α	average page occupancy factor, = 70%
B_{fan}	fan out of the B^+ tree, $B_{fan} = \lceil \frac{P \times \alpha}{PP+OIDL} \rceil$ where PP is the size of a page pointer, = 4
$\lvert C_i \rvert$	number of objects in class C_i
f_i	average number of references from an object in C_i to objects in C_{i+1}
n	length of access path
$OIDL$	size of object identifier, = 8
P	net size of page, = 4096
$shar_i$	average number of objects in class C_i referencing the same object in C_{i+1}, = $\frac{\lvert C_i \rvert \times f_i}{\lvert C_{i+1} \rvert}$
tn	size of a tuple in a triple-node, = $OIDL \times 2 + type + tnc$ where tnc is the size of the counter in a tuple of a triple-node, = 4 and $type$ is the size of the TYPE flag in a tuple of a triple-node, = 1

Table 2: Some Database Parameters

The probability that m objects in class C_{j-1} do not reference a certain object in class C_j is

$$\left(1 - \frac{f_{j-1}}{|C_j|}\right)^m$$

The probability that a certain object in class C_j is referenced by m objects in class C_{j-1} is

$$1 - \left(1 - \frac{f_{j-1}}{|C_j|}\right)^m$$

Therefore, the number of objects in class C_j referenced by these m objects in class C_{j-1} is

$$|C_j| \times \left(1 - \left(1 - \frac{f_{j-1}}{|C_j|}\right)^m\right)$$

Hence, the average number of distinct objects in class C_j referenced by a set of m objects in class C_i is

$$fwd(i,j,m) = \begin{cases} \lceil p(|C_j|, f_{j-1}, m) \rceil & , \quad j = i+1 \\ \lceil p(|C_j|, f_{j-1}, fwd(i, j-1, m)) \rceil & , \quad j > i+1 \end{cases}$$

and the average number of distinct objects in class C_i referencing a set of m objects in class C_j is

$$bwd(i,j,m) = \begin{cases} \lceil p(|C_i|, shar_i, m) \rceil & , \quad j = i+1 \\ \lceil p(|C_i|, shar_i, bwd(i+1, j, m)) \rceil & , \quad j > i+1 \end{cases}$$

where $p(x,y,z) = x \times (1 - (1 - \frac{y}{x})^z)$

The number of tuples in a triple-node $Tri(\perp, i, j)$ (and $Tri(i, j, \top)$) is

$$|Tri(\perp, i, j)| = |C_i| \times fwd(i, j, 1)$$

The number of tuples in a triple-node Tri(i,j,k) is

$$|Tri(i,j,k)| = |C_j| \times fwd(j,k,1) + |C_i| \times fwd(i,j,1)$$

The number of pages needed to store a triple-node is

$$\|Tri(i,j,k)\| = \left\lceil \frac{tn \times |Tri(i,j,k)|}{P \times \alpha} \right\rceil$$

8.2 Query Cost

From Yao [25], for accessing k records randomly distributed in a file of n records stored in m pages, the expected number of page access is given by

$$\begin{aligned} Yao(k,m,n) &= \left\lceil m \times \left(1 - \frac{\binom{n-\frac{n}{m}}{k}}{\binom{n}{k}}\right)\right\rceil \\ &= \left\lceil m \times \left(1 - \prod_{i=1}^{k} \frac{n - \frac{n}{m} - i + 1}{n - i + 1}\right)\right\rceil \end{aligned}$$

Following Valduriez [23], the number of disk access for a forward navigation from a set of n_i objects in class C_i to objects in C_j in a triple-node $Tri(\perp, i, j)$ is

$$1 + Yao\left(n_i, \left\lceil \frac{|C_i|}{B_{fan}} \right\rceil, |C_i|\right) + Yao(n_i, \|Tri(\perp, i, j)\|, |C_i|)$$

As in [24], it is assumed that a B^+-tree has two levels. In the above equation, the constant "1" indicates that one page is accessed for retrieving the root node. The second term is the number of page access to the leaf pages of the B^+-tree in order to find the page pointers for n_i object identifiers. The last term is the number of page access required to find the corresponding n_i object identifiers. The number of disk access for a forward navigation from a set of n_j objects in class C_j to objects in class C_k in a triple-node $Tri(i,j,k)$ is

$$1 + Yao\left(n_j, \left\lceil \frac{|C_j|}{B_{fan}} \right\rceil, |C_j|\right) + Yao(n_j, \|Tri(i,j,k)\|, |C_j|)$$

The case for backward navigation is similar.

8.3 Update Cost

Suppose there is an update on objects in class C_k which causes an update operation $\delta(k, k+1)$ (either insertion or deletion) on a triple-node $Tri(X, k, k+1)$, $0 \leq X < k$, with n_k identifiers of distinct objects of class C_k and n_{k+1} identifiers of distinct objects of class C_{k+1} in the tuples in $\delta(k, k+1)$.

The cost of updating the triple-node $Tri(X, k, k+1)$ is

$$\left[1 + Yao\left(n_k, \left\lceil \frac{|C_k|}{B_{fan}} \right\rceil, |C_k|\right) + Yao(n_k, \|Tri(X, k, k+1)\|, |C_k|)\right]$$
$$+Yao(n_k, \|Tri(X, k, k+1)\|, |C_k|)$$

In this equation, the first term (inside the square brackets) is the cost of navigating n_k tuples from the triple-node, and the last term is the cost of write-back operations for updating the related pages.

Updates in triple-node $Tri(X, k, k+1)$ will generate a set of update operations $\delta(X, k+1)$. The number of identifiers of distinct objects of class C_X in $\delta(X, k+1)$ is estimated to be $bwd(X, k, n_k)$ and the number of identifiers of distinct objects of class C_{k+1} in $\delta(X, k+1)$ is n_{k+1}.

Similarly, the cost of updating the triple-node $Tri(k, k+1, Y)$, for $Y > k+1$ is

$$1 + Yao\left(n_{k+1}, \left\lceil \frac{|C_{k+1}|}{B_{fan}} \right\rceil, |C_{k+1}|\right) + Yao(n_{k+1}, \|Tri(k, k+1, Y)\|, |C_{k+1}|)$$
$$+Yao(n_{k+1}, \|Tri(k, k+1, Y)\|, |C_{k+1}|)$$

and the set of update operations $\delta(k, Y)$ generated will consists of $fwd(k+1, Y, n_{k+1})$ identifiers of distinct objects of class C_Y and n_k identifiers of distinct objects of class C_k.

In general, for an update operation $\delta(i, j)$, with n_i identifiers of distinct objects of class C_i and n_j identifiers of distinct objects of class C_j in the tuples in $\delta(i, j)$,

the cost of updating the triple-node $Tri(X, i, j)$ is

$$1 + Yao\left(n_i, \left\lceil \frac{|C_i|}{B_{fan}} \right\rceil, |C_i|\right) + Yao(n_i, \|Tri(X, i, j)\|, |C_i|) + Yao(n_i, \|Tri(X, i, j)\|, |C_i|)$$

the cost of updating the triple-node $Tri(i, j, X)$ is

$$1 + Yao\left(n_j, \left\lceil \frac{|C_j|}{B_{fan}} \right\rceil, |C_j|\right) + Yao(n_j, \|Tri(i, j, X)\|, |C_j|) + Yao(n_j, \|Tri(i, j, X)\|, |C_j|)$$

the cost for updating the triple-node $Tri(\perp, i, j)$ is

$$1 + Yao\left(n_i, \left\lceil \frac{|C_i|}{B_{fan}} \right\rceil, |C_i|\right) + 2 \times Yao(n_i, \|Tri(\perp, i, j)\|, |C_i|)$$

the cost for updating the triple-node $Tri(i, j, \top)$ is

$$1 + Yao\left(n_j, \left\lceil \frac{|C_j|}{B_{fan}} \right\rceil, |C_j|\right) + 2 \times Yao(n_j, \|Tri(i, j, \top)\|, |C_j|)$$

h_{ch}	Internal height of the nCH-tree
$h_{i,j,k}$	Internal height of the triple-node $Tri(i,j,k)$
N_v	number of contiguous key values in the range specified for a given query
N_e	number of entries in a leaf page
kl	key length =2
n_l	number of nodes at level l in a B^+-tree
of	offset length =2

Table 3: System Parameters

$D_{i,j}$	Number of distinct values for attribute A_i of class $C_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq nc_i$
D_i	Number of distinct values for attribute A_i for all instances in the inheritance sub-hierarchy rooted at class $C_{i,1}$; $D_i = \sum_{j=1}^{nc_j} D_{i,j}$
$d_{i,j}$	Number of instances of class $C_{i,j}$, having a value different from Null for attribute A_i, $1 \leq i \leq n$ and $1 \leq j \leq nc_i$
$fan_{i,j}$	Average number of references to members of class $C_{i+1,1}$, contained in the attribute A_i for an instance of class $C_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq nc_i$. The average is computed with respect to all the instances of class $C_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq nc_i$.
$N_{i,j}$	Number of instances of class $C_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq nc_i$
Nh_i	Number of members of class $C_{i,1}$, $1 \leq i \leq n$; $Nh_i = \sum_{j=1}^{nc_i} N_{i,j}$
nc_i	Number of classes in the inheritance sub-hierarchy rooted at class $C_{i,1}$, $1 \leq i \leq n$

Table 4: Database Parameters

9 Appendix B: Cost Model for nCH-tree

An analytical model is constructed to study the performance of the nCH-tree. The study focuses on several major performance measurements, including the storage cost, cost of navigations and the cost of updates. The cost model constructed in the previous sections considers only the simple case of an aggregation hierarchy. Here such a cost model is extended to cover inheritance hierarchies.

Since we would compare our method with NIX, we shall set up experiments that follow the assumptions made by NIX, including the restriction that only query paths of the form $P(i,n)$, $i < n$ are considered. Our cost model is based on this assumption.

The parameters of $n, P, OIDL, tn, \alpha$, and B_{fan} are defined in Table 2. Table 3 lists some additional system parameters used in the cost analysis. Table 4 and Table 5 list the database parameters and derived parameters. These parameters describe the classes in a path expression $P = \langle (C_{1,1}), A_2(C_{2,1}), ..., A_n(C_{n,1}) \rangle$. The details of the estimation of the derived parameters are shown in Appendix A. We also make the following assumptions:

1. All key values have the same length.

$\overline{k}_{i,j}$	Number of instances of class $C_{i,j}$ having the same value for the nested attribute A_n $1 \leq i \leq n$, $1 \leq j \leq nc_i$
$\overline{fan}_{i,j}$	Average number of objects held in the nested attribute A_n of an instance of the class $C_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq nc_i$. The average is computed with respect to all the instances of class $C_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq nc_i$
$RefByh(i,y,k)$	average number of values contained in the nested attribute A_y for a set of k members of class $C_{i,1}$, with $1 \leq i \leq y \leq n$ and $1 \leq k \leq D_y$.
$Refh(i,y,k)$	average number of members of class $C_{i,1}$ having as value of the nested attribute A_y a value in a set of k elements, with $1 \leq i \leq y \leq n$.
$Def_{A_n}(i,j)$	Number of instances of class $C_{i,j}$ having at least one value different from Null, for the nested attribute A_n, $1 \leq i \leq n$, $1 \leq j \leq nc_i$

Table 5: Derived Parameters

2. The number of classes containing instances with the indexed key value is the same for each key value.
3. Key values are uniformly distributed among instances.
4. Each non-leaf (non-root) index node has the same fanout.
5. The cardinality of a class in a class hierarchy is independent of the cardinality of any of its super/subclasses, e.g., abstract class has no instances at all.

These assumptions are commonly found in analytical models for database access structures.

9.1 Storage

The average size of a primary (leaf-node) record of the nested CH-tree $nCH(j,n)$ is:

$$XCH_j = OIDL \times \left\lceil \sum_{i=1}^{nc_j} \overline{k}_{j,i} \right\rceil + kl + (OIDL + of) \times nc_j$$

The number of tuples in a triple-node $Tri(i,j,k)$ is

$$|Tri(i,j,k)| = Nh_j \times RefByh(i,j,1) + Nh_j \times Refh(j,k,1)$$

The number of pages needed to store a triple-node is

$$\|Tri(i,j,k)\| = \left\lceil \frac{tn \times |Tri(i,j,k)|}{P \times \alpha} \right\rceil$$

9.2 Query Cost

A query is evaluated by looking up an nCH-tree. The average number of pages required to store a leaf-node record of the nCH-tree $nCH(j, n)$ is given by:

$$np_{ch} = \left\lceil \frac{XCH_j}{P} \right\rceil$$

Let c be the number of classes targeted by a query, i.e. c represents the number of classes whose instances must be retrieved from the index lookup. The average number of pages accessed for an index lookup for a point query targeting at $P(j, n)$ is given by:

$$C_{ch}(retrieve_{point}) = \begin{cases} h_{ch} + 1 & , \quad XCH_j \leq P \\ h_{ch} + npa_{ch} + \left(1 - \frac{npa_{ch}}{np_{ch}}\right) & , \quad XCH_j > P \end{cases}$$

where $npa_{ch} = Yao(c, np_{ch}, nc_j)$.

The term npa_{ch} represents the number of pages that are surely accessed [7], and $\left(1 - \frac{npa_{ch}}{np_{ch}}\right)$ is the probability of accessing another page in the case when the class directory is not stored in one of the npa_{ch} pages. If the target of the query is the set of all classes in the inheritance sub-hierarchy, we have $c = nc_j$.

The number of entries in a leaf page of an nCH-tree $nCH(j, n)$ is :

$$N_e = \left\lfloor \frac{P}{XCH_j} \right\rfloor$$

the average number of pages accessed for an index lookup for a range query having N_v number of contiguous values in the range is given by:

$$C_{ch}(retrieve_{range}) = \begin{cases} h_{ch} + \left\lceil \frac{N_v}{N_e} \right\rceil & , \quad XCH_j \leq P \\ h_{ch} + \left(npa_{ch} + \left[1 - \frac{npa_{ch}}{np_{ch}}\right]\right) N_v & , \quad XCH_j > P \end{cases}$$

9.3 Update Cost

Consider two types of update operations in the relationship between two classes:

- Insert an instance $O_{i,t}$ into the attribute A_i of another instance $O_{i-1,s}$
- Delete an instance $O_{i,t}$ from the attribute A_i of another instance $O_{i-1,s}$

[7] Recall that $Yao(k, m, n)$ is the expected number of page accesses for accessing k records which are randomly distributed in a file of n records.

In both cases, an update operation $\delta(i-1,i)$ will be generated (either insertion or deletion). This update operation will be propagated from the triple-node hierarchy to the nCH-tree.

There are two cases to be considered.

1. The object $O_{i,t}$ has a Null value for the nested attribute A_n;
2. The object $O_{i,t}$ has values different from Null for the nested attribute A_n;

For case 1, no leaf node record in the nCH-tree is updated. Only the triple-node hierarchy is updated and the cost is estimated as follows[8]:

For an update operation $\delta(i,j)$, with n_i identifiers of distinct objects of class C_i and n_j identifiers of distinct objects of class C_j in the tuples in $\delta(i,j)$,

the cost of updating the triple-node $Tri(X,i,j)$ is

$$C_{tri\_retrive}(n_i, X, i, j) = VisBtree\,(n_i, Nh_i, h_{X,i,j}) + Yao(n_i, \|Tri(X,i,j)\|, Nh_i)$$

the cost of updating the triple-node $Tri(i,j,X)$ is given by:

$$C_{tri\_retrive}(n_j, i, j, X) = VisBtree\,(n_j, Nh_j, h_{i,j,X}) + Yao(n_j, \|Tri(i,j,X)\|, Nh_j)$$

where $VisBtree(k,n,h)$ denotes the cost of the batch scanning of a B^+-tree of height h with a total number of n keys to retrieve k keys. Suppose that the leaf node records may be stored in only one page, then the value of $VisBtree(k,n,h)$ is given by :

$$VisBtree(k,n,h) = \sum_{l=1}^{h} npa_l$$

where npa_l denotes the number of page accessed at level l in a B^+-tree:

$$npa_l = \begin{cases} Yao(k, n_l, n) & , \quad l = h \\ Yao(npa_{l+1}, n_l, n_{l+1}) & , \quad l < h \end{cases}$$

and n_l denotes the number of nodes at level l in a B^+-tree, with $n_h = \lceil \frac{n}{B_{fan}} \rceil$ and $n_l = \lceil \frac{n_{l+1}}{B_{fan}} \rceil, 1 \le l < h$.

For case 2, both the nCH-tree and the triple-node hierarchy are updated. The number of updated leaf pages of the nCH-tree $nCH(j,n)$ is

$$DNP_{ch}(i,t) = \begin{cases} Yao(\overline{fan}_{i,t}, LP_{ch}, D_n) & , \quad XCH_j \le P \\ \overline{fan}_{i,t} \times \left(NPC_{ch} + \left\lceil 1 - \frac{NPC_{ch}}{np_{ch}} \right\rceil\right) & , \quad else \end{cases}$$

[8]see Sections 3.2 and 8.3 for the details of the update propagation

where $NPC_{ch} = \lceil np_{ch}/nc_j \rceil$, and

LP_{ch} is the number of leaf-node pages of the nCH-tree.

Since an update requires the reading and writing of relevant pages, both of which may involve disk I/O's, the update cost of the nCH-tree is given by:

$$CH_{update}(i,t) = 2 \times DNP_{ch}(i,t) + VisBtree\left(\overline{fan}_{i,t}, D_n, h_{ch}\right)$$

The probability that case 1 occurs is:

$$P1 = \frac{N_{i,t} - Def_{A_n}(i,t)}{N_{i,t}}$$

The total update cost can be estimated by merging case 1 and case 2.

Appendix C: Evaluation of some Parameters

d_i	Number of members of class $C_{i,1}$, having a value different form Null for attribute A_i, $1 \leq i \leq n$; $d_i = \sum_{j=1}^{nc_i} d_{i,j}$
fan_i	Average number of references to members of class $C_{i+1,1}$, contained in the attribute A_i of a member of class $C_{i,1}$, $1 \leq i \leq n$. The average is computed with respect to all the members of class $C_{i,1}$, $1 \leq i \leq n$. The difference between this parameter and the previous is that this parameter is obtained as the average evaluated on all members of a class hierarchy, while in the previous the average is for each class. Note: $fan_i = \frac{\sum_{j=1}^{nc_i} d_{i,j} \times fan_{i,j}}{\sum_{j=1}^{nc_i} d_{i,j}}$
$k_{i,j}$	Number of instances of class $C_{i,j}$, having the same value for attribute A_i, $1 \leq i \leq n$ and $1 \leq j \leq nc_i$; $k_{i,j} = \frac{d_{i,j} \times fan_{i,j}}{D_{i,j}}$
kh_i	Number of members of class $C_{i,1}$, having the same value for attribute A_i, $1 \leq i \leq n$ Note: $kh_i = \frac{d_i \times fan_i}{D_i}$

Table 6: Additional Database Parameters

The probability that a given O_{i+1}, member of class $C_{i+1,1}$ and belonging to the definition domain of attribute A_i of cardinality D_i, is not referenced by a given member of class $C_{i,1}$ is: $1 - (fan_i/D_i)$.

If $kh_i > 1$, the probability that $Oi + 1$ is not referenced by any object belonging to a set of member of class $C_{i,1}$, with attribute A_i having a set of values $\{O_{i,j}^1, O_{i,j}^2, ..., O_{i,j}^k\}$ all different form Null is the following :

$$PrD(i,k) = \left(1 - \frac{fan_i}{D_i}\right)^k$$

If kh_i =1,

$$PrD'(i,k) = \prod_{y=0}^{k-1} \left(1 - \frac{fan_i}{D_i - y * fan_i}\right)$$

Let $RefBy(i,s,y,k)$ $(0 \le i \le y \le n,\ 1 \le s \le nc_i$ and $1 \le k \le D_y)$ denote the number of values contained in the nested attribute A_y for a set of k instances of class $C_{i,s}$.

$$RefBy(i,s,y,k) = \begin{cases} D_{i,s} * (1 - v(i,s) * Pr(i,s,k) - \\ (1 - v(i,s)) * Pr'(i,s,k)) & y = i \\ D_y * (1 - v'(i) * PrD(y, RefBy(i,s,y-1,k) * P_{A_y - 1}) - \\ (1 - v'(i)) * PrD'(y, RefBy(i,s,y-1,k) * P_{A_y} - 1)) & y > i \end{cases}$$

where

$$v(i,s) = \begin{cases} 1 & k_{i,s} > 1 \\ 0 & else \end{cases}$$

$$v'(i) = \begin{cases} 1 & kh_i > 1 \\ 0 & else \end{cases}$$

The number of values contained in the nested attribute A_y for a set of k members of class $C_{i,1}$ is

$$RefByh(i,y,k) = \begin{cases} D_i * (1 - v'(i) * PrD(i,k) - \\ (1 - v'(i)) * PrD'(i,k)) & y = i \\ D_y * (1 - v'(i) * PrD(y, RefByh(i,y-1,k) * P_{A_y - 1}) - \\ (1 - v'(i)) * PrD'(y, RefByh(i,y-1,k) * P_{A_y} - 1)) & y > i \end{cases}$$

Thus, the average number of objects held in the nested attribute A_n of an instance of the class $C_{i,j}$, $1 \le i \le n$, $1 \le j \le nc_i$ is:

$$\overline{fan}_{i,j} = RefBy(i,j,n,1)$$

For the formulations $\overline{k}_{i,j}$ and $Def_{A_n}(i,j)$, two other parameters must be derived; $Ref(i,j,y,k)$ and $Refh(i,y,k)$. These parameters represent, respectively, the average number of instances of class $C_{i,j}$ which have a value in a set of k elements ($1 \le i \le y \le n$ and $1 \le j \le nc_i$) as the value of the nested attribute A_y, and the average number of members of class $C_{i,1}$ which have a value in a set of k elements ($1 \le i \le y \le n$) as the value of the nested attribute A_y. The formulation of these parameters follows the same approach as the formulation of the parameters $RefBy(i,j,y,k)$ and $RefByh(i,y,k)$. A detailed description of the formulations of these parameters can be found in [3]. The following is obtained: $\overline{k}_{i,j} = Ref(i,j,n,1)$ and $Def_{A_n}(i,j) = Ref(i,j,n,D_n)$.