Join Index Hierarchies for Supporting Efficient Navigations in Object-Oriented Databases *

Zhaohui Xie
School of Computing Science
Simon Fraser University
Burnaby, B.C., V5A 1S6, Canada
zhaohui@cs.sfu.ca

Jiawei Han
School of Computing Science
Simon Fraser University
Burnaby, B.C., V5A 1S6, Canada
han@cs.sfu.ca

Abstract

A join index hierarchy method is proposed to handle the "goto's on disk" problem in object-oriented query processing. The method constructs a hierarchy of join indices and transforms a sequence of pointer chasing operations into a simple search in an appropriate join index file, and thus accelerates navigation in object-oriented databases. The method extends the join index structure studied in relational and spatial databases, supports both forward and backward navigations among objects and classes, and localizes update propagations in the hierarchy. Our performance study shows that partial join index hierarchy outperforms several other indexing mechanisms in object-oriented query processing.

1 Introduction

Query processing and optimization is crucial to the performance of object-oriented database systems. Substantial researches into query processing and query optimization in object-oriented databases have been conducted in recent years with encouraging progress reported, e.g., [2, 3, 4, 5, 7, 13, 15, 17].

Since object-oriented databases support complex data objects and enable explicit and natural representation of logical relationships among complex objects via class/subclass hierarchies, attributes, methods, object identities, etc., navigation among different classes and objects via class hierarchies and/or class composition hierarchies is an essential operation. Navigations from one object in a class to objects in other classes are essentially "pointer chasing" (using object identity "OID" references) operations which may cause significant performance degradation because the objects to be accessed may be stored at widely scattered locations and many disk read operations may be required to fetch them into main memory [4]. The attempts to solve this problem can be classified into three classes of techniques: the indexing method, the read-ahead buffering method (e.g., [11]), and parallel complex object assembly method (e.g., [5]).

Following the philosophy of indexing methods, a join index hierarchy method is proposed in this paper, which extends the join index technique developed in relational databases [16] and its variations in spatial databases [12, 9], constructs hierarchies of join indices to accelerate navigations via a sequence of objects and classes. In a broad sense, a join index in our method stores the pairs of identifiers of objects of two classes that are connected via *direct* or *indirect* logical relationships. Those formed by *direct* logical relationships are called **base join indices**; whereas those represent *indirect* logical relationships are called **derived join indices**. Base and derived join indices form a join index hierarchy. A join index hierarchy supports navigations through a sequence of classes in either forward or backward navigation direction and supports efficient update propagation starting with the base join indices

*research partially supported by NSERC Grant OGP03723, IRIS-2 Grants HMI-5 and IC-2, and the Centre for Systems Science of Simon Fraser University.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

by localizing update propagations in the hierarchy.

The following considerations motivate the proposal of the join index hierarchy structures.

First, by construction of join index hierarchies, the "pointer chasing" problem, that is, accessing objects and their properties via a sequence of referencing pointers to widely scattered disk locations, is transformed into simple accessing of appropriate join index files. This may significantly reduce the I/O accessing cost in object-oriented query processing. The price for this I/O cost reduction is the increase of space for storing join index files, which is practically implementable since large inexpensive disk memories are available with reasonable cost based on the current hardware technology.

Secondly, with join index hierarchies, appropriate join index files for specific navigation operations can be selected by consulting the index hierarchy directory. Moreover, update propagation can be localized to a few base and derived join index files in the hierarchy. Both forward and backward navigations can be supported with minimum storage and update overheads. The structure is especially good for frequent navigations and infrequent updates.

Thirdly, using join index hierarchies, object-at-a-time styled navigation is transformed into efficient, set-oriented and associative access of join indices. Moreover, it supports navigations among objects connected not only via a sequence of attribute relationships but also via a sequence of methods and deduction rules. This is accomplished by precomputing methods and rules and storing the related information in join indices. By doing so, the object-at-a-time evaluation of computationally intensive methods or deduction-intensive rules can be transformed into efficient and set-oriented accessing of precomputed relationships. Moreover, retrieval from either directions becomes available even for methods and deduction rules.

Fourthly, in some cases, the join of some classes on certain attributes may generate a substantially large join index file because of its large join selectivity, or some class may sustain regular and frequent updates. Joins involving such kind of characteristics should be considered as "fire walls" in the construction of join index hierarchies. The system should prohibit the construction of such join indices or the merge of such join indices into the hierarchy in order to avoid the potential explosion on the size of join index files or the heavy cost of updates. Queries involving such joins can be processed by performing concrete joins or using the base join index files, if available.

The remaining of the paper is organized as follows. In section 2, following a preliminary survey of the previous work on join indices and object-oriented navigation techniques, three join index hierarchy structures are introduced. In section 3, the construction and update maintenance of join index hierarchies are studied. In section 4, an analytical evaluation of three join index hierarchy structures and some potentially competitive associative indexing structures are presented. In section 5, implementation considerations, improvements and extensions of the approach are discussed. Finally, the study is summarized in section 6.

2 Join Index Hierarchy

2.1 Previous work

Join index structure was first proposed by Valduriez [16] for optimizing join and semijoin operations in relational databases. A join index file stores pairs of the surrogates of joining tuples from two relations, which transforms expensive joins to selections in join index files. Since efficient accessing structures can be constructed on join indices, it has been shown that relational join using join index structures outperforms other relational join methods in many cases [16].

Join index structures can be applied to different application domains. For example, a spatial join index structure was developed by Rotem [12] and organized in the form of grid files. Further, certain precomputed information (e.g. distance) can be associated with such spatial join index structure to speed up query processing as shown by Lu and Han [9].

In the studies of query optimization in object-oriented databases, special attention has been paid to path indices which associate the values of nested attributes with the objects in the head class of a path expression, e.g., by Maier and Stein [10], Bertino and Kim [2] and Bertino [1]. In Maier and Stein [10], a series of index components, indices on each level of the nested attributes, are maintained for the purpose of update propagations. In Bertino and Kim [2], three index structures are presented: nested index, path index and multiindex, which have been later extended to handle inheritance of classes appearing in a path expression [1]. The nested index structure facilitates associative search and update by storing together the key values of the tail attribute and the objects of the head class and intermediate objects of a path expression in primary records. An auxiliary index, which basically keeps direct reference information between objects, together with the extra information in the primary records are used to propagate updates. The nested index structure in general outperforms the other two index structures [1]. Shekita and Carey [14] describe a mechanism called field replications which replicate the values of the nested at-

tributes. In-place field replication stores the replicated data with the objects, whereas separate field replication stores the replicated data in a separated place. The separated replication is used to solve the issue of updating the shared replicated data. Inverted path structures, which are similar to the index components in [10], are used to support update propagation. These approaches support only the associated retrieval of objects through nested attributes but not navigations in both directions along a reference chain.

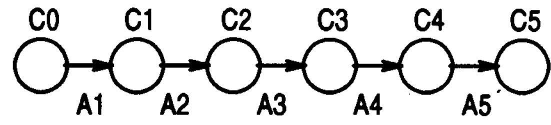
Kemper and Moerkotte [6] present a data structure called *access support relation* which keeps the identifiers of those objects connected by attribute relationships in a path expression and can span over the reference chains of a path expression. Several alternatives which include full, canonical, left and right extensions and decomposition of access support relations for a given path expression are discussed. The optimal one is determined according to the domain-specific information such as the probabilities of different types of queries and updates. The join index hierarchy approach proposed here shares certain similarity with this approach. However, the storage size of each component in an access support relation could be large because all the identifier sequences of the joinable objects along an object path corresponding to the component are stored, and any two objects in the two classes could be connected by more than one object path. Further, an update on one object may need to be propagated to several components or to the entire access support relation, which could be costly.

2.2 Preliminaries

Following the previous research, a join index hierarchy structure is proposed here to support efficient navigation through multiple object classes. The variations of a join index hierarchy can be constructed based on the richness of the derived join index structures. Three kinds of structures: ***based-only***, ***complete***, and ***partial***, are investigated in terms of their construction, navigation and update propagation.

A **database schema** is a directed graph in which the nodes correspond to classes, and edges to relationships between classes. 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, labeled with A_k. Moreover, if for $i = 0, 1, \ldots, n-1$, there is a directed edge from C_i to C_{i+1}, labeled with A_{i+1}, in a database schema, then $\langle C_0, A_1, C_1, A_2, \ldots, A_n, C_n \rangle$ is a **schema path**.

Regarding to a schema path $\langle C_0, A_1, C_1, A_2, \ldots, A_n, C_n \rangle$ over a database schema, a **join index file (node)** $JI(i,j)$ $(1 \leq i < j \leq n)$ consists of a set of tuples



A Schema Path in a Database Schema

Figure 1: A Schema Path of Length 5

$(OID(o_i), OID(o_j), m)$, where o_i and o_j are objects of classes C_i and C_j respectively, and there exists an object path $\langle o_i, o_{i+1}, \ldots, o_{j-1}, o_j \rangle$ such that for $k = 0, 1, \ldots, j-i-1$, 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.

Join index nodes connecting different object classes along a schema path form a **join index hierarchy**, denoted as $JIH(C_0, A_1, C_1, A_1, \ldots, A_n, C_n)$, or simply $JIH(0, n)$. The longest join index path, $JI(0, n)$, is the **root** of the hierarchy. Each node $JI(i, j)$ where $j - i > 1$ may have 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 $i = 0, 1, \ldots, n-1$, are at the bottom of the hierarchy, and are therefore, called **base join indices**.

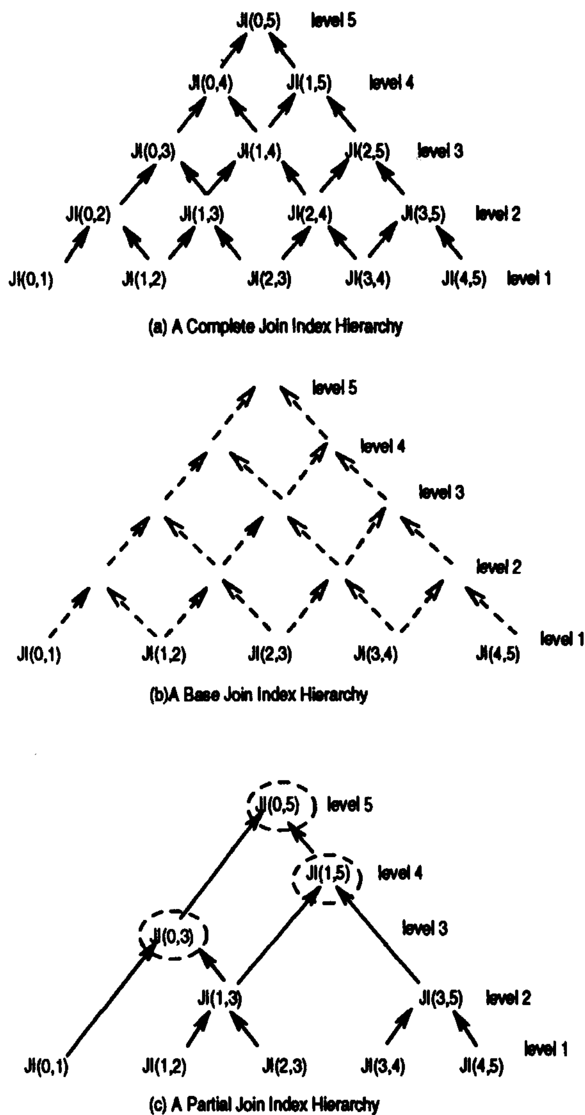
Figure 1 shows a schema path of length 5 on a class composition hierarchy and Figure 2(a)(b)(c) illustrates the following three join index hierarchy structures.

1. **A complete join index hierarchy (C-JIH)**, as shown in Figure 2(a), consists of a complete set of all the possible base and derived join indices. It supports navigations between any two directly or indirectly connected object classes along the schema path.

2. **A base join index "hierarchy" (B-JIH)**, as shown in Figure 2(b), consists of only base join indices. It supports direct navigations only between any two adjacent classes. It cannot be entitled as a "hierarchy" in a rigorous sense but can be viewed as a degenerate hierarchy with all the higher level join index nodes missing, and these nodes can be derived from the base join indices.

3. **A partial join index hierarchy (P-JIH)**, as shown in Figure 2(c), consists of a proper subset of the set of derived join indices in a complete join index hierarchy. It supports direct navigations between a pre-specified set of object class pairs since it materializes only the corresponding join indices and their related auxiliary (derived) join indices.

Figure 2(c) demonstrates a typical partial join index



(a) A Complete Join Index Hierarchy

(b)A Base Join Index Hierarchy

(c) A Partial Join Index Hierarchy

Figure 2: Three Kinds of Join Index Hierarchies Corresponding to the Schema Path in Figure 1

hierarchy which supports direct navigations between C_0 and C_5, C_1 and C_5, and C_0 and C_3. Their corresponding JI nodes: $JI(0,5)$, $JI(1,5)$ and $JI(0,3)$, circled in the figure, are called *target nodes*. Notice that a materialized intermediate level node $JI(i,j)$ may be used not only for supporting navigations between C_i and C_j but also (and sometimes more importantly) for accelerating update propagations from the base join indices to higher level join indices such as $JI(0,5)$. For example, if there were no intermediate level join index nodes in the hierarchy $JIH(0,5)$, four join-like (defined later) operations are needed on average to propagate an update from the base join indices to the target nodes $JI(0,5)$, $JI(1,5)$ and $JI(0,3)$. With the help of intermediate level join indices, it takes an average of 3.2 join-like operations to propagate an update from the base join indices.

In a join index hierarchy $JIH(0,n)$, the base join index nodes $JI(i,i+1)$ (for $i = 0,\ldots,n-1$) reside at level 1, and the root node $JI(0,n)$ at level n. Although a complete join index hierarchy could be quite large, each individual join index node is usually of reasonable size. In many cases, it is unnecessary to materialize all of the join index nodes in the hierarchy since it is beneficial to support only the frequently used navigations. Given a set of frequently accessed schema paths, a partial join index hierarchy can be constructed to support the corresponding navigations.

In a join index hierarchy, a set of join index nodes which must be supported (due to frequent references) are called **target join indices**, e.g. $JI(0,5)$, $JI(1,5)$ and $JI(0,3)$ in Figure 2(a); whereas the others which are mainly used for update propagation are called **auxiliary join indices**, e.g. $JI(1,3)$, and $JI(3,5)$. Auxiliary join indices can of course be used, as a by-product, for support of the navigations between the corresponding classes. The target, auxiliary and base join indices are *materialized join indices*. The unmaterialized join indices are called *virtual join indices*.

Update propagation includes three types of updates.

1. Insert an attribute relationship A_{i+1} between an object o_i in class C_i and an object o_{i+1} in class C_{i+1}. This corresponds to inserting a tuple $(OID(o_i), OID(o_{i+1}), 1)$ to the base join index $JI(i,i+1)$.
2. Delete an attribute relationship A_{i+1} between an object o_i in class C_i and an object o_{i+1} in class C_{i+1}. This corresponds to deleting a tuple $(OID(o_i), OID(o_{i+1}), 1)$ from $JI(i,i+1)$;
3. Modify an attribute relationship A_{i+1} from that between an object $o_i \in C_i$ and another object $o_{i+1} \in C_{i+1}$ to that between $o_i \in C_i$ and $o'_{i+1} \in C_{i+1}$. This corresponds to deleting an existing tuple $(OID(o_i), OID(o_{i+1}), 1)$ from $JI(i,i+1)$ and inserting a new tuple $(OID(o_i), OID(o'_{i+1}), 1)$ to $JI(i,i+1)$.

As a notational convention, $\Delta JI(i,j)$ denotes a set of tuples being inserted into $JI(i,j)$. $\Delta JI(i,j)$ consists of tuples $(OID(o_i), OID(o_j), m)$ and $m > 0$, indicating that there are m new object paths connecting o_i and o_j. Similarly, $\nabla JI(i,j)$ represents a set of tuples being deleted from $JI(i,j)$. It consists of tuples $(OID(o_i), OID(o_j), -m)$ and $m > 0$, indicating that there are m object paths connecting o_i and o_j being deleted. $\delta JI(i,j)$ denotes $\nabla JI(i,j)$ followed by $\Delta JI(i,j)$. A join operator "$\bowtie_c$", which is similar to a join operation in relational databases, is introduced. $JI(i,k) \bowtie_c$

$JI(k,j)$ contains a tuple $(OID(o_i), OID(o_j), m_1 \times m_2)$ if there is a tuple $(OID(o_i), OID(o_k), m_1)$ in $JI(i,k)$ and a tuple $(OID(o_k), OID(o_j), m_2)$ in $JI(k,j)$. That is, if there are m_1 distinct object paths from o_i to o_k and m_2 distinct object paths from o_k to o_j, there are $m_1 \times m_2$ object paths from o_i to o_j. Notice that identical tuples, such as $(OID(o_i), OID(o_j), m_k)$ (for $k = 0, 1, \ldots, p$) are automatically merged into one with their path numbers accumulated, i.e., $(OID(o_i), OID(o_j), \sum_{k=0}^{p} m_k)$.

3 Construction and Maintenance of Join Index Hierarchies

3.1 Construction of a partial join index hierarchy

A partial join index hierarchy can be constructed in three steps: (1) find a set of necessary auxiliary join indices for a given set of target indices; (2) build the corresponding base join indices; and (3) build the target and auxiliary join indices from the lowest level up.

Example 3.1 In Figure 2(a), the join index $JI(1,5)$ can be computed from $JI(1,4)$ and $JI(4,5)$, where $JI(1,4)$ can be derived in turn from $JI(1,3)$ and $JI(3,4)$, and $JI(1,3)$ from $JI(1,2)$ and $JI(2,3)$.

The base join indices for $JI(1,5)$ are the set:

$$\{JI(1,2), JI(2,3), JI(3,4), JI(4,5)\}$$

The auxiliary join indices for supporting efficient update of $JI(1,5)$ are:

$$\{JI(1,4), JI(1,3)\}$$

Notice that there could be other choices in selecting auxiliary JIs, such as $\{JI(1,3), JI(3,5)\}$, etc. □

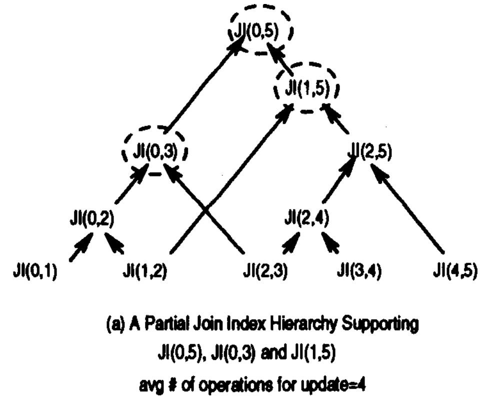
Example 3.2 To directly support the navigations between C_0 and C_5, C_1 and C_5, and C_0 and C_3, the set of target join indices are $\{JI(0,5), JI(0,3), JI(1,5)\}$, and the set of base join indices are

$$\{JI(0,1), JI(1,2), JI(2,3), JI(3,4), JI(4,5)\}.$$

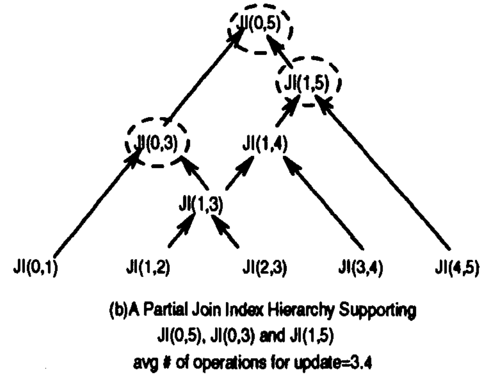
Three different kinds of partial join index hierarchies are presented in Figure 3(a)(b) and Figure 4.

The sets of auxiliary JIs which supports the three target JIs are $\{JI(0,2), JI(2,4), JI(2,5)\}$ in Figure 3(a), $\{JI(1,3), JI(1,4)\}$ in Figure 3(b) and $\{JI(1,3), JI(3,5)\}$ in Figure 4. □

Given a set of target join index nodes, the join index nodes which need to be materialized are the union



(a) A Partial Join Index Hierarchy Supporting JI(0,5), JI(0,3) and JI(1,5) avg # of operations for update=4



(b)A Partial Join Index Hierarchy Supporting JI(0,5), JI(0,3) and JI(1,5) avg # of operations for update=3.4

Figure 3: Two Partial Join Index Hierarchy Structures for Supporting JI(0,5), JI(0,3) and JI(1,5)

of the base and auxiliary sets derived from each target join index node. Since there could be more than one choice in the derivation, the optimal choice should be the one which minimizes (1) the total number of auxiliary join indices (and then the total storage costs); and (2) the total number of $\bowtie_c$ operations in updating the target join indices. This is performed by Algorithm 3.1.

Algorithm 3.1 Construction of a minimum auxiliary set of JIs

Input: A set of classes $C_0, \ldots, C_n$, and a set of target JI nodes (i.e., frequently referenced class pairs) in the schema path C_0, A_1, C_1, A_2, …, A_n, C_n.

Output: A minimum set of auxiliary JIs nodes.

Method: The method collects the set of auxiliary nodes which are used to generate the set of target nodes, and then selects those containing the minimum numbers of nodes, as shown below.

1. Starting with the set of target nodes, find S: the set of their immediate auxiliary nodes. Notice that the set of immediate auxiliary nodes

for a (target or auxiliary) node $JI(i,j)$ is $\{JI(i,k), JI(k,j)\}$ for $i < k < j$ with the removal of $JI(i,k)$ or $JI(k,j)$ if it is a target node or a base node. If there is an empty set resulted from this removal, return the empty set. Otherwise, if there are more than one such k available, each k generates one set, and the result is a set of sets. Thus, S is in the form of $\{\{JI(i,k),\ldots,JI(k,j)\}, \ldots, \{JI(i,m),\ldots,JI(m,j)\}\}$.

For each JI in the set s in S, find its immediate auxiliary nodes. If its immediate auxiliary nodes consists of l sets, $a_1, \ldots, a_l$, make l copies of s, and add each of a_i $(1 < i < l)$ to a copy, which forms l new sets. This process repeats until no new immediate auxiliary nodes can be found. The result is a set of auxiliary node sets which are used for generating the set of target nodes.

2. For each set s in the generated set of auxiliary nodes, count the number of (auxiliary) nodes. Only those with the minimum number of nodes are retained.

3. From the retained sets obtained in Step 2 (i.e., the set in which each set contains the minimum number of auxiliary nodes), calculate the number of $\bowtie_c$ operations required for updating each set and select the one which requires the minimum number of $\bowtie_c$ operations. □

Example 3.3 We examine how the algorithm works on Example 3.2. At the beginning,

$$S = \{\{JI(0,5), JI(1,5), JI(0,3)\}\}.$$

Since $JI(0,1)$ is a base join index and $JI(1,5)$ is a target join index and they can be employed to derive $JI(0,5)$, the immediate auxiliary set of $JI(0,5)$ is empty. Thus,

$$S = \{\{JI(1,5), JI(0,3)\}\}$$

The target join index $JI(1,5)$ has three immediate auxiliary sets $\{JI(1,4)\}$, $\{JI(2,5)\}$ and $\{JI(1,3), JI(3,5)\}$; whereas the target join index $JI(0,3)$ has two immediate auxiliary sets $\{JI(0,2)\}$ and $\{JI(1,3)\}$. Among these nodes, only $JI(1,4)$ and $JI(2,5)$ have nonempty auxiliary sets. The former has $\{JI(1,3)\}$ and $\{JI(2,4)\}$, and the latter has $\{JI(2,4)\}$, and $\{JI(3,5)\}$. Therefore, the set of possible auxiliary node sets should be all of their combinations, that is,

S={{$JI(1,4)$, $JI(1,3)$, $JI(0,2)$}, {$JI(1,4)$, $JI(2,4)$, $JI(0,2)$}, {$JI(2,5)$, $JI(2,4)$, $JI(0,2)$}, {$JI(2,5)$, $JI(3,5)$, $JI(0,2)$}, {$JI(1,3)$, $JI(3,5)$, $JI(0,2)$}, {$JI(1,4)$, $JI(1,3)$}, {$JI(1,4)$, $JI(2,4)$, $JI(1,3)$}, {$JI(2,5)$, $JI(2,4)$, $JI(1,3)$}, {$JI(2,5)$, $JI(3,5)$, $JI(1,3)$}, {$JI(3,5)$, $JI(1,3)$}}.

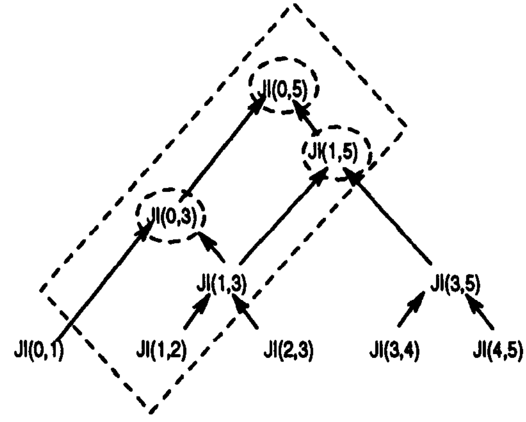


Figure 4: Build a Partial Join Index Hierarchy and Propagate Update

Both $\{JI(1,4), JI(1,3)\}$ and $\{JI(3,5), JI(1,3)\}$ have the minimum number of auxiliary join indices. The first one corresponds to the partial join index hierarchy structure in Figure 3(b), whereas the second one to that in Figure 4. The average numbers of $\bowtie_c$ operations for update propagation in Figure 3(b) and Figure 4 are 3.4 and 3.2 respectively. This is computed by averaging the sum of the numbers of all the $\bowtie_c$ operations needed for propagation of the updates on the base join index nodes. Obviously, the second partial join index hierarchy is the most preferable one. □

Algorithm 3.2 Construction of a partial join index hierarchy.

Input: A set of frequently referenced class pairs (i.e., target JI nodes) in a schema path C_0, A_1, C_1, A_2, $\ldots$, A_n, C_n and the corresponding classes.

Output: $JIH(C_0, A_1, C_1, A_1, \ldots, A_n, C_n)$, a partial join index hierarchy which supports navigations between these pairs of classes.

Method: The computation includes both finding the minimum set of auxiliary JI nodes and computing all the necessary JIs.

1. Find the minimum set of auxiliary JIs based on the set of target JIs by using Algorithm 3.1.
2. Build base JIs by computing $JI(i, i+1)$ for $i = 0, 1, \ldots, n-1$ and constructing the corresponding B^+-tree indices on i for each base JI.
3. Build auxiliary and target JIs. This is accomplished by computing the selected auxiliary JIs and/or target JIs from the bottom level up using the $\bowtie_c$ operation, and constructing the corresponding B^+-tree indices on i for each derived JI.

4. Build "reverse" JIs for searching in the reverse direction. (A reverse JI of $JI(i,j)$, $JI(j,i)$, supports the search from class j to class i via the schema path in reverse to that of $JI(i,j)$). $JI(j,i)$ is derived from $JI(i,j)$ by sorting on j in a copy of $JI(i,j)$ and constructing the B^+-tree indices on j. □

Notice that in step 3 there could be more than one pair (but at most $j-i$ pairs) of JIs of lower level nodes which can be used to compute $JI(i,j)$. A cost model should be constructed to determine the minimum cost pair. Moreover, B^+-trees can be used to build JIs for efficient retrieval and for efficient computation of JIs at higher levels.

The join index hierarchy computes the logical relationships between the objects not only in two adjacent classes but also in the "remote" classes linked via a specified schema path. It maintains both forward and backward join indices and supports both forward and backward navigations efficiently.

Furthermore, navigations on the virtual nodes (unmaterialized nodes) can still be performed efficiently using the partial join index hierarchy. For example, any virtual node in Figure 4 can be constructed by at most one join of two existing materialized JI nodes. Actually, it is easy to verify for $n \leq 6$, taking the root of $JIH(0,n)$ as the single target node, there always exists a set of minimum auxiliary nodes, with minimum update cost, and any virtual node in $JIH(0,n)$ can be obtained by at most one join of two existing (base/auxiliary) JI nodes. For example, $\{JI(0,3), JI(3,6), JI(1,3), JI(3,5)\}$ is such a minimum auxiliary node set for $JIH(0,6)$. This implies that any traversal from one object in any class to any other object class along the schema path with length less than 7 will need to search at most two (indexed) JI files using such a small partial join index hierarchy. Since one rarely constructs a $JIH(0,n)$ for $n \geq 7$ in practice, traversal along any subpath of a schema path in both directions can be performed fairly efficiently using the partial join index hierarchy.

3.2 Update maintenance of a partial join index hierarchy

An update in one class or in the relationship of one class with another may cause the update of a base join index, such as $JI(k,k+1)$ (and its update is denoted as $\delta JI(k,k+1)$). Such an update will not affect other base join indices but may affect some corresponding join indices at higher levels. It is easy to show that for an update on $JI(k,k+1)$, only the materialized $JI(i,j)$ with $i \leq k$ and $j > k$ will need to be updated accordingly. For example, if $JI(1,2)$ is updated in Figure 4, only those join indices in the dotted area need to be updated.

Algorithm 3.3 Update propagation in a join index hierarchy.

Input: A join index hierarchy $JIH(0,n)$ and $\delta JI(k,k+1)$.

Output: An updated join index hierarchy.

Method: Perform a bottom-up incremental update propagation starting at the base join index.

1. Update the base join index $JI(k,k+1)$ based on $\delta JI(k,k+1)$.
2. Update the auxiliary JIs and/or target JIs from the bottom level up using the $\bowtie_c$ operation. *This is implemented as follows (Note $\bigcup_c$ is a union operation with path count addition or deminution.).*

for level $l := 2$ to n do
  for $i := 0$ to $n-l$ do
    if $JI(i,i+l)$ is an auxiliary or target JI
      and $i \leq k$ and $i+l > k$
    then incrementally update $JI(i,i+l)$ to $JI'(i,i+l)$.
    Note: This is performed as follows.
    $\delta JI(i,i+l) := JI(i,i+p) \bowtie_c \delta JI(i+p,i+l)$, or
    $\delta JI(i,i+l) := JI(i+q,i+l) \bowtie_c \delta JI(i,i+q)$,
    where $1 \leq p < k-i$ and $k-i \leq q \leq l-1$;
  $JI'(i,i+l) := JI(i,i+l) \bigcup_c \delta JI(i,i+l)$; □

Notice that incremental updates are performed on both forward and backward join indices. Also, there are often more than one way to compute $\delta JI(i,i+l)$ in Step 2, e.g., either $JI(0,1) \bowtie_c \delta JI(1,5)$ or $\delta JI(0,3) \bowtie_c JI(3,5)$ to compute $JI(0,5)$ in Figure 4, and the choice can be determined by a cost analysis.

3.3 Base and complete join index hierarchies

A base join index hierarchy (BJIH) can be constructed and updated in a way simpler than Algorithms 3.2 and 3.3 (only Step 1 of the algorithms need to be performed) since BJIH is a degenerate hierarchy and no upward propagation need to be considered.

However, navigation between C_i and C_{i+l} in a base join index hierarchy requires the retrieval of a sequence of l base join indices:

$$JI(i,i+1), \ldots, JI(i+l-1,i+l).$$

This is the major overhead of the base join index hierarchy in comparison with the partial join index hierarchy which requires the retrieval of only one or a very small number of join indices.

Table 1: Database Parameters

Parameters	Meaning, Derivation and Default
$\|C_i\|$	number of objects in class C_i
$\|\|C_i\|\|$	number of pages or blocks of class C_i
f_i	average number of references from an object in C_i to objects in C_{i+1} (fan-out)
r_i	average number of objects in class C_i referencing the same object in $C_{i+1}(=\frac{\|C_i\|*f_i}{\|C_{i+1}\|})$
$sz(OID)$	number of bytes for storing an object identifier (= 8)
$sz(m)$	number of bytes for the counter in a tuple of a join index (= 4)
$sz(ji)$	number of bytes of a tuple in a join index ($= 2 * sz(OID) + sz(m)$)
$sz(p)$	number of bytes of a page pointer (= 4)
B	number of bytes in a block or page of a disk (= 4096)
α	average page occupancy factor(= 70%)
BT_f	fan out of a B^+-tree ($\lceil = \frac{\alpha * B}{sz(p)+sz(OID)}\rceil$)
$fwd(i,j,k)$	average number of distinct objects in C_j referenced by a set of k objects in C_i
$bwd(i,j,k)$	average number of distinct objects in C_i referencing a set of k objects in C_j
$\|JI(i,j)\|$	number of tuples in $JI(i,j)$
$\|\|JI(i,j)\|\|$	number of blocks or pages of $JI(i,j)$

Since all the join indices are materialized in a complete join index hierarchy (CJIH), Step 1 of Algorithm 3.2 does not need to be performed in the construction of CJIH: All of the join indices at each level are considered as target join indices. The retrieval could be faster using a complete JIH in comparison with that using a corresponding partial JIH if the retrieval requires to access a (virtual) node which is not directly materialized in the partial JIH. However, a complete JIH obviously takes more storage space and more update propagation cost than a partial JIH although the update algorithm is similar to Algorithm 3.3.

4 Performance Evaluation of Join Index Hierarchies

An analytical model is constructed to study the performance of different join index hierarchies and access support relation [6], a competitive index structure for navigation through a sequence of object classes. The study is focused on several crucial performance measurements, including the storage size of a join index hierarchy, the cost of navigation (query processing), and the cost of update propagation over a join index hierarchy. Table 1 lists some database parameters used in the cost analysis.

4.1 Storage, navigation and update costs

The number of pages for a join index $JI(i,j)$ is

$$||JI(i,j)|| = \lceil \frac{sz(ji) * |JI(i,j)|}{B * \alpha} \rceil.$$

Following Valduriez[16], the number of disk accesses for a forward navigation from a set of n_i objects in C_i to objects in C_j using a target join index is

Table 2: Database Parameters

Parameters	C_0	C_1	C_2	C_3	C_4	C_5
$\|C_i\|$	1000	2000	4000	3000	1000	5000
f_i	1.0s	2.0s	1.0s	1.0s	1.0s	3.0s
s	$s = 0.1, 0.5, 1, 1.5, 2.0, 2.5, 3$					

$$1 + y(n_i, \lceil \frac{|C_i|}{BT_f} \rceil, |C_i|) + y(n_i, ||JI(i,j)||, |C_i|),$$

where y is a function from Yao[18],

$$y(k,m,n) = \lceil m * (1 - \prod_{i=1}^{k} \frac{n - \frac{n}{m} - i + 1}{n - i + 1}) \rceil.$$

It represents the number of page accesses for retrieving k objects out of n objects distributed over m pages. Here it is assumed that a typical B^+-tree is of two levels [1]. One page access is needed to retrieve the root node. To find the page pointers for n_i object identifiers, $y(n_i, \lceil \frac{|C_i|}{BT_f} \rceil, |C_i|)$ leaf pages of the B^+-tree are accessed. There are $y(n_i, ||JI(i,j)||, |C_i|)$ pages need to be accessed to find the tuples corresponding to n_i object identifiers. Thus the number of disk accesses for a forward navigation from a set of n_i objects in C_i to objects in C_j using a base join index hierarchy structure is

$$(1 + y(n_i, \lceil \frac{|C_i|}{BT_f} \rceil, |C_i|) + y(n_i, ||JI(i,i+1)||, |C_i|))$$

[1] The results for a B^+-tree of more than two levels can be calculated similarly as in Valduriez[16].

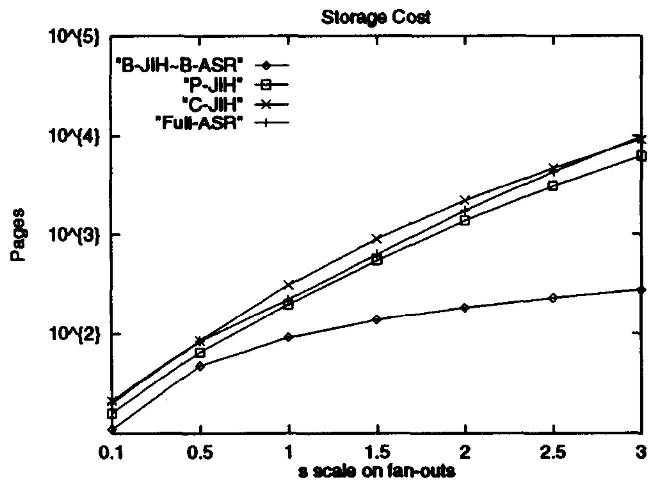


Figure 5: Storage Costs of B-JIH, P-JIH, C-JIH and Full-ASR vs. Fan-outs

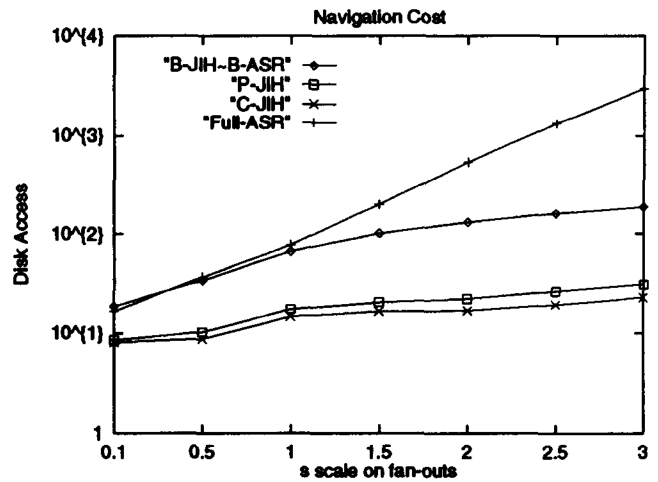


Figure 6: Navigation Costs of B-JIH, P-JIH, C-JIH and Full-ASR vs. Fan-outs

$$+\sum_{k=i+1}^{j-1}(1+y(fwd(i,k,n_i),\lceil\frac{|C_k|}{BT_f}\rceil,|C_k|)$$
$$+y(fwd(i,k,n_i),||JI(k,k+1)||,|C_k|)).$$

The first sum is the number of page accesses when the join index $JI(i,i+1)$ is scanned and related tuples retrieved. The second sum covers the case when $fwd(i,k,n_i)$ object identifiers from the previous join index $JI(k-1,k)$ are used to search the join index $JI(k,k+1)$. One page access is needed to retrieve the root node, and $y(fwd(i,k,n_i),\lceil\frac{|C_i|}{BT_f}\rceil,|C_i|)$ leaf pages of the B^+-tree are accessed to find the page pointers for the $fwd(i,k,n_i)$ object identifiers. Finally, there are $y(fwd(i,k,n_i),||JI(k,k+1)||,|C_k|)$ pages need to be accessed to find the tuples corresponding to the $fwd(i,k,n_i)$ object identifiers.

The update cost is computed similarly.

4.2 Explanation of performance results

Four data structures are compared in our performance study: (1) C-JIH as shown in Figure 2(a); (2) B-JIH as shown in Figure 2(b); (3) P-JIH as shown in Figure 2(c); and (4) Full-ASR (full *access support relation*), which stores the full sequences of object identifiers of the path (of length 5) in one full *access support relation*. Notice that cases (2) and (4) correspond to two extreme cases of the *access support relation* method proposed in [6], in which the former (case 2) decomposes each class pair into one component (i.e., binary decomposition of a full ASR: thus, a B-JIH is labeled B-JIH/B-ASR in the performance curves.), whereas the latter (case 4) merges the access path (sequence) into one relation.

The fan-out factors (join selectivities) is taken as the x-axis variable in Figures 5, 6, 7, and 10 because the performance is sensitive to the increase of the fan-out factors (join selectivities), which matches our expectation and experimentation. The set of class sizes, fan-out values, and scale changes in the analysis are in Table 2. The scale change factor s is introduced so that the performance under varying fan-outs can be presented in one graph. Other database parameters are set to the default values as shown in Table 1.

Figure 5 shows that the storage costs increase as the fan-outs do. Full-ASR stores all the sequences of object identifiers in complete or incomplete paths. P-JIH materializes the root node $JI(0,5)$ of the join index hierarchies and some higher level join indices; whereas C-JIH materializes all of the higher level join indices. These are reflected in the storage cost graph. Obviously, the storage sizes of Full-ASR, P-JIH and C-JIH increase faster than that of B-JIH/B-ASR.

Figure 6 presents how the navigation costs increase as the fan-outs grow. It is assumed that the forward and backward counts 50% and 50% in the total cost of the navigation respectively. The navigations between C_0 and C_5, C_1 and C_5, C_0 and C_3, and C_1 and C_4 weigh 50%, 20%, 20% and 10% in the total cost respectively. Notice that the navigation between C_1 and C_4 is not supported directly in the chosen P-JIH. The selectivity of navigation starting point is fixed as follows. If the navigation starts at C_i, the selectivity is chosen to be $sel * \frac{C_0}{C_i}$ where sel is the selectivity of the navigation starting at C_0. Here sel is set at 0.01, therefore, every navigation starts with 10 objects. P-JIH and C-JIH perform much better than B-JIH/B-ASR and Full-ASR. Full-ASR has the poorest performance because the whole ASR has to be retrieved (the relation is usually sorted on both head and tail classes to facilitate retrieval from the starting and the end points) when the navigations other than the one between head

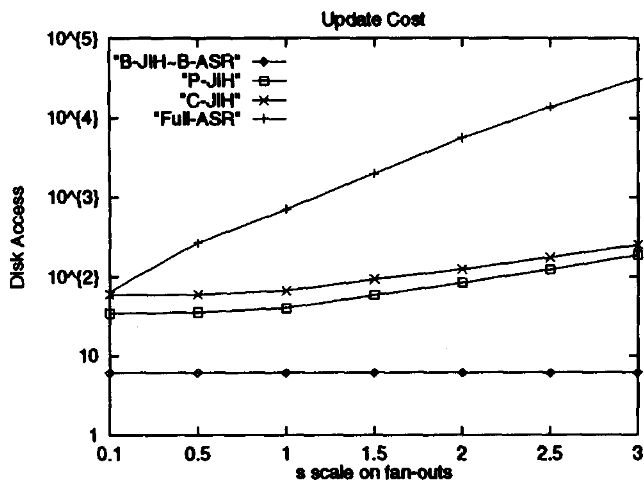


Figure 7: Update Costs of B-JIH, P-JIH, C-JIH and Full-ASR vs. Fan-outs

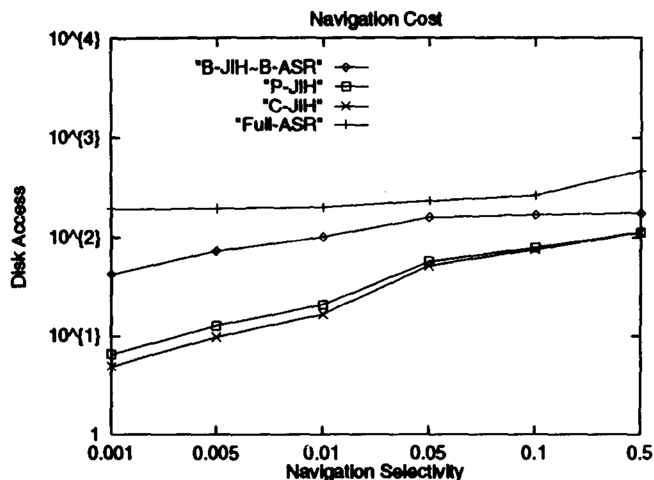


Figure 9: Navigation Costs of B-JIH, P-JIH, C-JIH and Full-ASR vs. Navigation Selectivities

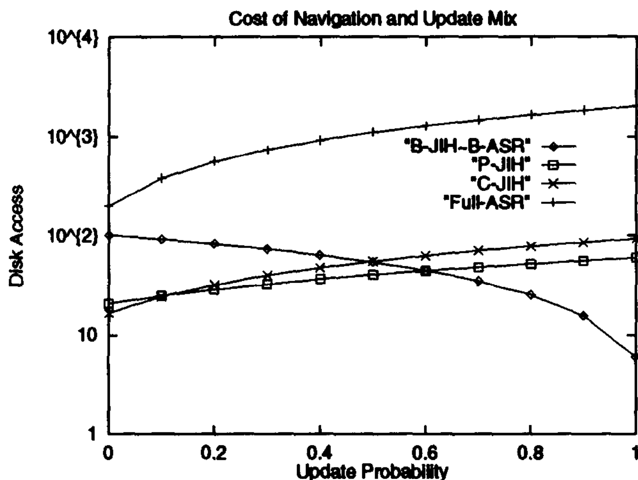


Figure 8: Costs of Navigation and Update mix for B-JIH, P-JIH, C-JIH and Full-ASR

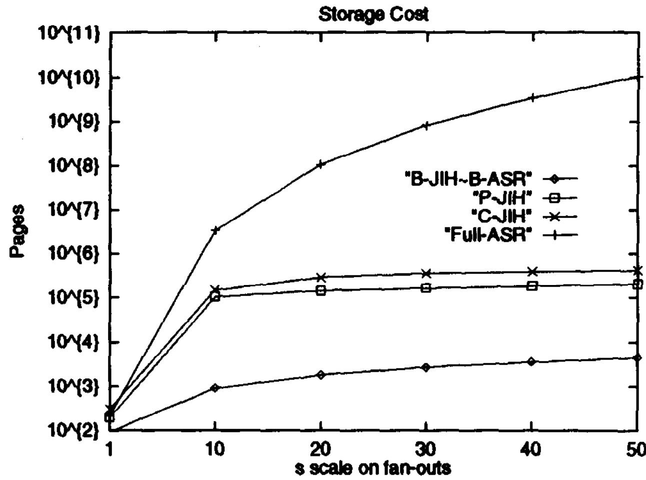


Figure 10: Storage Explosion with Large Fan-outs

and tail classes are required.

Figure 7 illustrates the update costs. It is assumed that the update probability of all the base join indices are equal. Obviously, B-JIH/B-ASR has the lowest update overhead since each time only base join indices need to be updated. The update cost of Full-ASR is higher than those of other index structures and grows faster.

Figure 8 describes the cost of navigation and update operation mix. The total cost is defined as $(1-p)*$ *NavigationCost*$+p*$*UpdateCost*, where p is the update probability, and $p = 0.2$ means that there are 20% probability of updates and 80% probability of navigations among all the operations. The scale s on fan-out is set to be 1.5. With less frequent update (update probability less than 0.5), the overall performance of P-JIH and C-JIH is much better than that of B-JIH/B-ASR. All the three structures perform better than Full-ASR.

Figure 9 presents the navigation costs vs. navigation selectivities. The scale s on fan-outs is set to be 1.5. The selectivity at C_0 is set from 0.001 to 0.5. The navigation cost grows as the navigation selectivity increases.

Figure 10 presents the storage requirements vs. large fan-outs. The reason that only large fan-outs are analyzed but not large cardinalities of classes is because our other performance results[2] shows that the costs of storage, navigation and updates do not grow very fast as the cardinalities of classes increase. As one can predict, the storage cost (and hence the navigation and update costs) grows rapidly when the fan-out ratio grows. Full-ASR has the highest storage cost since multiple access paths from C_{i-1} to C_i will have to be multiplexed when pairing with the objects in C_{i+1}, etc.

[2]not shown here due to space limitation.

This also suggests that the fan-outs should be considered as an important factor for setting "fire walls" to avoid cost explosion.

In summary, the performance study shows that both P-JIH and C-JIH outperform B-JIH/B-ASR and Full-ASR in navigation and overall performance. P-JIH has better storage and better update costs than C-JIH. Clearly, join index hierarchy, especially the partial one, provides an interesting data structure to support efficient navigations in object-oriented databases.

5 Discussion

5.1 Join index hierarchy which supports other kinds of navigations

The join index hierarchies discussed in the previous sections are designed for support of class composition hierarchies, i.e., navigations through a sequence of object classes via their attribute relationships. Similar join index hierarchies can be applied to support of navigations through class/subclass hierarchies, or through a sequence of classes via the relationships specified by methods and/or deduction rules.

Some relationships between different classes of objects may not be specified by existing attributes but by deduction rules or computational methods. For example, the relationships between the objects in two classes, *Parks* and *Lakes*, could be specified by a spatial computational routine, which computes, based on a geographic map, whether one is inside the other, or whether two intersect, or their shortest (or highway) distances, otherwise.

The method- or deduction rule- specified object linkage can be constructed using the structure of join index hierarchy as well, by evaluation of the method/rule at the join index construction time rather than at the query processing time.

One advantage of the construction of join indices for rule- or method- defined object linkages could be the transformation of the expensive rule/method computation from query evaluation time to join index construction time. Since a method or a rule may involve recursion or iterative computation of a relatively large number of complex (such as spatial) objects, it could be quite expensive to perform such computation at the query processing time. The evaluation of such linkages at the join index construction time and the storage of the join indices together with other frequently used information (such as distance, etc. [9]) in join indices will trade storage space for query evaluation efficiency. It will be especially beneficial if such computation must be performed repeatedly or iteratively.

Furthermore, by storage of important information in join indices, some queries, especially those involving traversing in the direction in reverse to those specified in the methods or rules, can be answered efficiently. For example, to find all the lake and park pairs whose intersected regions greater than 1 square kilometer, one can retrieve the join indices and return the results directly (if the information-associated join indices [9] are constructed and the area of intersection is the associated information). However, it is impossible to compute a region from an area based on the *same* method which defines only the computation of an area from a geographic object but not in reverse.

5.2 "Fire walls" in the construction of join index hierarchies

There may exist long object referencing sequences in queries, and any object class may serve as the starting point in a sequence of object referencing. Nevertheless, this does not suggest the construction of join index hierarchies on a very long sequence of schema path because of the size of such a hierarchy and the cost of updates. Therefore, it is often necessary to partition a long schema path into a few short ones, or prohibitive to build some join indices or merge them into join index hierarchies.

A class linkage (by either attribute relationship, methods, or rules) which is not suitable for constructing join indices or for being merged into a join index hierarchy is called the "fire wall" of the hierarchy. It is important to identify fire walls and partition a long schema path into a set of smaller ones for the construction of easily accessible or updatable join index hierarchies.

"Fire walls" are suggested to set in the following places in the design of a join index hierarchy.

1. *Rarely referenced class linkages*: Some class linkages, though referable, are rarely used in applications, based on the examination of a relatively long history of referencing patterns. It is relatively safe to set up a fire wall at a rarely referenced point since it is fair to let rarely used referencing pay a little higher cost in accessing.

2. *Large join selectivities*: A large join selectivity implies a potentially large (or huge) join index relation. The further construction of upper level join indices would usually result in large join index relations as well. The break of the chain at this point may contribute to a relatively small join index relation and/or hierarchy.

3. *Frequently updated or multiple-source class linkages*: Some join index may sustain frequent updates or be derived from multiple objects, classes

or class relationships (such as, those computed using multiple objects or classes by methods). Such kind of class linkages may need frequent or sophisticate updates and update propagation to upper level join indices will likely be costly and thus it could be beneficial to set up "fire walls" there.

6 Conclusions

A join index hierarchy approach has been proposed and investigated here for efficient navigation through a sequence of object classes in object-oriented databases. The join index hierarchy organizes a set of (direct and indirect) join index nodes into a hierarchy. Three kinds of join index hierarchies are proposed and studied. Our analysis and performance study show that partial join index hierarchy has reasonably small space and update overheads and speeds up query processing considerably in both forward and backward navigations.

Join index hierarchy is an interesting indexing structure which could be a promising candidate at solving "pointer chasing" problems in object-oriented database query processing. More experiments need to be conducted in the performance study of join index hierarchies. Furthermore, it is interesting to compare and/or integrate the join index hierarchy method with other object query optimization techniques, such as read-ahead buffering [11] and complex object assembly [5].

References

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

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

[3] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. ACM-SIGMOD Conf. Management of Data*, 383–392, 1992.

[4] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Survey*, 25(2):73–170, June 1993.

[5] T. Keller, G. Graefe, and D. Maier. Efficient assembly of complex objects. In *Proc. ACM-SIGMOD Conf. Management of Data*, 148–157, Denver, CO, May 1991.

[6] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. ACM-SIGMOD Conf. Management of Data*, 364–374, Atlantic City, NJ, May 1990.

[7] A. Kemper and G Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. Int. Conf. Very Large Database*, 290–301, Brisbane, Australia, August 1990.

[8] K. C. Kim, W. Kim, and A. Dale. Indexing techniques for object-oriented databases. In W. Kim and F. H. Lochovsky, editors, *Object-oriented concepts, Databases, and Applications*, 371–394. Addison-Wesley, 1989.

[9] W. Lu and J. Han. Distance-associated join indices for spatial range search. In *Proc. 8th Int. Conf. Data Engineering*, 284–292, Phoenix, AZ, Feb. 1992.

[10] D. Maier and J. Stein. Indexing in an object-oriented DBMS. In *Proc. IEEE Int. Workshop on Object-oriented Database System*, 171–182, Asilomar, Pacific Grove, CA, September 1986.

[11] M. Palmer and S. B. Zdonik. FIDO: a cach that learns to fetch. In *Proc. Int. Conf. Very Large Database*, Barcelona, Spain, 1991.

[12] D. Rotem. Spatial join indices. In *Proc. 7th Int. Conf. Data Engineering*, 500–509, Kobe, Japan, April 1991.

[13] G. M. Shaw and S. B. Zdonik. A query algebra for object-oriented databases. In *Proc. Int. Conf. Data Engineering*, 154–165, Los Angeles, CA, February 1990.

[14] E. J. Shekita and M. J. Carey. Performance enhancement through replication in an object-oriented DBMS. In *Proc. ACM-SIGMOD Conf. Management of Data*, 325–336, 1989.

[15] D. D. Straube and M. T. Ozsu. Queries and query processing in object-oriented database systems. *ACM Trans. Office and Information Systems*, 6(4):387–430, Oct 1990.

[16] P. Valduriez. Join indices. *ACM Trans. Database Systems*, 12(2):218–246, 1987.

[17] S. L. Vandenberg and D. J. DeWitt. Algebraic support for complex objects with arrays, identity, and inheritances. In *Proc. ACM-SIGMOD Conf. Management of Data*, 158–167, Denver, CO, May 1991.

[18] S. B. Yao. Approximating block accesses in database organizations. *Communications of the ACM*, 20(4):260–261, April 1977.