

Join Index Hierarchy: An Indexing Structure for Efficient Navigation in Object-Oriented Databases *

Jiawei Han Zhaohui Xie Yongjian Fu

Abstract

A novel indexing structure, join index hierarchy, 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.

Index Terms: index structures, join indices, query processing, query optimization, object-oriented database.

1 Introduction

Query processing and optimization is crucial to the performance of object-oriented database systems. Substantial research into query processing and query optimization in object-oriented databases has been conducted in recent years with encouraging progress reported, e.g., [1, 3, 4, 5, 6, 8, 10, 14, 15, 16, 19, 20, 24, 25, 27, 29, 31, 33]. An important direction that has been pursued in previous studies is the extensions of various kinds of query optimization techniques developed in relational systems towards query processing in object-oriented databases. However, since object-oriented database systems support many object-oriented features including abstract data types, methods, encapsulation, inheritance, object identity and complex object structures, new techniques must be developed to meet the challenges of supporting these features.

Since object-oriented databases support complex data objects and enable explicit and natural representation of logical relationships among complex objects via class/subclass hierarchies, attributes¹, methods, object identities, etc., navigation among different classes and objects via class hierarchies and/or class composition hierarchies is an essential operation.

Example 1.1 The schema of a simple object-oriented database is given in Figure 1, in which several classes are connected via the relationships induced by attributes, methods and sub/superclasses. A sample database is shown in Figure 2 with instances of each class in the schema in Figure 1. To find out the departments of the professors who teach courses taken by *Joe Jones*, navigation is performed from the object *1* in the class *STUDENT* to objects {*5*, *7*} in the class *COURSE* via the attribute *TakeCourses* of *STUDENT*, to object *10* in *PROF* via the attribute *Instructor* of *COURSE*, and finally to object *12* in *DEPT* via the attribute *Dept* of *PROF*. □

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 [9]. The attempts to solve this problem can be classified into three classes of techniques: the indexing method (e.g., [15, 12]), the read-ahead buffering method (e.g., [26]), and complex object assembly method (e.g., [14]).

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 [32] and its variations in spatial databases [28, 22], by constructing 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 represented by *indirect* logical relationships are called derived join indices. A join index hierarchy supports navigations through a sequence of classes in either

*The research was partially supported by the Natural Sciences and Engineering Research Council of Canada under the Grant OGP03723, the Institute for Robotics and Intelligent Systems under the Grants HMI-5 and IC-2, and the Centre for Systems Science of Simon Fraser University. This article is a substantially extended and modified version of a conference paper, “*Join Index Hierarchies for Supporting Efficient Navigations in Object-Oriented Databases*”, by Z. Xie and J. Han, in the Proceedings of the 20th Int. Conf. on Very Large Data Bases, pp. 522-533, Santiago, Chile, Sept., 1994. Part of the work of the second author was done while studying in Simon Fraser University.

¹An attribute could take a single value, a set of values or multi-set of values.

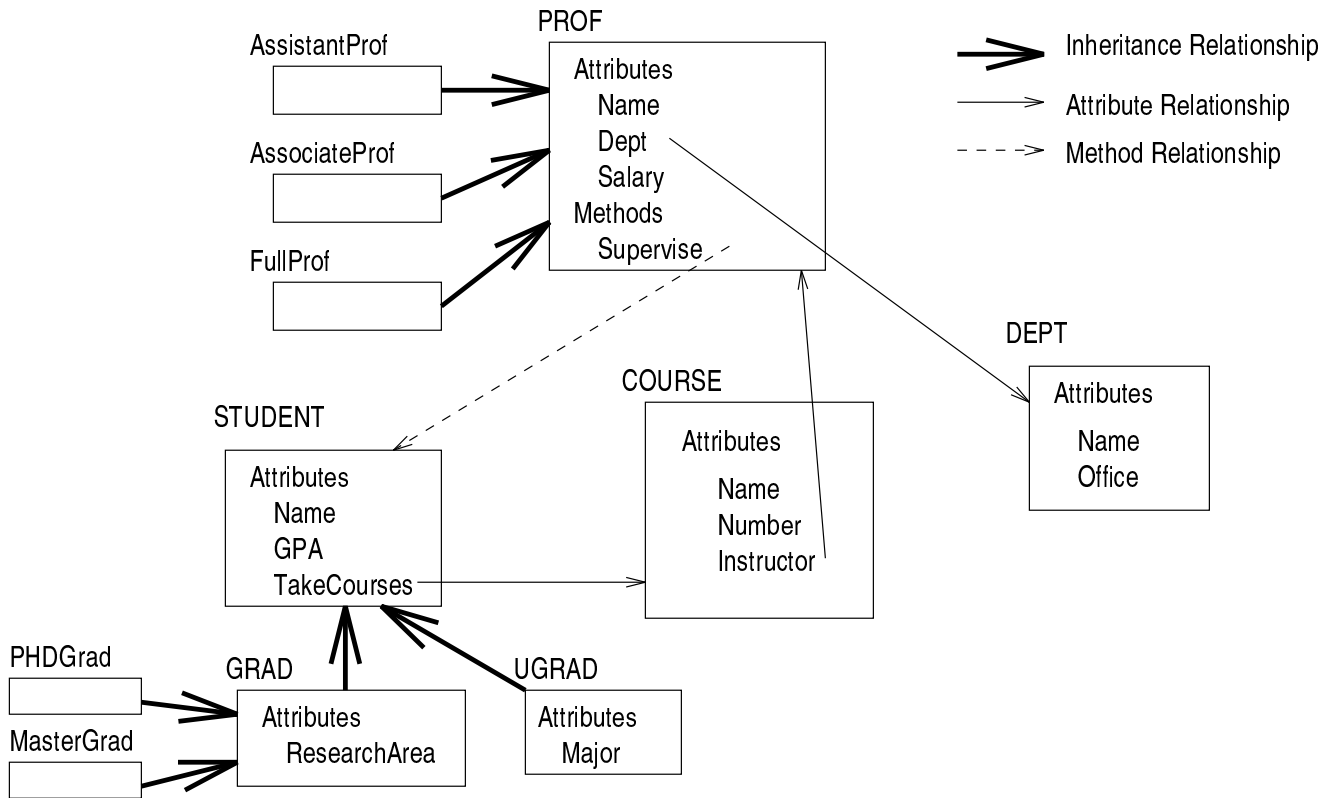


Figure 1: Pointer Chasing via Class Composition/Inheritance Hierarchies.

STUDENT	COURSE	PROF	DEPT
OID: 1 Name: Joe Jones GPA: 3.22 TakeCourses: {5, 7}	OID: 4 Name: Fortran Number: 51 Instructor: 9	OID: 8 Name: Carl Hines Dept: 11 Salary: 34000	OID: 11 Name: CS Office: 216 CS Building
OID: 2 Name: May Wilson GPA: 2.78 TakeCourses: {4,6,7}	OID: 5 Name: Algebra Number: 12 Instructor: 10	OID: 9 Name: Lou Cox Dept: 11 Salary: 25000	OID: 12 Name: Math Office: 110 Fulton Hall
OID: 3 Name: Steve Smith GPA: 3.10 TakeCourses: {4}	OID: 6 Name: Compiler Number: 234 Instructor: 8	OID: 10 Name: Kim Hunt Dept: 12 Salary: 42000	
	OID: 7 Name: Calculus Number: 101 Instructor: 10		

Figure 2: A Sample Object Base.

forward or backward navigation direction and supports efficient update propagation starting with the base join indices 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, and update overheads on join index hierarchies.

Second, 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 hierarchies. 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.

Third, 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.

Fourth, 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 remainder 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

Haerder [11] proposed links to optimize joins. The links are implemented by chaining tuples using tuple identifiers, an implementation similar to that of CODASYL systems. The join index structure was proposed by Valduriez [32] 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 access 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 [32].

Join index structures can be applied to different application domains. For example, a spatial join index structure was developed by Rotem [28] 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 [22].

Two kinds of index structures have been studied in object-oriented databases². The first one aims at associative search. The typical examples are the 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 [23], Bertino and Kim [4], and Bertino and Foscoli [3]. The second kind of index structures, which support navigations, includes the access support relations proposed by Kemper and Moerkotte [15], and the object skeleton by Hua and Tripathy [12].

In Maier and Stein [23], 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 [4], 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 [3, 2]. 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

²Since we concentrate on the index structures for efficient navigations, the index structures for class hierarchies, e.g., Multikey index[24], χ -tree[6], CH-tree [18], H-tree [21] and CG-tree [17], will not be discussed in this paper.

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 [3, 2]. Choenni et. al. [7] propose an optimal index configuration by splitting a long path expression into shorter ones, and by indexing the shorter paths with existing index structures such as those in [4, 2]. Shekita and Carey [30] describe a mechanism called field replications which replicate the values of the nested attributes. 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 [23], are used to support update propagation. Kato and Masuda [13] present a mechanism called persistent caching which is similar to the field replication [30]. In this approach, the referenced objects are cached into the referencing objects. Update is delayed until the cached objects are required. A hash table stored in the main memory is employed to maintain the cached values consistent with the original objects. 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 [15] 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 similarities 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. Hua and Tripathy [12] propose a navigation structure called *object skeleton* which essentially is a network of object identifiers. The two object identifiers are connected if the corresponding objects are associated by, for example, attribute relationship. The approach is more general in the sense that the navigations can be supported between two classes not only in a path expression but also over a network of classes. The navigations, however, are supported efficiently only if the starting points of the navigations can be located by using some nested indexes such as those in [4, 2]. Besides, an update is required to be propagated over the network of object identifiers and the nested indexes.

2.2 Preliminaries

Following the previous research, a join index hierarchy structure is proposed here to support efficient navigation through multiple object classes. For example, in Figure 1, one may like to find which departments offer courses taken by Jones' students, or which courses the undergraduate student "John" is taking, which departments offer the courses taken by a PhD student "Mary", etc. These queries correspond to navigations through a set of classes, such as *AssistantProf*, *DEPT*, *UGRAD*, *COURSE*, etc. via appropriate relationships.

The variations of a join index hierarchy can be constructed based on the richness of the derived join index structures. Three kinds of structures: *base-only*, *complete*, and *partial*, are investigated in terms of their construction, navigation and update propagation.

For the clarity of presentation, only the relationships between the *existing* attributes among object classes are considered in the construction and maintenance of join index hierarchies. Join index hierarchy which handles the relationships *induced* by attributes, methods, rules, and class/subclass hierarchies will be discussed in Section 5.

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 $i = 0, 1, \dots, n - 1$, there is a directed edge from C_i to C_{i+1} , labeled with A_{i+1} , in a database schema, then $\langle C_0, A_1, C_1, A_2, \dots, A_n, C_n \rangle$ is a schema path.

Given a schema path $\langle C_0, A_1, C_1, A_2, \dots, A_n, C_n \rangle$ over a database schema, a join index file $JI(i, j)$ ($1 \leq i < j \leq n$) 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 at least one object path $\langle o_i, o_{i+1}, \dots, o_{j-1}, o_j \rangle$ such that for $k = 0, 1, \dots, j - i - 1$, 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 (possibly) different object classes along a schema path form a join index hierarchy, denoted as $JIH(C_0, A_1, C_1, A_2, \dots, A_n, C_n)$, 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 $0 < l < j - i$. Such child-parent relationship represents direct update dependency between the child node and the parent node, i.e., whenever the child node is updated, the parent should be updated as well. The join index nodes $JI(i, i + 1)$, for $i = 0, 1, \dots, n - 1$, are at the bottom of the hierarchy, and are therefore, called base join indices.

Example 2.1 A path in the schema shown in Figure 1 is given in Figure 3. The classes and the attributes in the path $\langle STUDENT, TakeCourses, COURSE, Instructor, PROF, Dept, DEPT \rangle$ are displayed. \square

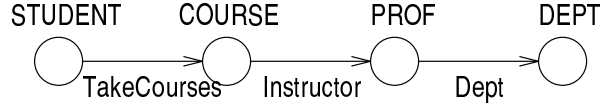


Figure 3: A Schema Path.

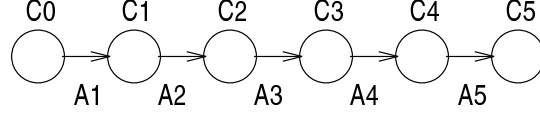


Figure 4: A Schema Path of Length 5.

Figure 4 shows a schema path of length 5 on a class composition hierarchy and Figure 5 illustrates the following three join index hierarchy structures.

1. A **complete join index hierarchy (C-JIH)**, as shown in Figure 5(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 5(b), only consists of all 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, as denoted by the dotted links.
3. A **partial join index hierarchy (P-JIH)**, as shown in Figure 5(c), consists of a proper subset of the set of base and 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. A directed link from a child node to a parent node denotes that the parent node is derived from the child node, which implies update dependency.

Example 2.2 Based on the objects shown in Figure 2, a complete join index hierarchy for the schema path in Figure 3 can be constructed. The hierarchy is shown in Figure 6 where s , c , p , d represents objects in $STUDENT$, $COURSE$, $PROF$, $DEPT$, respectively. \square

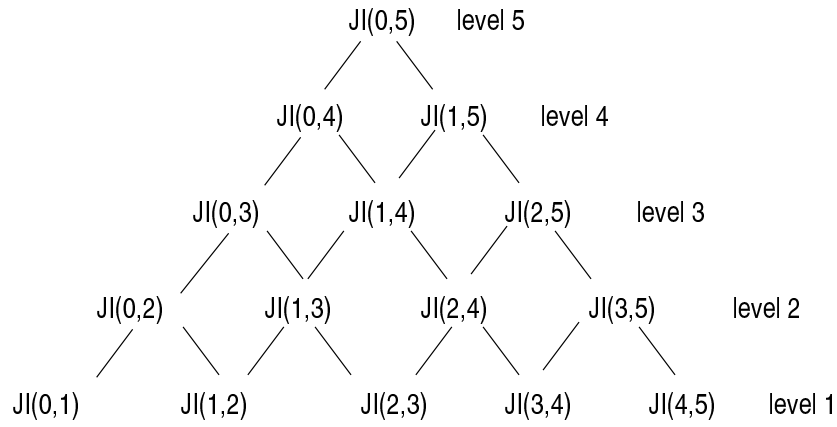
Example 2.3 Figure 5(c) demonstrates a typical partial join index hierarchy which supports direct navigations between C_0 and C_4 , and C_2 and C_5 . Their corresponding JI nodes: $JI(0, 4)$ and $JI(2, 5)$, circled in the figure, are called *target nodes*. Notice that a materialized intermediate level 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 $JI(0, 4)$.

Suppose there were no intermediate level join index nodes in the hierarchy $JIH(0, 5)$, an average of 3.2 join-like (defined later) operations are needed to propagate an update from the base join indices to the target nodes $JI(0, 4)$ and $JI(2, 5)$. Here it is assumed that the probability of an update on each base join index is the same. If there is an update on $JI(0, 1)$ or $JI(1, 2)$, $JI(0, 4)$ needs 3 join-like operations for update propagation and $JI(2, 5)$ does not need to be modified. If there is an update on $JI(4, 5)$, $JI(2, 5)$ needs 2 join-like operations for update propagation and $JI(0, 4)$ does not need to be modified. If there is an update on $JI(2, 3)$ or $JI(3, 4)$, both $JI(0, 4)$ and $JI(2, 5)$ need to be updated with 4 join-like operations.

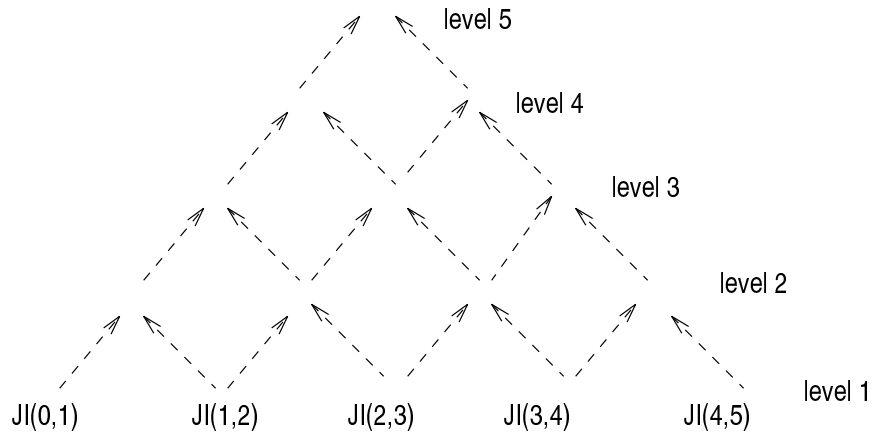
With the help of intermediate level join indices, it takes an average of 2.2 join-like operations to propagate an update from the base join indices. If there is an update on $JI(0, 1)$ or $JI(1, 2)$, 2 join-like operations are needed to update $JI(0, 2)$ and $JI(0, 4)$. Other join indices do not need to be updated. If there is an update on $JI(4, 5)$, one join-like operation is required to update $JI(2, 5)$ while other join indices do not need to be updated. If there is an update on $JI(2, 3)$ or $JI(3, 4)$, $JI(2, 4)$, $JI(0, 4)$ and $JI(2, 5)$ are required to be modified with 3 join-like operations. \square

In a join index hierarchy $JIH(0, n)$, the base join index nodes $JI(i, i + 1)$ (for $i = 0, \dots, n - 1$) 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.

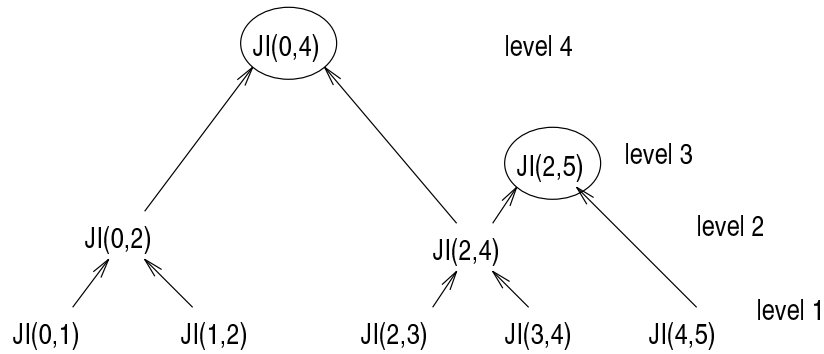
In a join index hierarchy, a set of join index nodes which must be supported (due to frequent references) are called *target join indices*, e.g. $JI(0, 4)$ and $JI(2, 5)$ in Figure 5(a); whereas the others which are mainly used for update propagation are



(a) A Complete Join Index Hierarchy



(b) A Base Join Index Hierarchy



(c) A Partial Join Index Hierarchy

Figure 5: Three Kinds of Join Index Hierarchies Corresponding to the Schema Path in Figure 4.

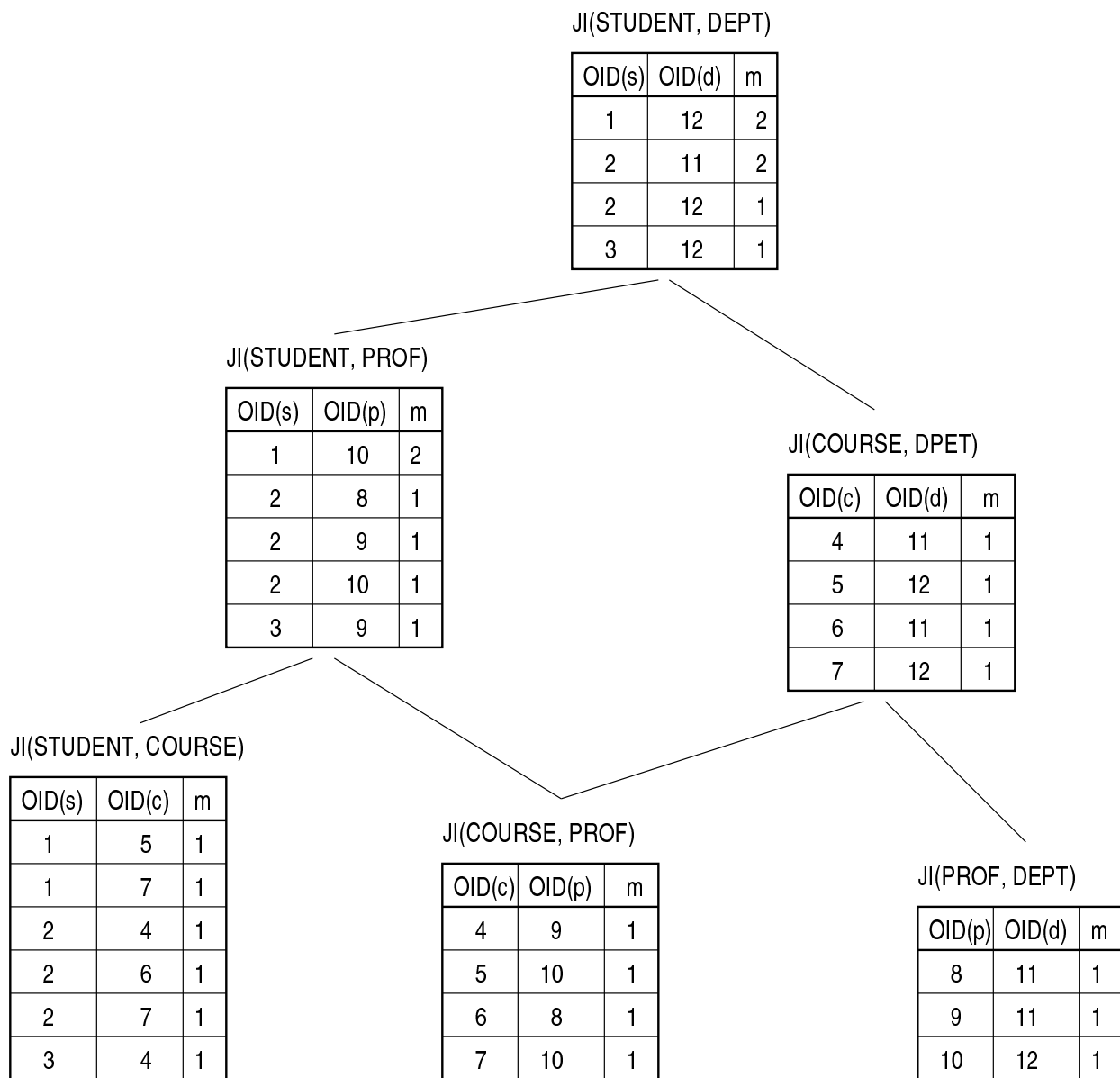


Figure 6: A Complete Join Index Hierarchies for the Schema Path in Figure 3.

called auxiliary join indices, e.g. $JI(0, 2)$, and $JI(2, 4)$. 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 that include the updates on base join indices.

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 $(OID(o_i), OID(o_j), m)$ where $m > 0$, 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 $(OID(o_i), OID(o_j), -m)$ where $m > 0$, indicating that there are m object paths connecting o_i and o_j being deleted. The $-m$ representation is chosen because it allows us to treat insertion and deletion equally as explained below. $\delta JI(i, j)$ denotes either $\nabla JI(i, j)$ or $\Delta JI(i, j)$.

The propagation of update is carried out by the join operator “ \bowtie_c ”, which is similar to a join operation in relational databases. $JI(i, k) \bowtie_c JI(k, j)$ 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 $JI(k, j)$. An update on a lower level join index, $JI(C_i, C_k)$, is propagated to a higher level join index, say $JI(C_i, C_j)$, $j > k$, by adding $\delta JI(i, k) \bowtie_c JI(k, j)$ to $JI(i, j)$. Notice that identical tuples, such as $(OID(o_i), OID(o_j), m_k)$ (for $k = 0, 1, \dots, p$) are automatically merged into one with their path numbers accumulated, i.e., $(OID(o_i), OID(o_j), \sum_{k=0}^p m_k)$. In the case of deletion, if the counters are zero, the corresponding tuples should be deleted.

Another operator “ \bigcup_c ” is introduced in the update algorithm. $JI(i, j) \bigcup_c \Delta JI(i, j)$ indicates an insertion into $JI(i, j)$. If there exists a path in $JI(i, j)$ for the corresponding objects, the number of paths connecting o_i and o_j will increase. Similarly, $JI(i, j) \bigcup_c \nabla JI(i, j)$ indicates a deletion from $JI(i, j)$, and the number of paths connecting the corresponding objects o_i and o_j will decrease. Again, if the counters are zero, the corresponding tuples should be deleted.

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 5(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. □

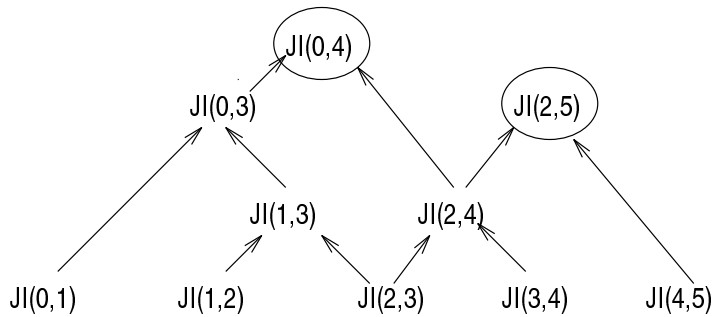
Example 3.2 To directly support the navigations between C_0 and C_4 , and C_2 and C_5 , the set of target join indices are $\{JI(0, 4), JI(2, 5)\}$, and the set of base join indices are

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

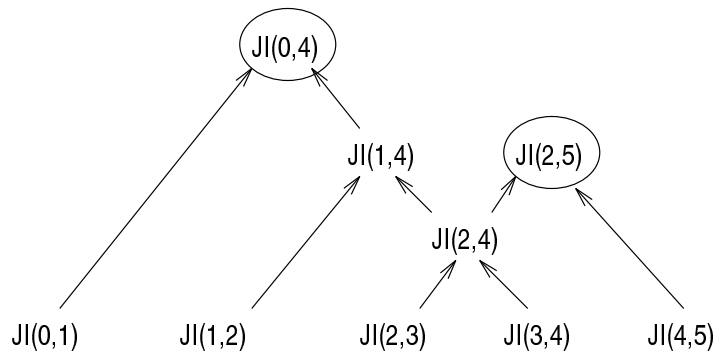
Three different kinds of partial join index hierarchies are presented in Figure 7(a)(b) and Figure 8.

The sets of auxiliary JIs which supports the two target JIs are $\{JI(0, 3), JI(1, 3), JI(2, 4)\}$ in Figure 7(a), $\{JI(1, 4), JI(2, 4)\}$ in Figure 7(b) and $\{JI(0, 2), JI(2, 4)\}$ in Figure 8. □

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



(a) A Partial Join Index Hierarchy with average # of operations for update=2.8



(b) A Partial Join Index Hierarchy with average # of operations for update=2.4

Figure 7: Two Partial Join Index Hierarchy Structures for Supporting $JI(0,4)$ and $JI(2,5)$.

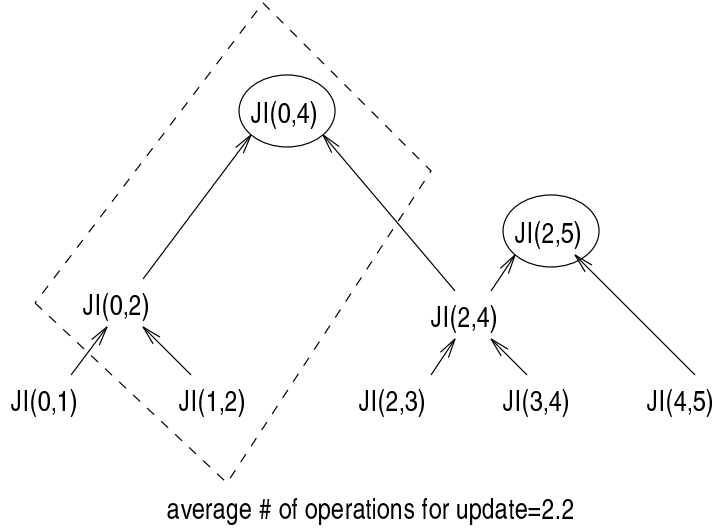


Figure 8: Build a Partial Join Index Hierarchy and Propagate Update.

Algorithm 3.1 Construction of a minimum auxiliary set of JIs

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

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. Initialize S to be a set with a single element which is the set of all target nodes. S is recursively expended by adding immediate auxiliary nodes.

The set of immediate auxiliary nodes for a (target or auxiliary) node $JI(i, j)$ is $\{\{JI(i, i+1), JI(i+1, j)\}, \dots, \{JI(i, j-1), JI(j-1, j)\}\}$ with the removal of $JI(i, k)$ or $JI(k, j)$, $i < k < j$, if it is a target node or a base node. If there is an empty set resulted from this removal, remove the empty set.

For each $JI(i, j)$ in a set s in S , find the set of its immediate auxiliary nodes, $\{a_1, \dots, a_l\}$. Make l copies of s , and add elements in a_m into the m -th copy, $m = 1, \dots, l$, which forms l new sets. Remove $JI(i, j)$ in all copies if it is a target node. Replace s in S with the 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 S , 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. \square

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

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

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

$$\begin{aligned} S = & \{\{JI(0, 2), JI(0, 3), JI(2, 4)\}, \{JI(1, 3), JI(0, 3), JI(2, 4)\}, \{JI(1, 3), JI(1, 4), JI(2, 4)\}, \\ & \{JI(1, 4), JI(2, 4)\}, \{JI(0, 2), JI(0, 3), JI(3, 5)\}, \{JI(1, 3), JI(0, 3), JI(3, 5)\}, \\ & \{JI(2, 4), JI(1, 4), JI(3, 5)\}, \{JI(0, 2), JI(2, 4)\}, \{JI(0, 2), JI(2, 4), JI(3, 5)\}\}. \end{aligned}$$

Both $\{JI(1, 4), JI(2, 4)\}$ and $\{JI(0, 2), JI(2, 4)\}$ have the minimum number of auxiliary join indices. The first one corresponds to the partial join index hierarchy structure in Figure 7(b), whereas the second one to that in Figure 8. The

average numbers of \bowtie_c operations for update propagation in Figure 7(b) and Figure 8 are 2.4 and 2.2 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. \square

A few remarks about Algorithm 3.1 should be noted.

- The worst time complexity of Algorithm 3.1 is exponential to n , the number of classes in the path. This is acceptable in many cases since n is usually quite small (e.g., no more than 7). When n is large, some heuristics should be exploited to reduce the computational cost. For example, in Example 3.3, the auxiliary sets $\{JI(0, 3), JI(2, 4)\}$ do not need to be expanded because another set $\{JI(0, 2), JI(2, 4)\}$ in S will always have less auxiliary nodes. A more efficient algorithm can be worked out with such heuristics incorporated.
- Algorithm 3.1 generates the optimal hierarchy in terms of total number of auxiliary nodes and updating cost. If the number of objects in each class is relative even and the fan-out factors (average number of references from an object in C_i to objects in C_{i+1}) are uniform, the algorithm finds the best join index hierarchy. When the number of objects in a class and the fan-out factor vary a lot from class to class, a more sophisticated algorithm, discussed in Section 5.3, should be employed to find the best hierarchy.
- When the join indices become too large because of large fan-out factor or long path, a “fire wall” should be set up to break up the schema path. More about “fire walls” is discussed in Section 5.2.

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 $C_0, A_1, C_1, A_2, \dots, A_n, C_n$ and the corresponding classes.

Output: $JIH(C_0, A_1, C_1, A_1, \dots, A_n, C_n)$, 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 $i = 0, 1, \dots, n - 1$ 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 . \square

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.

Given the root node $JI(0, n)$ as the single target node, the minimal number of required auxiliary node is $(n - 2)$. This can be proved by induction on n . The result holds true when $n = 2$. Let us assume that the result is true for $2, \dots, n - 1$. Obviously, the root node needs a pair of auxiliary nodes $JI(0, k)$ and $JI(k, n)$ where $0 < k < n$. Both these two nodes can be considered as the root nodes of the two *disjoint* join index hierarchies as in Figure 9. Therefore, the minimal number of auxiliary nodes is the sum of the minimal numbers of auxiliary nodes for these two join index hierarchies, i.e.,

$$(k - 2) + ((n - k) - 2) + 2 = n - 2.$$

Therefore, we have the following theorem.

Theorem 3.1 If the root node, $JI(0, n)$, is the only target node, $(n - 2)$ is the minimal number of auxiliary join indices that are required to support the navigation.

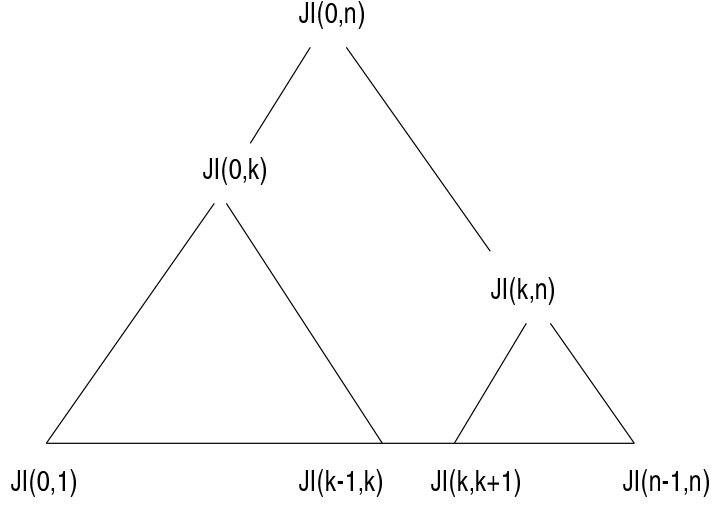


Figure 9: The Target Node $JI(0,n)$ needs a Pair of Auxiliary Nodes $JI(0,k)$ and $JI(k,n)$

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 8 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.

Corollary 3.1 If the path length is less than 7 and the root node is the only target node, a minimum set of auxiliary nodes exists such that the unmaterialized nodes can be computed by at most one join of two existing materialized nodes.

Obviously, if $n < 7$, given any set of target nodes, the minimal number of auxiliary node is not greater than $(n - 2)$. Therefore, we have the following conclusion.

Corollary 3.2 Given any set of target nodes, at most $(n - 2)$ auxiliary join indices are required to support a set of target navigations along a path with length $n < 7$.

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 8, 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.*

for level $l := 2$ to n do
 for $i := 0$ to $n - l$ do

Table 1: Database Parameters

Parameters	Meaning, Derivation and Default Values
$ 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 $JI(i, j)$

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

$\delta JI(i, i+l) := JI(i, i+q) \bowtie_c \delta JI(i+q, i+l)$,

where $1 \leq p < k-i$ and $k-i \leq q \leq l-1$;

$JI'(i, i+l) := JI(i, i+l) \cup_c \delta JI(i, i+l)$; \square

Notice that incremental updates are performed on both forward and backward join indices. Also, there could be more than one way to compute $\delta JI(i, i+l)$ in Step 2, and the choice can be determined by the cost analysis presented in the later section.

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:

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

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.

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, access support relation [15], a competitive index structure for navigation through a sequence of object classes, and nested index [4, 2] for associative search. 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. The details of the estimation of some of these parameters are in Appendix A.

4.1 Storage and navigation costs

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

$$||JI(i, j)|| = \lceil \frac{sz(ji) * |JI(i, j)|}{B * \alpha} \rceil. \quad (4.1)$$

Following Valduriez [32], the number of disk accesses ³ 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

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

where y is a function from Yao [34],

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

It represents the number of page accesses for retrieving k objects out of n objects distributed over m pages. 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

$$(1 + y(n_i, \lceil \frac{|C_i|}{BT_f} \rceil, |C_i|) + y(n_i, ||JI(i, i + 1)||, |C_i|)) + \sum_{k=i+1}^{j-1} (1 + y(fwd(i, k, n_i), \lceil \frac{|C_k|}{BT_f} \rceil, |C_k|) + y(fwd(i, k, n_i), ||JI(k, k + 1)||, |C_k|)). \quad (4.2)$$

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 $JI(k, k + 1)$.

4.2 Update cost

Assume that there is an update on an object in C_k which causes the update on $JI(k, k + 1)$, either deletion or insertion $\delta JI(k, k + 1)$. The cost of updating a partial join index hierarchy consists of three parts. The first part is the cost of updating $JI(k, k + 1)$ itself in Step 1 of Algorithm 3.3. The cost of updating forward $JI(k, k + 1)$ is

$$1 + y(|\delta JI(k, k + 1)|_k, \lceil \frac{|C_k|}{BT_f} \rceil, |C_k|) + 2 * y(|\delta JI(k, k + 1)|_k, ||JI(k, k + 1)||, |C_k|),$$

where $|\delta JI(i, j)|_i$ stands for the number of identifiers of distinct objects of C_i in the tuples of $\delta JI(i, j)$ and $|\delta JI(i, j)|_j$ stands for the number of identifiers of distinct objects of C_j in the tuples of $\delta JI(i, j)$. Here $|\delta JI(k, k + 1)|_k$ and $|\delta JI(k, k + 1)|_{k+1}$ are initialized, e.g., to 1 at the beginning. One page access is needed to retrieve the root node of the B^+ -tree of $JI(k, k + 1)$. The second sum covers the cost of retrieving the leaf pages of the B^+ tree for finding the page pointers. The third sum handles the cost of inserting or deleting the related tuples which includes reading and writing back the related pages. The cost of updating backward $JI(k, k + 1)$ is similar. The second part is the cost $UJI(i, i + l)$ for updating materialized join index $JI(i, i + l)$ at level l . According to step 2 of Algorithm 3.3, $|\delta JI(i, i + l)|_i$ and $|\delta JI(i, i + l)|_{i+l}$ can be calculated iteratively from $|\delta JI(k, k + 1)|_k$ and $|\delta JI(k, k + 1)|_{k+1}$. If the first expression in the step 2 of Algorithm 3.3 is chosen, then

$$|\delta JI(i, i + l)|_i = bwd(i, i + p, |\delta JI(i + p, j + l)|_{i+p}), \text{ and} \\ |\delta JI(i, i + l)|_{i+l} = |\delta JI(i + p, j + l)|_{j+l}.$$

If the second expression in the step 2 of Algorithm 3.3 is chosen, then

$$|\delta JI(i, i + l)|_i = |\delta JI(i, k + q)|_i, \text{ and} \\ |\delta JI(i, i + l)|_{i+l} = fwd(i + q, i + l, |\delta JI(i, i + q)|_{i+q}),$$

³Here it is assumed that a typical B^+ -tree is of two levels. The results for a B^+ -tree of more than two levels can be calculated similarly as in Valduriez [32].

where $1 \leq p < k - i$ and $k - i \leq q \leq l - 1$. Also, if the first expression in the step 2 of Algorithm 3.3 is chosen, $UJI(i, i + l)$ is calculated as

$$1 + y(|\delta JI(i + p, i + l)|_{i+p}, \lceil \frac{|C_{i+p}|}{BT_f} \rceil, |C_{i+p}|) + y(|\delta JI(i + p, i + l)|_{i+p}, ||JI(i, i + p)||, |C_{i+p}|).$$

If the second expression in step 2 of Algorithm 3.3 is chosen, $UJI(i, i + l)$ is calculated as

$$1 + y(|\delta JI(i, i + q)|_{i+q}, \lceil \frac{|C_{i+q}|}{BT_f} \rceil, |C_{i+q}|) + y(|\delta JI(i, i + q)|_{i+q}, ||JI(i + q, i + l)||, |C_{i+q}|).$$

As it is noticed that there could be more than one choice of updating $JI(i, i + l)$ in step 2 of Algorithm 3.3, p or q is chosen such that the cost of updating $JI(i, i + l)$, i.e., $UJI(i, i + l)$ is the minimum. The third part is the cost of inserting or deleting $\delta JI(i, i + l)$ into or from $JI(i, i + l)$. The cost of updating the forward $JI(i, i + l)$ is

$$1 + y(|\delta JI(i, i + l)|_i, \lceil \frac{|C_i|}{BT_f} \rceil, |C_i|) + 2 * y(|\delta JI(i, i + l)|_i, ||JI(i, i + l)||, |C_i|). \quad (4.3)$$

The cost of updating the backward $JI(i, i + l)$ is similar. The way of calculating the update cost for a complete join index hierarchy structure is similar to that of a partial join index hierarchy structure. The update cost for a base join index hierarchy structure includes either deleting or inserting $\delta JI(k, k + 1)$ from or to $JI(k, k + 1)$.

4.3 Explanation of performance results

The performance study is conducted in the two group experiments for the index structures supporting navigations and associative searches. In the first one, five data structures, which support navigations, are compared in our performance study: (1) C-JIH as shown in Figure 5(a); (2) B-JIH as shown in Figure 5(b); (3) P-JIH as shown in Figure 5(c); (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*; and (5) P-ASR (partitioned *access support relation*), which stores the access support relations for paths $\langle C_0, A_1, C_1, A_2, C_2 \rangle$, $\langle C_2, A_3, C_3, A_4, C_4 \rangle$, and $\langle C_4, A_5, C_5 \rangle$. Notice that cases (2) and (4) correspond to two extreme cases of the *access support relation* method proposed in [15], 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 10, 11, 12, 15, 17 and 18 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.

Table 2: Class Parameters

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

Figure 10 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 some higher level join indices of the join index hierarchies; whereas C-JIH materializes all of the higher level join indices. P-ASR stores sequences of object identifiers for the partitioned paths. 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 P-ASR and B-JIH/B-ASR. The storage cost of P-ASR is almost the same as B-JIH/B-ASR when fan-outs are small and a little more when fan-outs increase, because the lengths of the partitioned paths are 2, 2, and 1.

Figure 11 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_0 and C_4 , and C_2 and C_5 weigh 20%, 40% and 40% in the total cost respectively. Notice that the navigation between C_0 and C_5 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, Full-ASR, and P-ASR. Full-ASR has the poorest performance because the whole ASR has to be retrieved (the relation is

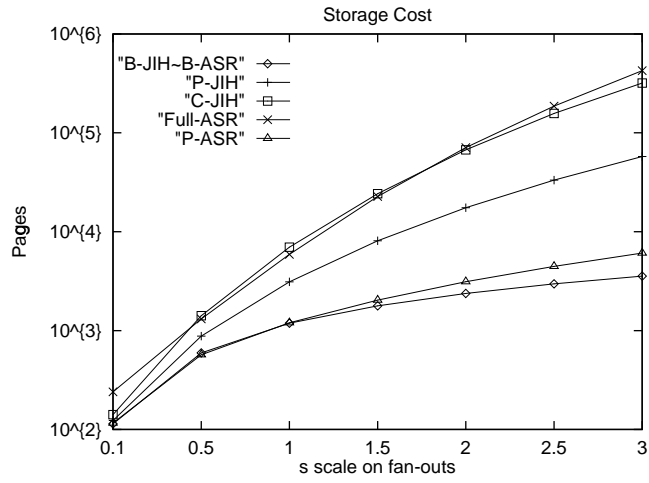


Figure 10: Storage Costs of B-JIH, P-JIH, C-JIH, Full-ASR, and P-ASR vs. Fan-outs.

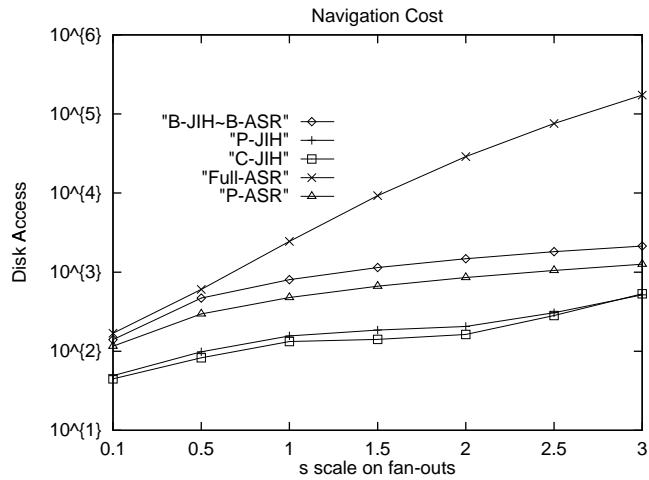


Figure 11: Navigation Costs of B-JIH, P-JIH, C-JIH, Full-ASR, and P-ASR vs. Fan-outs.

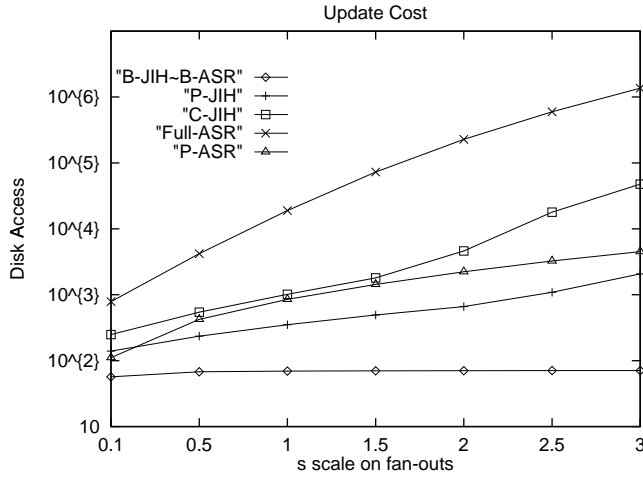


Figure 12: Update Costs of B-JIH, P-JIH, C-JIH, Full-ASR, and P-ASR vs. Fan-outs.

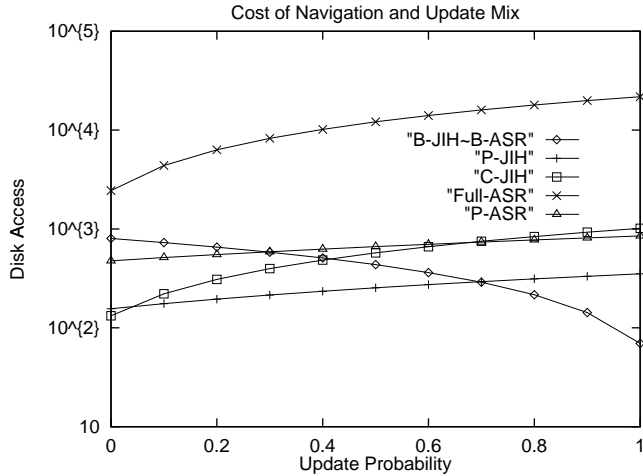


Figure 13: Costs of Navigation and Update mix for B-JIH, P-JIH, C-JIH, Full-ASR, and P-ASR.

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 and tail classes are required.

Figure 12 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 13 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 1.0. With medium frequent update (update probability between 0.05 and 0.7), the overall performance of P-JIH is better than that of others. C-JIH and B-JIH/B-ASR outperform others when the update frequency is at lower extreme and higher extreme, respectively.

Figure 14 presents the navigation costs vs. navigation selectivities. The scale s on fan-outs is set to be 1.0. The selectivity at C_0 is set from 0.001 to 0.5. The navigation cost grows as the navigation selectivity increases.

Figure 15 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 results⁴ 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. This also suggests that the fan-outs should be considered as an important factor for setting “fire walls” to avoid cost explosion.

In the second group experiment, six index structures, which support associative searches, are compared: (1) C-JIH as shown in Figure 5(a); (2) B-JIH as shown in Figure 5(b); (3) P-JIH as shown in Figure 16; (4) Full-ASR; (4) P-ASR; and (5) Nest which denotes the nested index in [4, 2]. Notice that the target node is $JI(0, 5)$ in the partial join index hierarchy in Figure 16.

⁴not shown here due to space limitation.

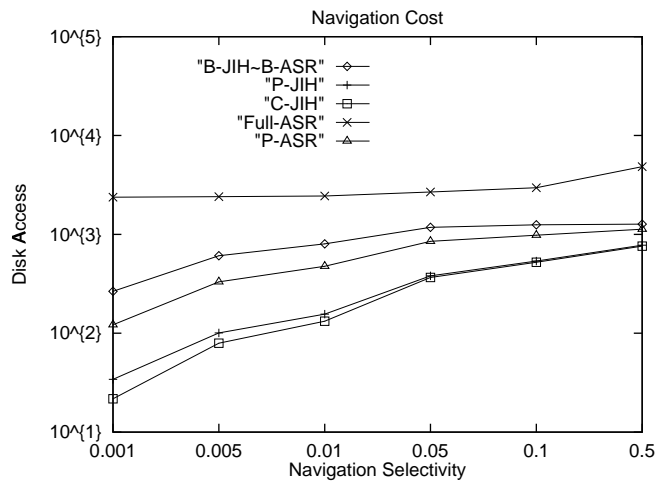


Figure 14: Navigation Costs of B-JIH, P-JIH, C-JIH, Full-ASR, and P-ASR vs. Navigation Selectivities.

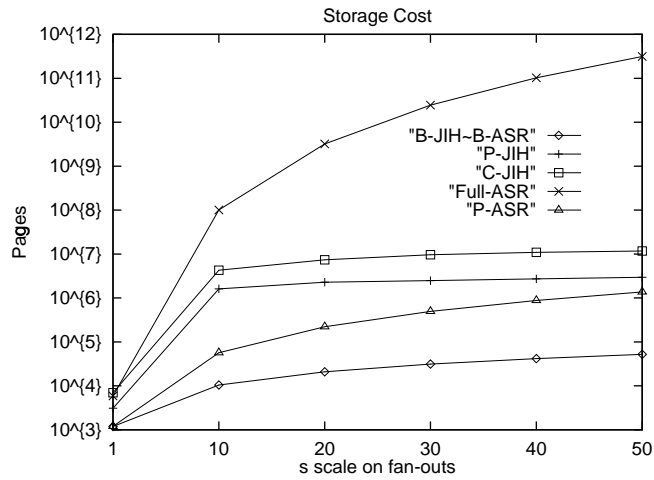


Figure 15: Storage Explosion with Large Fan-outs.

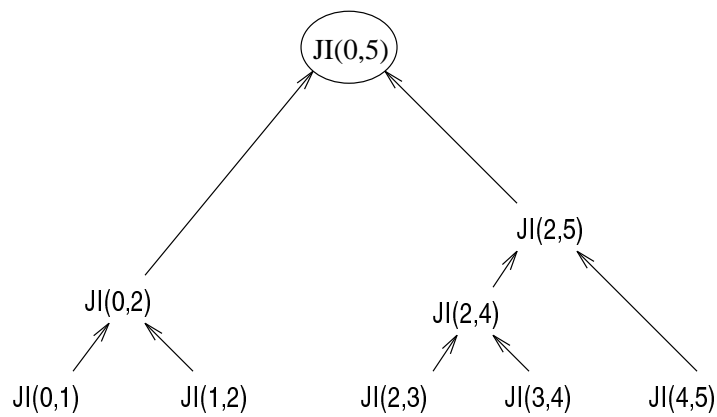


Figure 16: Partial Join Index Hierarchy for Supporting JI(0,5).

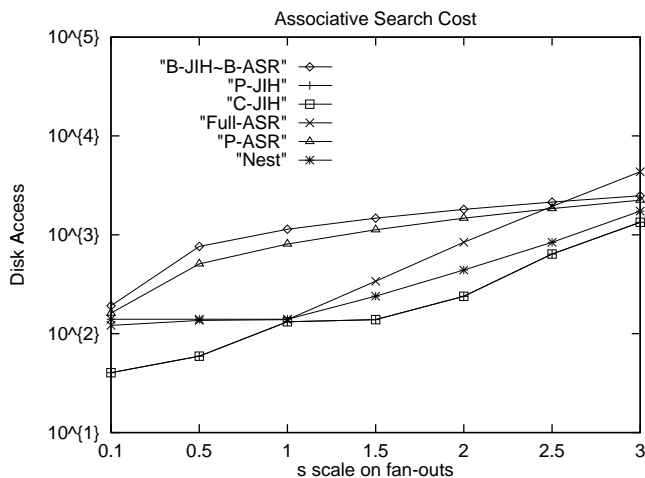


Figure 17: Associative Search Costs of B-JIH, P-JIH, C-JIH, Full-ASR, P-ASR and Nest vs. Fan-outs.

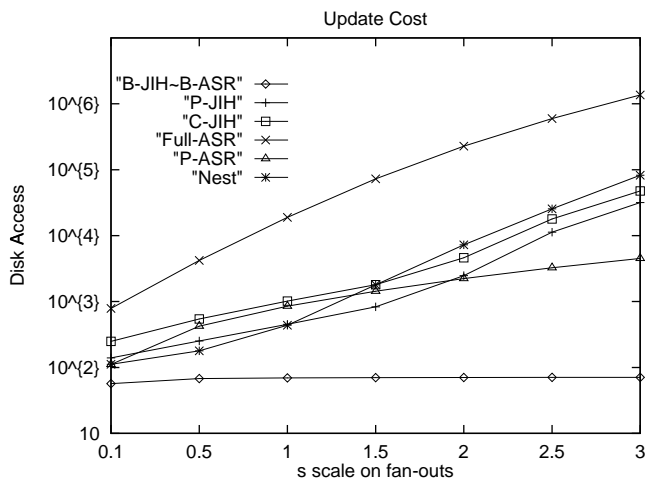


Figure 18: Update Costs of B-JIH, P-JIH, C-JIH, Full-ASR, P-ASR and Nest vs. Fan-outs.

Figure 17 presents how the associative search costs increase as the fan-outs grow (only the backward navigation between C_0 and C_5 is considered.). Since P-JIH and C-JIH support $JI(0, 5)$ directly, their associative search costs are the same. This is indicated by the overlap of their performance curves. P-JIH and C-JIH perform better than Nest since the root node $JI(0,5)$ of P-JIH and C-JIH is smaller than the nested index.

Figure 18 illustrates the update costs. When the fan-outs are small, the update costs of P-JIH and C-JIH are higher than that of the nested index. This reflects the fact that P-JIH and C-JIH maintain two copies for both forward and backward navigations while the nested index structure only keeps one (backward) copy and can only be employed for associative search (backward navigation). It is significant, however, that the update cost of the nested index grows faster than those of P-JIH and C-JIH, and exceeds them when the fan-outs become large (fan-outs scale $s > 1.5$). In a nested index, the reference information in a path has to be retrieved iteratively from the auxiliary index so that all the appropriate records in the primary index can be updated accordingly.

In summary, the performance study shows that P-JIH and C-JIH outperform B-JIH/B-ASR, Full-ASR, P-ASR, and Nest in navigation, associative search 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.

In a schema path involving class/subclass hierarchies, one may construct one (combined) base join index node or several join index files based on the cohesion of subclasses and access patterns. For example, if a set of subclasses have similar kinds of attributes and their objects are usually accessed together, it could be beneficial to construct one (combined) base join index node, which is in the same spirit of Kim, Kim and Dale [18] and Bertino [2]. A dynamic indexing approach is proposed by Chan, Goh and Ooi [6] which builds a multi-dimensional index for objects in a class hierarchy based on not only class hierarchy and attribute dimension but also query patterns. Similar ideas may be adopted when constructing base join index nodes.

Furthermore, there may exist more than one semantic linkage between two object classes. For example, a professor may *teach* a student (in a course), *supervise* a student (on research work), or *hire* a student (for some programming job). Thus, there may exist three kinds of semantic linkages between *professor* and *student* in this database. A join index node is for a particular kind of semantic association which cannot be mixed up with other kinds of semantic linkages since they carry different semantics. The schema paths should be stored in the schema (data dictionary) with the identification (such as by labeling) of each semantic linkage for each join index node.

Some relationships between different classes of objects may not be specified by existing attributes but by deduction rules or computational methods. For example, the voting eligibility of a stockholder could be defined by deduction rules based on his/her current shares of stocks, the stock holding history, etc. Thus, the linkage between the two classes, *Stockholder* and *Voter*, are defined by rules and instantiated by rule evaluation. Similarly, 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. [22]) 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 [22] 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 it is 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 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.

5.3 Optimal Join Index Hierarchy

In Section 3.1, an algorithm, Algorithm 3.1, is developed which discovers a join index hierarchy with minimal number of auxiliary nodes and minimal number of \bowtie_c operations for updating, for a given set of target nodes. Algorithm 3.1 may be used when information about class and query is not available, or one can expect that the variants of instance number and fan-out among classes are small.

If statistical information about classes and queries is known, a cost function can be associated with every join index hierarchy and Algorithm 3.1 can be modified to find the optimal hierarchy in terms of the cost function.

The cost of a join index hierarchy H , $C(H)$, can be defined as:

$$C(H) = w_1 * C_s(H) + w_2 * C_n(H) + w_3 * C_u(H)$$

where $C_s(H)$, $C_n(H)$, and $C_u(H)$ are cost of storage, navigation, and update, respectively; and w_1 , w_2 , and w_3 are relative weights for the costs.

The storage cost is the total number of pages needed to store the join indices in the hierarchy:

$$C_s(H) = \sum_{JI(i,j) \in H} ||JI(i,j)||$$

where $||JI(i,j)||$ is given in Equation 4.1.

The navigation cost is the weighted cost of all possible navigations:

$$C_n(H) = \sum_{i,j} p_{ij} * C_n(i,j)$$

where p_{ij} is the probability of navigation from class C_i to class C_j and $C_n(i,j)$ is the navigation cost from class C_i to class C_j , which can be calculated using Equation 4.2.

The update cost can be computed as:

$$C_u(H) = \sum_{i=0}^n q_i * \sum_{j \leq i \leq k \wedge JI(j,k) \in H} C_u(JI(j,k))$$

where q_i is the probability of update in class C_i and $C_n(JI(j,k))$ is the update cost for $JI(j,k)$, which is given in Equation 4.3.

Changes to Algorithm 3.1 are outlined as below.

- Information about the classes and queries, such as number of objects in a class, fan-out, navigation selectivity, etc., as well as parameters, such as w_i , p_{ij} , page size, etc., should be added to the input of the algorithm.
- The output of the algorithm is a set of auxiliary nodes which minimizes the cost function.
- Step 2 and 3 of the algorithm will be replaced by a procedure that returns a set s in S which minimizes $C(H)$, where H consists of nodes in s .

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. It is interesting to compare and/or integrate the join index hierarchy method with other object query optimization techniques, such as read-ahead buffering [26] and complex object assembly [14].

References

- [1] J. Banerjee, W. Kim, and K. C. Kim. Queries in object-oriented databases. In *Proc. Int. Conf. Data Engineering*, pages 31–39, Los Angeles, CA, February 1988.
- [2] E. Bertino. An indexing technique for object-oriented databases. In *Proc. Int. Conf. Data Engineering*, pages 160–170, Kobe, Japan, April 1991.
- [3] E. Bertino and P. Foscoli. Index organizations for object-oriented database systems. *IEEE Trans. on Knowledge and Data Engineering*, 7:193–209, 1995.
- [4] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Trans. Knowledge and Data Engineering*, 1(2):196–214, 1989.
- [5] J. A. Blakeley, W. J. McKenna, and G. Graefe. Experiences building the open OODB query optimizer. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 287–296, Washington, DC, May 1993.
- [6] Chee Yong Chan, Cheng Hian Goh, and Beng Chin Ooi. Indexing oodb instances based on access proximity. In *Proc. 13th International Conf. on Data Engineering*, pages 14–21, Birmingham, UK, 1997.
- [7] S. Choenni, E. Bertino, H. M. Blanken, and T. Chang. On the selection of optimal index configuration in OO databases. In *Proc. Int. Conf. Data Engineering*, pages 526–537, Phoenix, AZ, USA, February 1994.
- [8] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 383–392, 1992.
- [9] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Survey*, 25(2):73–170, June 1993.
- [10] G. Graefe and D. Maier. Query optimization in object-oriented database systems: a prospectus. In *Advances in Object-Oriented Database Systems*. Springer-Verlag, 1988.
- [11] T. Haerder. Implementing a generalized access path structure for a relational database system. *ACM Trans. Database Systems*, 3(3):285–298, September 1978.
- [12] K. A. Hua and C. Tripathy. Object skeletons: an efficient navigation structure for object-oriented database systems. In *Proc. Int. Conf. Data Engineering*, pages 508–517, Houston, TX, Feb 1994.
- [13] K. Kato and T. Masuda. Persistent caching: an implementation technique for complex objects with object identity. *IEEE Trans. Software Engineering*, 18(7), July 1992.
- [14] T. Keller, G. Graefe, and D. Maier. Efficient assembly of complex objects. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 148–157, Denver, CO, May 1991.
- [15] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 364–374, Atlantic City, NJ, May 1990.
- [16] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. Int. Conf. Very Large Database*, pages 290–301, Brisbane, Australia, August 1990.
- [17] C. Kilger and G. Moerkotte. Indexing multiple sets. In *Proc. Int. Conf. Very Large Database*, Santiago, Chile, September 1994.
- [18] 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*, pages 371–394. Addison-Wesley, 1989.
- [19] H. A. Kuno and H. A. Rundensteiner. Augmented inherited multi-index structure for maintenance of materialized path query views. In *Proc. Sixth International Workshop on Research Issues in Data Engineering*, pages 128–137, New Orleans, LA, 1996.
- [20] R. S. G. Lanzelotte, P. Valduriez, M. Ziane, and J. Cheiney. Optimization of nonrecursive queries in OODBs. In *Proc. Int. Conf. Deductive and Object-Oriented Databases(DOOD)*, Munich, Germany, December 1991.
- [21] C. C. Low, B. C. Ooi, and H. Lu. H-tree: a dynamic associative search index for OODB. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 134–143, San Diego, CA, May 1992.
- [22] W. Lu and J. Han. Distance-associated join indices for spatial range search. In *Proc. 8th Int. Conf. Data Engineering*, pages 284–292, Phoenix, AZ, Feb. 1992.

- [23] D. Maier and J. Stein. Indexing in an object-oriented DBMS. In *Proc. IEEE Int. Workshop on Object-oriented Database System*, pages 171–182, Asilomar, Pacific Grove, CA, September 1986.
- [24] T.A. Mueck and M.L. Polaschek. The multikey type index for persistent object sets. In *Proc. 13th International Conf. on Data Engineering*, pages 22-31, Birmingham, UK, 1997.
- [25] Beng Chin Ooi, Jiawei Han, Hongjun Lu, and Kian Lee Tan. Index nesting – an efficient approach to indexing in object-oriented databases. *VLDB Journal*, 5:215–228, 1996.
- [26] M. Palmer and S. B. Zdonik. FIDO: a cach that learns to fetch. In *Proc. Int. Conf. Very Large Database*, Barcelona, Spain, 1991.
- [27] S. Ramaswamy and P.C. Kanellakis. Oodb indexing by class-division. *SIGMOD Record*, 24:139–150, 1995.
- [28] D. Rotem. Spatial join indices. In *Proc. 7th Int. Conf. Data Engineering*, pages 500–509, Kobe, Japan, April 1991.
- [29] Sang Koo Seo and Yoon Joon Lee. Methodology for index configurations in object-oriented databases. *Information Sciences*, 93:187–210, 1996.
- [30] E. J. Shekita and M. J. Carey. Performance enhancement through replication in an object-oriented DBMS. In *Proc. ACM-SIGMOD Conf. Management of Data*, pages 325–336, 1989.
- [31] D. D. Straube and M. T. Ozsü. Queries and query processing in object-oriented database systems. *ACM Trans. Office and Information Systems*, 6(4):387–430, Oct 1990.
- [32] P. Valduriez. Join indices. *ACM Trans. Database Systems*, 12(2):218–246, 1987.
- [33] Z. Xie and J. Han. Optimization of queries containing complex selections, joins and aggregations. In *Proc. International Conference on Computing and Information*, Peterborough, Ontario, Canada, May 1994.
- [34] S. B. Yao. Approximating block accesses in database organizations. *Communications of the ACM*, 20(4):260–261, April 1977.

A Evaluation of Some Parameters in Cost Model

Table 1 lists some database parameters which will be used in our analytical cost model. The probability of an object in C_{j-1} which does not reference a particular object in C_j is

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

i.e.,

$$1 - \frac{|f_{j-1}|}{|C_j|}.$$

The probability of m objects in C_{j-1} which do not reference a particular objects in C_j is

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

The probability of a particular object in C_j which is referenced by m objects in C_{j-1} is

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

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

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

Hence,

$$fwd(i, j, k) = \begin{cases} [p(|C_{i+1}|, |f_i|, k)] & \text{if } j = i + 1 \\ [p(|C_j|, |f_{j-1}|, fwd(i, j - 1, k))] & \text{if } j > i + 1, \end{cases}$$

where

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

The number of tuples in $JI(i, j)$ is

$$|JI(i, j)| = |C_i| * fwd(i, j, 1).$$

Similarly,

$$bwd(i, j, k) = \begin{cases} [p(|C_i|, |r_i|, k)] & \text{if } j = i + 1 \\ [p(|C_i|, |r_i|, bwd(i + 1, j, k))] & \text{if } j > i + 1 \end{cases}$$

The number of tuples in $JI(i, j)$ can also be calculated by

$$|JI(i, j)| = |C_j| * bwd(i, j, 1).$$