The Indispensability of Dispensable Indexes

Elisa Bertino and Beng Chin Ooi

***Abstract*— The design of new indexes has been driven by many factors such as data types, operations, and application environment. The increasing demand for database systems to support new applications such as online analytical processing (OLAP), spatial databases, and temporal databases, has continued to fuel the introduction of new indexes. In this paper, we summarize the major considerations in developing new indexes, paying particular attention to progress made in the design of indexes for spatial, temporal and object-oriented databases (OODB). Our discussion focuses on the general concepts or features of these indexes, thus presenting the building blocks for meeting the challenges of designing new indexes for novel applications to be encountered in the future.**

***Keywords*— Indexing structures, query processing.**

1. Introduction

Database management systems (DBMS) have become widely accepted as a standard tools for manipulating data stored in secondary storage. *Indexes* — structures associated with database files — have been employed to provide fast and selective retrieval of records from large collections of data objects.

Traditionally, indexes are optional components (and hence *dispensable*) since a database file can be scanned in its entirety to retrieve the desired records. This was reasonable in the early days since database files were generally small then. However, DBMSs are increasingly being used for advanced applications (such as geographic information systems, multimedia information systems, text retrieval systems, and so on) where databases are usually voluminous because of the large number of records, huge object size and complex data types (such as images and spatial data). Accessing data in these applications without utilizing any indexing structures is inconceivable: indexes are *indispensable* in a DBMS today!

While there are other means of reducing I/O cost such as having more efficient buffer management, better clustering techniques, and more effective query optimization, the effects of indexes are much more evident. They are, therefore, the primary means of reducing redundant disk I/Os. Because performance is a crucial issue in database systems, the development of new indexing techniques has been an area of intense research and development.

In the last decade, we have witnessed the proliferation of new indexes. This has been fueled by the demands of advanced applications which are far more complex than the traditional business applications. Traditional indexing techniques like the B$^+$-tree cannot directly support these applications. Therefore, several new indexing techniques have been recently developed, specifically tailored to accommodating new applications. These new indexes often extend existing methods with additional data structures to accommodate the specific requirements of new applications, which often differ only in detail. This raises issues of extensibility, scalability, robustness, and general efficiency rather than just query efficiency of indexes. In this paper, we describe the development of indexes over the last decade. In particular, we focus on indexes for spatial, temporal, and object-oriented database systems without enumerating the many indexes described in the literature. We examine the main features and concepts behind some of these indexes, which form the building blocks for designing more specific indexes. We also use these indexes as examples in our discussion on issues such as specialized versus general-purpose indexes, single versus multiple indexes, and precomputation versus dynamic evaluation.

2. Application of Indexes

Recent advances in hardware technology have reduced the access times of both memory and disk tremendously. However, the ratio between the two remains at about 5 orders of magnitude. With the ever increasing volume of data that need to be retrieved from disk, the number of disk accesses becomes a very important performance parameter to optimize. Data must be organized on the disk in such a way that relevant data can be quickly located. This calls for the use of indexing structures that are designed to provide fast retrieval of a small set of answers from large data collections.

An index is a data structure that helps to organize the data by mapping a key value to one or more records containing the key value, providing a mechanism to efficiently identify the storage location of records. Balanced trees, such as B$^+$-trees, are examples of effective and adaptive structures for indexing and ordering large volumes of data. Their main strengths lie in how they organize the tree nodes into pages, and their logarithmic access and update time. They are so well accepted that they have been generalized for use in other applications such as multi-dimensional indexing. Apart from indexing techniques based on hierarchical organizations, many indexing structures have been developed based on address computation, such as hash based techniques.

E. Bertino is with Dipartimento di Scienze dell'Informazione, Universita' di Milano, 20135 Milano, Italy.
B. C. Ooi is with the Department of Information Systems and Computer Science, National University of Singapore, Singapore 119260.

Many indexes are designed based on the concept of proximity, by either treating objects with close indexed values as clusters or partitioning indexing space into subspaces. This inherent characteristic enables the index to be readily used as a guide in data clustering. Objects whose indexed values are near to each other can be co-located within the same disk block.

In addition to supporting direct access to a small set of objects for simple retrieval operations, indexes are heavily used in reducing the cost of expensive join operations where relationships between objects are dynamically computed. Consider the basic nested-loop join algorithm which computes the θ-join of the relations involved. It consists of choosing one relation R, called *outer* relation, and comparing each tuple of R with all the tuples of the other(*inner*) relation S against the θ join condition. The algorithm is outlined below.

```
JOIN(R,S)
input:      relation $R$ and $S$,
            predicate $\theta$.
output:     a list of pairs of objects.
begin
    while (get next tuple, $tupR$, of $R$) do
        while (get next tuple, $tupS$, of $S$) do
            if ($\theta(tupR, tupS)$) then
                add $< tupR, tupS >$ to the resulting list;
end.
```

The nested-loop algorithm can be applied to all kinds of joins by changing the predicate from simple relationships between alpha-numerical values to more complex relationships between objects, such as spatial relationships in a spatial database system. Assuming that both relations are very large, an obvious way of avoiding a sequential scan of the inner relation for every tuple of the outer relation is to use an index on the join attribute of the inner relation. The outer relation R is sequentially scanned, and for each tuple, the index is used to find the matching tuples of S. The number of pages accessed may be thus reduced from all the pages storing tuples of the relation to a small number of index pages and data pages. The use of indexes is so effective in optimizing join executions, that in some cases the indexes are dynamically built just for executing the join.

The nested-loop algorithm can be further optimized to exploit indexes of both relations. When indexes on the join attribute for both relations R and S exist, the join can be performed by traversing both indexes concurrently. Due to different data spaces defined by two subtrees, a subtree may be traversed more than once for a join between two relations. Such a join behaves like a hash-based join, where data are pre-partitioned into buckets before executing the join, buckets with the same address space are then fetched and tuples from these buckets are checked. However, unlike hash-based partitioning where data records are as randomly distributed into buckets as possible, data values are close within a defined range. A join execution strategy based on the use of indexes on both relations is hence much more flexible and efficient than a hash-based join when there is a selection since ranges of subtrees can be derived easily and values within each range are ordered.

3. Dimensions in Index Design

In this section, we briefly describe the requirements of spatial, temporal and object-oriented databases and their benefits on the design of indexes.

3.1. Spatial Indexes

Spatial data — data that are associated with spatial coordinates and extents such as points, lines, polygons, and volumetric objects — are becoming increasingly common in many of today's applications, including computer-aided design (CAD), geographic information systems (GIS), computational geometry and computer vision. In these applications, retrieval of data is primarily based on spatial proximity relationships amongst objects rather than on exact match of attribute values of objects. For example, in GIS, a query may be "retrieve all buildings that are *within* a certain distance from a particular road". *Spatial operators* like *intersection*, *adjacency*, and *containment* are more expensive to compute than conventional relational *join* and *select* operators. This is due to irregularity in the shapes of the spatial objects. For example, consider the intersection of two polyhedra. Besides the need to test all points of one polyhedron against the other, the result of the operation is not always a polyhedron but it may sometimes consist of a set of polyhedra.

Object representation and indexing structures are treated separately in spatial database applications. Objects of complex shape are approximated by simpler containers for retrieval purposes. The aim of such approximation is to filter out as much false hits as possible before performing more expensive testing on the actual complex spatial objects. Spatial query processing therefore consists of the following three logical steps:

1. The search space is pruned by traversing an index structure to determine a set of candidate objects, which is usually a superset of the answer. Unlike traditional indexes, the index structure is built on simpler containers of spatial objects. This approach reduces the index size (and hence storage) and results in faster traversals since the approximations rather than the actual objects are compared.
2. Some of the false hits in the first step can be further filtered away. This further filtering is based on the fact that two objects do not match if their approximations do not match. The effectiveness of this step depends on the approximation techniques. More accurate approximations are more expensive to compute and maintain but are more effective in reducing the actual object tests.
3. The actual objects are fetched and examined to determine those that satisfy the query.

Once again, we see how the index can play an important role in reducing I/O cost; its use reduces the amount of redundant data to be fetched and minimizes the computation costs as a side effect.

A critical issue in the design of a spatial access structure is how it partitions the data space and how it associates data with subspaces. Two points do not overlap unless they are the same point, and therefore point data can be organized into partitions with non-overlapping data subspaces. The association of an extended object with a data subspace or cell is not as straightforward as it is for a point, because extended objects may overlap with each other and may span multiple data subspaces. The complexity of indexes for extended objects is related to how such indexes organize these objects into groups. Indexing structures for extended objects can be characterized according to the way they associate objects with data spaces:

1. **The transformation approach.** The transformation approach comes in two flavors:

   - *Parameter space indexing.* Objects described by n vertices in a k-dimensional space are mapped into points in an nk-dimensional space. For example, a two-dimensional rectangle described by the bottom left corner (x_1, y_1) and the top right corner (x_2, y_2) is represented as a point in a four-dimensional space, where each attribute is taken from a different dimension. The points generated by such transformation can be stored directly in a point index. An advantage of such an approach is that it does not require any modification to the multi-dimensional base structure. The main drawback of the mapping scheme is that the spatial proximity between the k-dimensional objects may no longer be preserved. Consequently, intersection search can be inefficient. Also, the complexity of the insertion operation typically increases with higher dimensionality.

   - *Mapping to a single attribute space.* The data space is partitioned into grid cells of the same size, which are then numbered according to some curve-filling methods [XXX]. A spatial object is then represented by a set of numbers or one-dimensional objects. These one-dimensional objects can be indexed using conventional indexes such as B$^+$-trees. A B$^+$-tree can be directly used to index objects that have been mapped from a k-dimensional space into points in a one-dimensional space.

2. **The non-overlapping native space indexing approach.** This category comprises two classes of techniques:

   - *Object duplication.* A k-dimensional data space is partitioned into pairwise disjoint subspaces. These subspaces are then indexed. An object identifier is duplicated and stored in all the subspaces it intersects so that it can be correctly retrieved .

   - *Object clipping.* This technique is similar to the object duplication approach. Instead of duplicating the identifier, an object is decomposed into several disjoint smaller objects so that each smaller sub-object is totally included in a subspace.

   The most important property of object duplication and clipping approaches is that the data structures used are straightforward extensions of the underlying point indexing structures. Both points and objects with extensions can be stored together in one file without having to modify the basic structure. However, an obvious drawback is the duplication of objects which requires extra storage space and hence implies more expensive insertion and deletion operations. Another concern is that the density (the number of objects that contain one point) in a map space must be less than the page capacity (the maximum number of objects that can be stored in one page).

3. **The overlapping native space indexing approach.** The basic idea of this approach is to hierarchically partition the data space into a manageable number of smaller subspaces. While a point object is entirely included in one unpartitioned subspace, a non-zero sized object may extend over more than one subspace. To define a data space that properly contains all objects, data subspaces are allowed to overlap. These subspaces are organized as a hierarchical index and spatial objects are indexed in their native space. A major design criterion for indexes using such an approach is to minimize both the overlap between bounding subspaces and the coverage of subspaces. A poorly designed partitioning strategy may lead to unnecessary traversal of multiple paths. Further, the dynamic maintenance of effective bounding subspaces results into higher update costs.

Figure shows the evolution of spatial indexing structures over the last decade [XXX] A solid arrow indicates a relationship between a new structure and the original structures that it is based upon. A dashed arrow indicates a relationship between a new structure and the structures from which the techniques used in the new structure originated, even though some were proposed independently of the others. For a thorough survey and full references, please refer to [XXX].

Among all the indexes, the R-tree [XXX] has received most attention. As such, we single it out for further discussion here. The R-tree is a multi-dimensional generalization of the B-tree. Like the B-tree, the R-tree is height-balanced, and may involve node splitting and merging when objects are inserted and deleted. The hierarchical nature of the R-tree recursively partitions a data space into smaller subspaces to the level where objects in an unpartitioned subspace can be stored into one physical page. An internal node has a data space that bounds the data spaces of all the subtrees. Logically, the organization consists of a number of nested rectangles arranged so that each node corresponds to a spatial entity and contains descriptions of smaller spatial entities. Each non-root entity is completely contained in the entity corresponding to its parent node.

When performing a search, the decision to visit a subtree depends on whether the covering rectangle overlaps the query region. It is quite common for several covering rectangles in an internal node to overlap the query rectan-

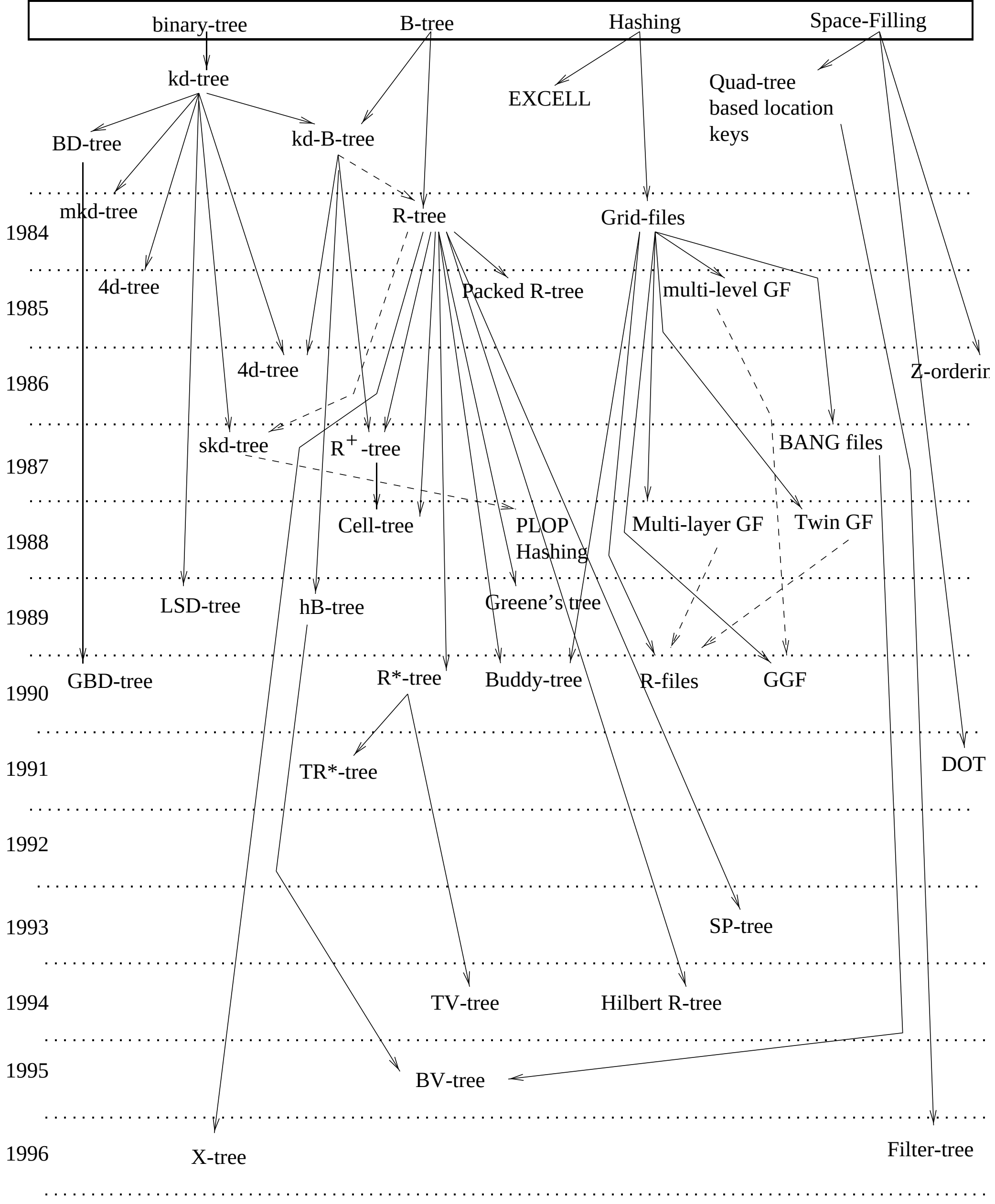

Fig. 1 Evolution of Spatial Index Structures

gle, resulting in the traversal of several subtrees. Therefore, minimization of overlap among the covering rectangles as well as the coverage of these rectangles is of primary importance in constructing the R-tree. Two algorithms involved in the process of minimization are the insertion and its node splitting algorithms. Of the two, the splitting algorithm affects the index efficiency more. An optimized splitting algorithm is able to yield some improvement [XXX].

Dynamic hierarchical spatial indexes are sensitive to the order of data insertion. A tree may behave differently for the same data set over a different sequence of insertions. Data rectangles previously inserted may result in a bad split of the R-tree after some insertions. Hence it may be worth to perform some local reorganization, which is however expensive. The R-tree deletion algorithm supports the reorganization of the tree to some extent, by forcing the entries in underflowed nodes to be inserted from the root. Using the idea of reinsertion of the R-tree, Beckmann et al. [XXX] proposed a reinsertion algorithm to handle node overflows. The reinsertion increases the storage utilization, but it can be expensive for large trees. The R*-tree improves upon the R-tree at the price of having more expensive insertion operations. Many variants of the R-tree have been proposed, which mainly differ on their node splitting algorithms, which employ different strategies to reduce overlap as well as coverage. Performance gains, however, depend very much on data distribution and volume [XXX]. Quite often, the gain becomes less significant as the size of database grows.

3.2. Temporal Indexes

In a temporal database, temporal data are modeled as collections of line segments. These line segments have a *start time* (T_s), an *end time* (T_e), a *time-invariant attribute* (key), and a *time-varying attribute*. Each object may be associated with a *valid time* and a *transaction time*. Applications can support one of the two or both times. Valid time represents the time interval when the database fact is true in the modeled world, whereas transaction time is time interval when the database fact is current in the database. With this definition, retroactive and predictive updates are allowed in a valid time database but not in a transaction time database. Valid time intervals of a time-invariant object may overlap although they are often disjoint, and they are typically closed intervals due to pro-active insertion. On the other hand, transaction time intervals of a time-invariant object do not overlap and they are defined at the time when new versions are inserted. Transaction times which are system-generated follow a serialization order of transactions and hence are monotonically increasing. The lastest versions have progressive time span and have open time intervals. These properties pose unique problems to the design of temporal indexes.

Indexes are typically designed to meet the demands of the intended applications by considering their data types and classes of queries. The choice of temporal index is thereforely dependent on the time dimension(s) supported. In general, temporal indexes can be categorized as transaction-type, valid-time and bitemporal, based on the time dimension(s) supported. Within each category, indexes can be further classified according to query types: key-only, time-only and time-key. Most temporal indexes have been designed for transaction time databases and based on variants of R-trees and variants of B^+-trees.

Temporal data can be indexed based on its start time, end time, or the whole interval, together with the time-varying attribute or time invariant attribute. A B^+-tree can be used to index transaction times on start time (or end time). It is not efficient for answering time-slice queries since no information on the data space is captured in the index. To answer a time-slice query, the end time of the search interval is used to get the leaf node that contains the record whose start time is just before the search end time. From there, all the nodes to the left have to be searched. While this problem can be alleviated by replicating records in leaf nodes whose smallest start time intersects the record time interval, replication increases storage cost as well as the height of the index which affects the query and update cost.

The Write-Once B-tree (WOBT) [XXX] is a modification of the B^+-tree, that indexes out-of-date data on write-once optical disks (WORMs). Many variations of this method have since been proposed, including the Time-Split B-tree (TSB-tree) [XXX], the Persistent B-tree [XXX], and the Multiversion B-tree [XXX].

The TSB-tree is one of the first temporal indexes that support search based on key attribute and transaction time. An internal node contains entries of the form <*att-value*, *trans-time*, *Ptr*>, where *att-value* is the time-invariant attribute value of a record, *trans-time* is the timestamp of the record and *Ptr* is a pointer pointing to a child node.

Searching algorithms are affected by how a node is split and the information it captures about its data space. In the TSB-tree, two types of node splits are supported, key value and time splits. A key split is similar to a node split in a conventional B^+-tree where a partition is made based on a key value. For the time split, an appropriate time is selected to partition a node into two. Unlike key split, all record entries that persist through the split time are replicated in the new node which stores entries with time greater than the split time. If the number of different attribute values in a node is more than $\lfloor M/2 \rfloor$ (M is the number of entries in a node), a key split is performed; otherwise the node is split based on time. If no split time can be used except the lowest time value among the index item, a key split is executed instead of time split. To search based on key and time, index keys and times of internal nodes are used respectively to guide the search. With data replication, data whose time intersects the data space defined in the index entries are properly contained in its subtree, and this enables very fast search space pruning. Node splitting along time dimension entails some amount of data replication which may affect its storage requirement and query performance. The constraint that data versions of a time invariant key do not overlap, i.e. the semantics of data type, is also built into the index, somewhat restricting the extensibility of the index.

The underlying characteristics of temporal databases naturally lead to a direct use of spatial indexes. However, unlike spatial applications where non-spatial data are usually stored and indexed separately from spatial data, temporal attribute data such as time-invariant key and time-varying key are indexed together with temporal data. For temporal applications, to index temporal data and its key, the R-tree can be implemented as a two-dimensional R-tree (2-D R-tree) or a three-dimensional R-tree (3-D R-tree). To use a 2-D R-tree, time intervals [T_s, T_e] are treated as line segments in a two-dimensional space, with keys on the other dimension. To index temporal data using a 3-D R-tree, the time intervals and keys have to be mapped into points (key, T_s, T_e) in a three-dimensional space. Both implementations can handle all queries typical of temporal databases. However, unlike spatial databases, the attributes have different domains. The ranges of key attributes are well defined and are comparatively much smaller to the time range which is progressing with time. A node splitting algorithm that splits based on coverage is likely to keep on splitting along one dimension.

A spatial index like the R-tree cannot directly handle intervals with open end-time. An entry in the internal node of the R-tree contains a minimum bounding rectangle (MBR) that describes the data space of its child node. When data intervals are not closed, the MBR cannot be defined properly, and these affect the splitting algorithm that makes use of space coverage to distribute the data into two groups. It is however possible to use the current time or the largest time due to the proactive insertion as an estimate during node splitting and data insertion. The approximation is likely to have some effect on the performance.

One approach to temporal indexing using the R-tree directly is by making use of two R-trees, one to index temporal data that are 'dead' as line segments and the other to index data that are 'alive' as points. When intervals in the second R-tree become dead, they are moved to the first R-tree. This is necessary, as searching on the second R-tree is more complex than searching on the first R-tree. To answer a time-slice query, both R-trees must be searched.

3.3. OODB Indexing

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

1. **Class Hierarchy**. A key concept in object-oriented data models (OODM) is the notion of a class hierarchy (also known as an inheritance hierarchy). In an OODM, each entity is modeled by an object; objects are classified into classes based on their properties and behavior. A class can be specialized into a number of subclasses, giving rise to a hierarchy of classes which captures the IS-A relationship between classes: an object of a class is also an object of its ancestor classes (superclasses), and the properties of a class are inherited by all its descendant classes (subclasses). An implication of the inheritance hierarchy for the evaluation of an object-oriented query is the *class scope* of the query; that is, the classes over which the query will be evaluated. Unlike the relational model where a query on a relation R retrieves tuples from only R itself, an object-oriented query on a class C has two possible interpretations. In a *single-class query,* objects are retrieved from only the queried class C itself. In a *class-hierarchy query,* objects are retrieved from all the classes in the class hierarchy rooted at C since any object of a subclass of C is also an object of C. The interpretation of the query type (single-class or class-hierarchy) is specified by the user. To facilitate the evaluation of such types of queries, a *class-hierarchy index* needs to support efficient retrieval of objects from a single class, as well as from all the classes in the class hierarchy.
2. **Aggregation Hierarchy**. Besides the class-subclass relationship, another important type of relationship between classes is the attribute-domain relationship. In the OODM, the domain of an attribute of a class can be either a primitive type (e.g. integer, real, string) or a complex domain (i.e. a class domain). A class can thus be defined as a nested structure of classes, giving rise to an aggregation hierarchy (or class composition hierarchy). Note that the concepts of a class hierarchy and an aggregation hierarchy are orthogonal. An aggregation index must index object paths efficiently. Without efficient index support, the evaluation of queries involving classes along aggregation paths can be slow as it requires access to multiple classes.
3. **Methods** While the structural properties of objects in the OODM are described by attributes, their behaviorial properties are defined by methods which are procedures to either access the values of the attributes, or operate on the attribute values. A method can be used in object-oriented queries as a derived attribute that returns a value (or an object), or as a predicate that returns a boolean value. To speed up the evaluation of OO query predicates that involve methods, efficient index support is necessary.

A class-hierarchy index is characterized by two parameters: the hierarchy of classes to be indexed, and the index attribute of the indexed hierarchy. There are two approaches to class-hierarchy indexing:

- Class-dimension based approaches [XXX] partition the data space primarily on the class of an object.
- Attribute-dimension based approaches [XXX] partition the data space primarily on the indexed attribute of an object.

Space partitioning of both approaches is illustrated in Figure . While the class-dimension based approach supports single-class queries efficiently, it is not effective for class-hierarchy queries due to the need traversing multiple single-class indexes. On the other hand, the attribute-dimension based approach generally provides efficient support for class-hierarchy queries on the root class (i.e., retrieving objects of all the indexed classes), but is inefficient for single-class queries or class-hierarchy queries on a sub-hierarchy of the indexed class hierarchy, as it may need to access many irrelevant leaf nodes of the single index structure. To support both types of queries efficiently, the index must support both ways of data partitioning [XXX]. However, this is not a simple or direct application of multi-dimensional indexes, since totally ordering of classes is not possible and hence partitioning along the class dimension is problematic.

A second important issue in indexing techniques for OODBs is related to aggregation hierarchies and navigational accesses along these hierarchies. Navigational access is based on traversing object references; a typical example is represented by graph traversal. Navigations from one object in a class to objects in other classes in a class aggregation hierarchy are essentially expensive pointer chasing operations. To support navigations efficiently, indexing structures that enable fast path instantiation are necessary. In practice, many of these structures are based on pre-computing traversals along aggregation hierarchies. Consider the following example: class Organization references class Author, class Author references class Book, class Book references class Chapter. From the path, objects from one class can be retrieved via objects of another class along the path. To efficiently supporting aggregation hierarchies traversal several techniques have been defined.

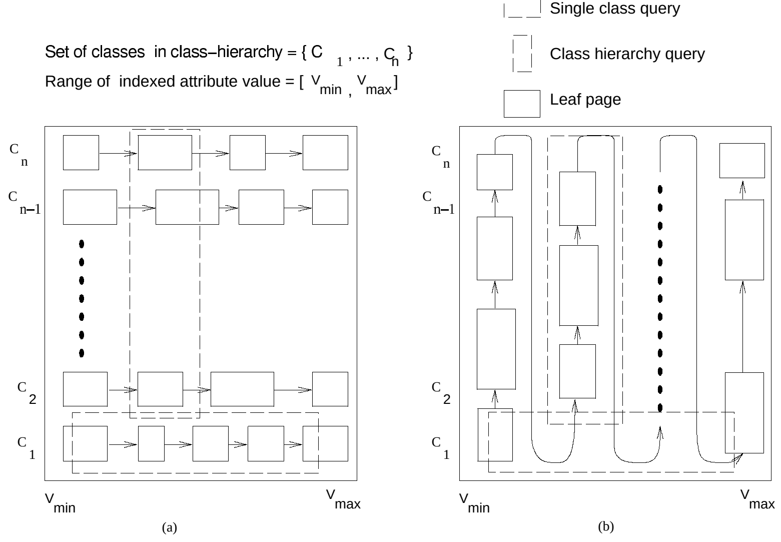
Single class query
Class hierarchy query
Leaf page
Set of classes in class–hierarchy = { C 1 , ... , Cn }
Range of indexed attribute value = [Vmin , Vmax]
Cn
Cn–1
C2
C1
Vmin
Vmax
(a)
(b)

Fig. 2 Space partitioning: a) Class-based; b) Attribute-based

Figure illustrates six path indexing schemes by using the notion of *indexing graph*. An indexing graph contains a node for each class in the indexed path; it moreover contains an edge from the node representing a class C to the node representing a class C' if there is an indexing relationship from class C to class C'. An indexing relationship from class C to class C' means that an entry in the index contains as key values the identifiers (or values) of the instances of class C and it associates with each such key value the identifiers of the instances of class C' that reference, directly or indirectly, the key value. The multi-index approach maintains an inverted index for each edge. As can be seen from Figure .a, only backward traversal is supported. The second approach is that based on the well known join index [XXX], where two B$^+$-trees are maintained for join pairs between two classes. Both forward and backward traversals are supported, but the update cost and storage cost is much higher than before (see Figure .b). For both methods, the number of indexes that need to be accessed for navigational access is proportional to path length. To alleviate this problem, direct association between objects of the first class and last class along the path are maintained in a single nested index [XXX]. The major problem of such an index is update operations that require access to several objects in order to determine the entries that need update (see Figure .c). The nested index can be further generalized to support access from the objects to all parent objects [XXX], as shown in Figure .d. To reduce update overhead and yet maintain the efficiency of path indexing structures, paths can be broken into subpaths which are then indexed separately [XXX]. Figures .d and .e show two examples where a path is split into several subpaths and different indexing techniques are allocated on each subpath. An effective splitting and allocation strategy is highly dependent on the query and update patterns and frequencies. Therefore adequate index allocation tools should be developed to support the optimal index allocation.

Query processing involving an user-defined method may require the execution of such method for a large number of instances. Because a method can be a general program the query execution costs may become prohibitive. Possible solutions, not yet fully investigated, are based on method precomputation; such approaches, however, make object updates rather expensive.

4. Scalability

Many application databases grow in size daily. In spatial databases, an increase in database size means more overlap between extended data, which implies that it becomes harder for indexes to partition data objects into cells effectively. For applications involving temporal databases, data volume increases as time progresses. For OODB, new classes and their instances, and new relationships, may be added to an existing database. Thus, a good index structure must be able to support growth in data volume without performance degradation.

For multi-dimensional indexing structures, another scalability criteria is in terms of number of dimensions of the data space. An efficient index like the R-tree has been shown to perform poorly for high dimensional data [XXX]. With increase in the number of dimensions, the number of entries per node decreases, resulting in a higher tree and increased overlap between subtrees.

To avoid the complexity resulting from a high number of dimensions, it may be more efficient to partition the attributes into groups and employ a multi-dimensional index for each group. Of course, the issue on how partitioning should be performed needs to be addressed as attributes are not totally independent. Very often, we use some more important attributes to remove false hits. For example, in image content-based retrieval by color, the specific color is more important than where that color appears in the image. Given that color has more discriminating power compared to other attributes, an index based on color can be used to filter objects before using their spatial descrip-

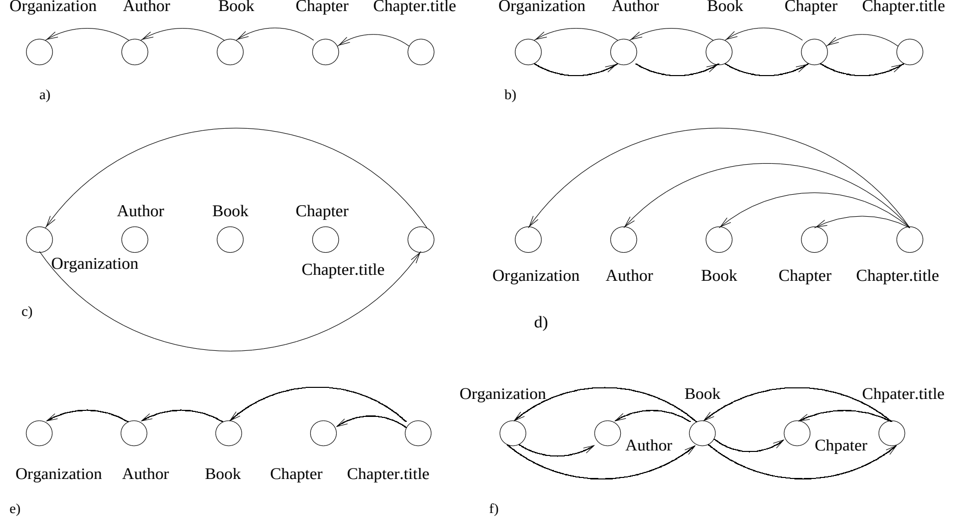

Fig. 3 Indexing graphs: a) Multi-index; b) Join Indexes; c) Nested Index; d) Path-index; e) Path Splitting; f) Path Splitting.

tions. For this sort of application, a general k-dimensional index may not be efficient.

5. Precomputation Versus Dynamic Retrieval

A join index [XXX], just as its name implies, is an indexing structure used for join operations. The main idea of join indexing is to precompute the join and store the result in an index file. Hence, the computation of join operations can be converted to looking up the join index file. Each record of the join index file contains only object identifiers of the matching object pairs to make the file relatively small. The efficiency of a join index will be proportional to its size which is in turns dependent on the join selectivity. If the join is more selective, the join index is small. However, a join with a larger join selectivity, which can be close to the Cartesian product, can make the index quite large. In this case, the join index is not appropriate.

Join indexes reduce join processing time at the expense of higher update cost. The update cost has been the main deterrent factor for adopting a join index. A compromise is to precompute the join results partially. Rotem [XXX] extended the concept of the join index to spatial join, i.e. join of spatial objects under spatial predicate, with grid file [XXX] as the underlying structure. This approach converts geometric computation for certain spatial relationships such as *within a certain distance* into a simple join index. Based on the join index, we can tell which are the data pages that contain objects having predefined spatial relationships.

The conventional join index can be generalized to a cluster-based join index in which objects are grouped into clusters based on proximity, and each record of the join index represents a pair of clusters in which some of their members satisfy the join condition. Most high dimensional databases are indexed using multidimensional indexes such as the R-tree, in which entries in the internal nodes provide effective and precomputed clusters. The size can be used to fine tune the join index to achieve a balance between update cost and retrieval cost to suit individual application.

6. Summary and Outlook

Over the last decade, many indexes have been designed and developed primarily driven from the need of supporting different data models and different data types. Many indexes are similar in features and design concepts, differing only in detail. While new indexes are being designed for new applications, existing indexes are enhanced or extended to cope with increasing data volume and new features that need to be indexed.

Another factor driving significant extensions to indexing techniques is related to advances in computer architectures. Such extensions are required to fully exploit the performance potential of new architectures, such as in the case of parallel architectures, or to cope with limited computing resources, such as in the case of mobile computing systems. Taking into account the specific architecture introduces an additional dimension in the design of indexes; questions related to the scalability of many indexes, such as the ones discussed in this paper, to different architectures (in particular parallel ones) are important and still need to be fully addressed.

New applications, in addition to the ones already mentioned, also play an important role in dictating extensions to indexing techniques and in offering wider contexts in which traditional techniques can be used. Two relevant such areas include data warehousing systems and the Web.

Data warehousing systems are characterized by queries involving large number of joins and computation of aggregate functions. However, updates are seldom. Therefore, techniques such as join indexes and view materialization strategies can be used here; techniques such as bit-mapped indexes, projection indexes, and bit-sliced indexes have also been proposed and adopted in commercial systems. However, research on indexing techniques and index allocation for data warehouses is still at the beginning.

The Web is characterized by the fact that information is higly distributed, data are not regularly structured, and the situation evolves day-by-day. Information retrieval techniques can be used here; some of the simplest techniques, developed in the information retrieval area, are in fact used by commercial browsers. However, it is not clear yet how more sophisticated techniques, such as clustering or search reformulation based on the user feedback, could be used here.

Acknowledgement: We would like to thank anonymous referees and our colleagues for valuable suggestions and comments.

References

[1] B. Becker, S. Gschwind, B. Seeger T. Ohler, and P. Widmayer. On optimal multiversion access structures. In *Proc. 3rd International Symposium on Large Spatial Databases*, pages 123–141. 1993.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331. 1990.

[3] S. Berchtold, D.A. Keim, and H.P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. 22nd International Conference on Very Large Data Bases*, pages 28–39. 1996.

[4] E. Bertino. On indexing configuration in object-oriented databases. *VLDB Journal*, 3(3):355–399, 1994.

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

[6] E. Bertino, R. Sacks-Davis, B. C. Ooi, K. L. Tan, J. Zobel, B. Shidlovsky, and B. Cantania. *Indexing Techniques for Advanced Database Systems.* Kluwer, (to appear), 1997.

[7] C.Y. Chan, C.H. Goh, and B. C. Ooi. Indexing OODB instances based on access proximity. In *Proc. 13th International Conference on Data Engineering*, pages 14–21. 1997.

[8] M.C. Easton. Key-sequence data sets in indelible storage. *IBM Journal of Research and Development*, 30(12), 1986.

[9] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proc. 1989 ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 247–252. 1989.

[10] D. Greene. An implementation and performance analysis of spatial data access methods. In *Proc. 5th International Conference on Data Engineering*, pages 606–615. 1989.

[11] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57. 1984.

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

[13] S. Lanka and E. Mays. Fully persistent b+-trees. In *Proc. 1991 ACM SIGMOD International Conference on Management of Data*. 1991.

[14] D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proc. 1989 ACM SIGMOD International Conference on Management of Data*, pages 315–324. 1989.

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

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

[17] J. A. Orenstein. A comparison of spatial query processing techniques for native and parameter spaces. In *Proc. 1990 ACM SIGMOD International Conference on Management of Data*, pages 343–352. 1990.

[18] D. Papadias, Y. Theodoridis, T. Sellis, and M. J. Egenhofer. Topological relations in the world of minimum bounding rectangles: A study with R-trees. In *Proc. 1995 ACM SIGMOD International Conference on Management of Data*, pages 92–103. 1995.

[19] D. Rotem. Spatial join indices. In *Proc. 7th International Conference on Data Engineering*, pages 500–509. 1991.

[20] B. Salzberg, and V.J. Tsotras. A comparison of access methods for time-evolving data. *ACM Computing Survey*, (to appear), 1997.

[21] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R$^+$-tree: A dynamic index for multi–dimensional objects. In *Proc. 13th International Conference on Very Large Data Bases*, pages 507–518. 1987.

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

[23] Z. Xie and J. Han. Join index hierarchy for supporting efficient navigation in object-oriented databases. In *Proc. 20th International Conference on Very Large Data Bases*, pages 522–533. 1994.