A Tree Index Framework For Databases

BIN NIE

A Thesis
In
The Department
Of
Computer Science

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science

Concordia University

Montréal, Québec, Canada

NOVEMBER 2003

Your file Votre référence
ISBN: 0-612-91097-0
Our file Notre référence
ISBN: 0-612-91097-0

Abstract

A Tree Index Framework For Databases

Bin Nie

The search tree index framework, a sub-framework of the Know-It-All Project, is used to develop a generalized search tree that provides the basis for common tree access methods used in database systems. The search tree index framework covers one-dimensional tree structures, point access structures, and special multi-dimensional structures. It also covers sequential queries, exact match queries, range queries, approximate queries, and similarity queries.

It applies the Standard Template Library modularity concept in the analysis, architecture, design and interface and takes advantage of reuse capabilities in modern programming languages such as generic programming and design pattern. By using modularization design, the system is designed as an integrated set of layers including algorithms layer, proxy layer and physical storage layer. Layering design technique provides a mechanism to decompose functionality. The design separates index, data, and data Reference/Page reference; uses iterators perform general queries on search tree structures, while providing a clean interface for these queries to define both positions within indexes or files, as well as to refer to a collection of information.

The framework can be adapted to the needs of any application by simply changing some of the building blocks and is designed for maximum flexibility and the simplest extension.

Acknowledgments

I would like to thank my supervisor, Dr Gregory Butler, for his guidance and finance support. I also extend my gratitude to Dr. Peter Grogono and Dr. Thomas Fevens who carefully read the final draft of the thesis and gave valuable suggestions. Also I would like to acknowledge Ashraf Gaffar. My thesis is based on his work. I had helpful discussions with Ju Wang, Lugang Xu, Jingxue Zhou, Yue Wang when doing the implementation. Finally, I would like to thank the Quebec government for the financial support I received during my studies at Concordia University.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Database index is the search engine for database management system. A specialized index is a good way to support a specific database application in a specific domain using domain-specific access methods to enhance searching performance. However, these specialized access methods are usually hand-coded from scratch. As a result, developing, extending, and modifying an index system is a difficult task. Furthermore, the effort required to implement and maintain it is high.

Reusing the products of the software development process is an important way to reduce software costs and to make programmers and designers more efficient. As a good object-oriented reuse technology, framework development [FS99] is a rapidly gaining acceptance due to its ability to promote reuse of design and source code. Frameworks are application generators that are directly related to a specific domain and typically resulting in increased productivity and faster time-to-market. Hence, customizing an index framework can largely reduce the cost of providing a new index.

1.1 The Problem

Efficient data access is one of the properties provided by a database management system. The most common mechanism to achieve this goal is to associate an index with a large, randomly accessed data file on secondary storage. The index is an auxiliary data structure intended to help speed up the retrieval of records in response to certain searching conditions.

For disk files, an index allows the number of disk accesses to be reduced. An index speeds up retrieval by directing the searcher to the small part of the file containing the desired item. An index may be physically integrated with the file or physically separated. Usually the index itself is a file. If the index file is large, another index may be built on top of it to speed retrieval further, and so on.

Many techniques for organizing index structure and algorithm have been proposed. A large variety of specialized access methods have been developed to solve specific problems. Among them are those based on the tree hierarchical approach such as B+-tree [DC79], B-tree [BM72], B*-tree [DK73], K-D-B-tree [JTB81], LSD-tree [HSW89], hB-tree [LS901,LS902], R-tree [GA84], R+-tree [SRF87] and so on, those based on hashing techniques such as Grid file [NHS84], BANG file [MF87], R-file [HSW90], Z-hashing [HSW88] , and so on. While some of this work has supported a specific database application in a specific domain using predetermined structures and access methods as well as data types and queries, the approach of developing domain specific access methods is problematic. These specialized access methods are usually hand-coded from scratch, normally requiring substantial knowledge of the underlying file system to be built correctly and efficiently. As a result, the effort requires a lot of development time, and cost, normally affordable only by large corporations. Furthermore, there is obvious reason to assume this trend will continue. As new database applications need to be supported, new access methods will be developed to deal with them efficiently.

Hellerstein et al. [HNP95] proposed a way to further generalize searching trees at the tree functionality level by introducing a generalized searching tree called GiST. The

GiST provides abstraction of common tree operations, such as search, insertion and deletion. Other tree structures, such as B+-trees and R-trees, can be built as extensions of the GiST. These do not add generalization-covering alternatives such as the internal structure of a tree's node, nor several other aspects of search tree design. GiST generates flexible code with some hot spots that can be easily adjusted to develop different index search tree in different applications. However, as other previous works, its source code is often complex and not easy to follow or comprehend, especially when there is no design documents other than the few pages explaining how to adapt the hot spots. The source code itself, largely influenced by the C programming language, has poor object-oriented style.

An alternative of developing new data structures from scratch makes use of framework technology to develop a framework for a family of indexes, and reuse it to develop different indexes for different applications. This largely reduces the cost of providing a new index. These frameworks produce an application that is less expensive in terms of cost and time investments.

1.2 The Solution

Today, a framework is recognized as the specialized infrastructure code which enhances efficiency, understandability, and maintainability. It provides reuse at the level of domain knowledge, requirements, architecture, micro-architecture, design, and code, in the context of a product line. Mohamed Fayad et al [FS99] describe the follow benefits to developers as below:

- ***Modularity*** -- Frameworks enhance modularity by encapsulating volatile implementation details behind stable interfaces. Framework modularity helps improve software quality by localizing the impact of design and implementation changes. This localization reduces the effort required to understand and maintain existing software.

- ***Reusability*** -- The stable interfaces provided by frameworks enhance reusability by defining generic components that can be reapplied to create new applications. Framework reusability leverages the domain knowledge and prior effort of experienced developers in order to avoid re-creating and re-validating common solutions to recurring application requirements and software design challenges. Reuse of framework components can yield substantial improvements in programmer productivity, as well as enhance the quality, performance, reliability and interoperability of software.

- ***Extensibility*** -- A framework enhances extensibility by providing explicit hook methods that allow applications to extend its stable interfaces. Hook methods systematically decouple the stable interfaces and behaviors of an application domain from the variations required by instantiations of an application in a particular context. Framework extensibility is essential to ensure timely customization of new application services and features.

- ***Inversion of control*** -- The run-time architecture of a framework is characterized by an "inversion of control." This architecture enables canonical application processing steps to be customized by event handler objects that are invoked via

the framework's reactive dispatching mechanism. When events occur, the framework's dispatcher reacts by invoking hook methods on pre-registered handler objects, which perform application-specific processing on the events. Inversion of control allows the framework (rather than each application) to determine which set of application-specific methods to invoke in response to external events (such as window messages arriving from end-users or packets arriving on communication ports).

The Standard Template Library (STL) [SL95] for C++ is a good example of generic programming. STL is a library of high quality and efficiency with great emphasis on code reuse, certainly influenced by the language of implementation of that library.

Another recent literature of OO-design is integration of design patterns [GHJ95] together with frameworks. Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience. Patterns can be viewed as abstract descriptions of frameworks that facilitate widespread reuse of software architecture. Similarly, frameworks can be viewed as concrete realizations of patterns that facilitate direct reuse of design and code. One difference between patterns and frameworks is that patterns are described in language-independent manner, whereas frameworks are generally implemented in a particular language. However, patterns and frameworks are highly synergistic concepts, with neither subordinate to the other. The next generation of object-oriented frameworks will explicitly embody many patterns and patterns will be widely used to document the form and contents of frameworks.

1.3 Our Work

Know-It-All (KIA) [BCC02] is a framework methodology research project. Its goals are to investigate methodologies for the development, application, and evolution of frameworks and to develop a framework for database management systems as a case study for the methodology research. It is written in C++, with some Java for user interfaces, and XML for communication of data between the C++ framework and the Java tools. The user interfaces will provide a full range of query mechanisms, from icons for canned queries, to forms, to textual queries in set comprehension languages, and diagrammatic queries. The database index framework is one of the sub frameworks.

This thesis intends to generalize a good quality framework of database search trees, create a sub framework of the KIA project and produce several instances of the index framework. The framework applies the Standard Template Library [SL95] modularity concept in the analysis, architecture, design and interface, which is used for primary-memory data structure libraries both to classify functionality and reuse implementation. STL attempts to provide taxonomy among primary-memory data structures, which is used both to classify functionality and reuse implementation. This work attempts to start a similar taxonomy but for manipulating data on secondary storage. The difference in performance and capability of secondary storage from primary storage requires a separate solution. It covers one-dimensional access method such as B+-tree, point access method such as K-D-B-tree [JTB81], hB-tree [LS901,LS902], and spatial access method, such as R-tree [GA84], R*-tree [BKS90], SS-tree [WJ96], SR-tree [KS97], X-tree [BKK96] and so on. It also covers sequential queries, exact match queries, range queries, approximate queries, and similarity queries. By generalizing the

parameter part that search-tree developers possibly need to specialize, a developer is not required to write all the components from scratch so that new search trees can be built more easily. By understanding the core, reusable components of a series of layers of a search tree index, a generalized search tree can be built and different search trees can be developed from the generalized components significantly faster than working from scratch. This generalization can be done by taking advantage of reuse capabilities in modern programming languages such as generic programming and design pattern. This thesis does not intend to construct new access methods but is an engineering exercise, combining GiST algorithm and plenty of well known design ideas in a way that provides a new and interesting solution to this important problem.

1.4 Overview of the Thesis

Chapter 1 gives an introduction with an overview of the work of the research group. Chapter 2 gives background introduction so that the reader can understand our work. First, the background of index search technology is introduced. Second, the development environment of generic programming, STL modularization concepts and design patterns are discussed. Chapter 3 presents the design of the index tree framework. Chapter 4 describes our implementation of the framework. Problems and implementation considerations are discussed in detail at the end of the chapter. General issues in framework extension and instantiation are discussed on the basis of related literature and our experience. Chapter 5 is the conclusion and suggests some future work.

Unless otherwise indicated, all diagrams in the thesis are UML [BEJ99] diagrams, and we discuss data types and programming mechanisms in terms of C++.

Chapter 2 Background

Basically, our work involved three fields: frameworks, design patterns and tree access methods. In our work, we apply framework and design patterns technology to build a tree index. Therefore, it is necessary to give brief background information about the three fields.

2.1 Search Tree Access Methods.

Index is a key technology for database systems. Indexes help access information. To make a large amount of data useful, it has to be organized and classified into logical parts, and then an index added. The index tells us where to find the data of interest in a data collection. Using a specific search method, we can go through an index and get a location where the data is to be found. Many techniques for organizing index structure and algorithm have been proposed. Search trees are efficient data structures that are widely used as indexes in database system.

A large variety of search trees have been developed to solve specific problems. They can be viewed in three categories, one-dimensional search trees, point search trees and spatial search trees. In the next section, we will briefly present some of these index search trees including some one-dimensional search trees such as B-tree, B+-tree, and some spatial access methods such as R-tree, R*-tree, SS-tree, SR-tree.

2.1.1 B-tree

B-trees [BM72] are designed to solve how to access and maintain efficiently an index that is too large to hold in memory, so the index itself must be external and is

organized in pages that are blocks of information transferred between main memory and backup storage such as disk file.

A B-tree [BM72] of order 2d is a multi-way search tree of order 2d with the following properties:

1. All the leaves are on the same level.
2. It is completely balanced.
3. All non-leaf nodes (except root node) have at most 2d keys and 2d +1 pointers and at least d keys and d+1 pointers.
4. The root is either a leaf or has at least two children.

In a B-tree, one tree node can be made to correspond to a page. Non-leaf pages are called index pages here as shown in Figure 2.1, containing search keys and pointers to lower-level children. Leaf pages as shown in Figure 2.2 contain search keys and references to data items or records. The minimum number of keys in a B-tree of order 2d and depth h is $1+\sum_{i=1}^{h} 2d(d+1)^{i-1}$, the maximum height of a B-tree with n keys is $\log_{d+1}(n/2d)$.

The B-tree is always completely balanced, and the length of retrieval path is at most the height of the B-tree. The cost of a search for a B-tree, which has order 2d and has n keys is O (log n). Insertion of B-tree will proceeds from the root to locate the proper leaf node for insertion. If the leaf node is already full, it will split it. When insertion splits a node, 2d+1 keys are split to 2 new nodes. A separator key is promoted to the parent ode. If the parent node is already full, the parent node is split too. In the worse case, splitting propagates to the root node and the tree increases in the height by one level. Hence, insertion cost of B-tree is equal to find operation cost O (log n) + splitting cost O (log n),

which is at most twice as much as a find. Deletion in a B-tree will call search operation to locate the node. The key can be found in either a leaf or a non-leaf node. The deletion operation may cause underflow, if a node contains less than d keys. It needs to redistribute the keys to a neighbor node. The deletion cost of B-tree is same as the insertion cost.

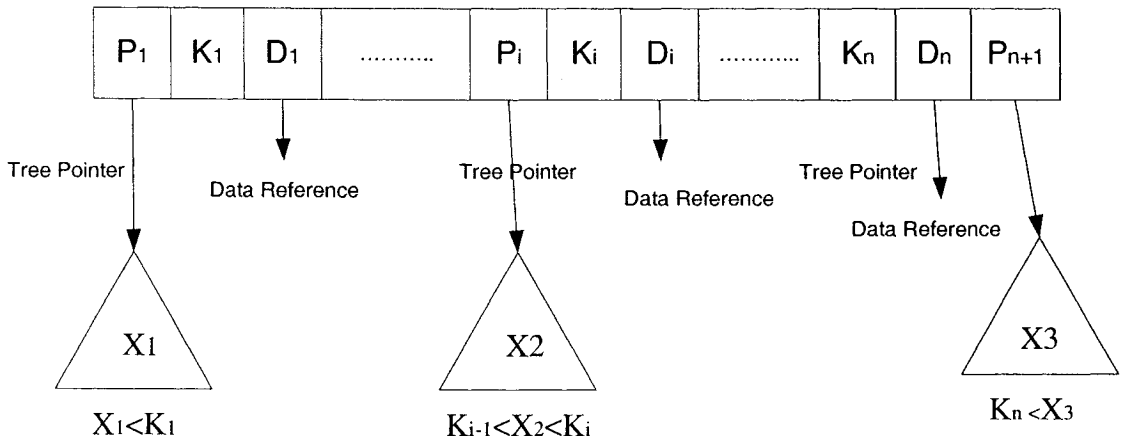

Figure 2.1: B-tree node with n-1 search values

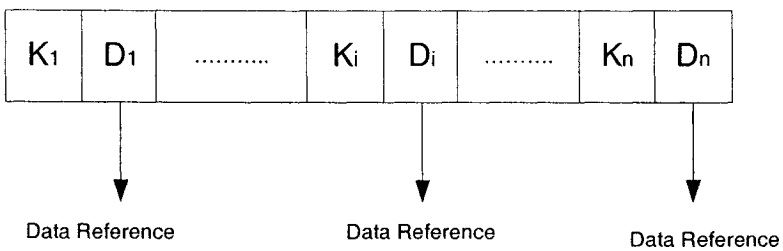

Figure 2.2: B-tree Leaf Page with m-1 Search Keys

The power of B-tree lies in the following significant advantages:

1. Storage utilization is guaranteed to be at least 50% and should be considerably better in the average case [BM72]

2. The balance is maintained dynamically at a relatively low cost. No overly long branches exist, and random insertions and deletions are accommodated to maintain balance [FZ92]

2.1.2 B+-tree

The B+-tree, proposed by D. Comer [DC79], is designed to improve the sequential search for the key in B-tree. All keys in the B+-tree reside in the leaves. The upper levels, which are organized as a B-tree, consist only of an index, a road map to enable rapid location of the index and leaves.

Figure 2.3 shows the logical separation of the index and leaves, which allows *index nodes* and leaf nodes to have different formats or even different sizes. In particular, leaf nodes are usually linked together left-to-right, as shown in Figure 2.3. To improve the performance of reverse traversal, the leaf pages are doubly linked together. The doubly linked list of leaves is referred to as the sequence set. The key and record information are not in the upper-level, but contained in the sequence set. Sequence set links allow easy sequential processing.

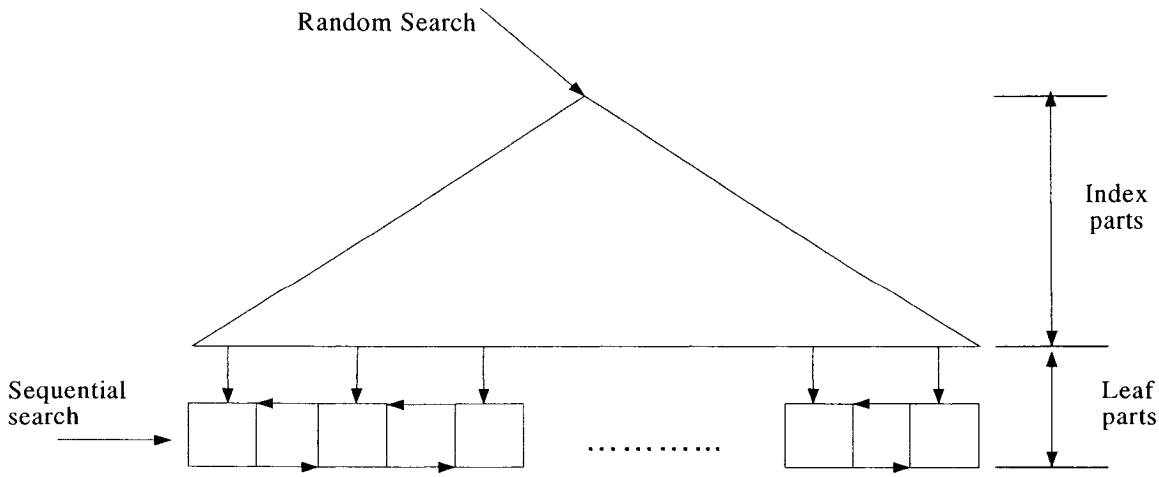

Figure 2.3: B+-tree with Separate Index and Key Parts

Insertion and find operations in a B+-tree are processed in a way similar to insertion and search operations in a B-tree. When a leaf splits in two, instead of promoting the middle key, the algorithm promotes a copy of the key, retaining the actual key in the right or left leaf. The resulting node structure, as shown in Figure 2.4, is different from that in a B-tree. Search operations differ from those in a B-tree in that searching does not stop if a key in the index equals the query value. Instead, the nearest right or left pointer is followed, and the search proceeds all the way to a leaf.

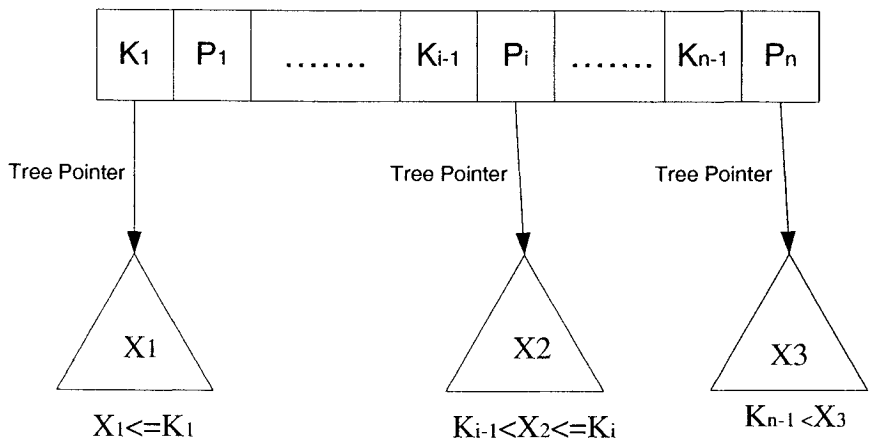

Figure 2.4: B+-tree Node

The B+-tree retains the search and insertion efficiencies of the B-tree but increases the efficiency of searching the next record in the tree from **O** (log n) to O (1). Instead, the nearest right or left pointer is followed, and the search proceeds all the way to a leaf.

During deletion in a B+-tree, the ability to leave non-key values in the index part as separators simplifies the operation. The key to be deleted must always reside in a leaf so its removal is simple. As long as the leaf remains at least half full, the index need not be changed, even if a copy of a deleted key is propagated up into it.

2.1.3 R-tree

An R-tree [GA84] is a height-balanced tree similar to a B+-tree with index records in its leaf nodes. Nodes correspond to disk pages and the structure is designed so that a spatial search requires visiting only a small number of nodes. Nodes at the same tree level may overlap.

A spatial database consists of a collection of tuples representing spatial objects, and each tuple has a unique identifier, which can be used to retrieve it. Leaf nodes in an R-tree contain entries of the form (*I, tuple-identifier*) where *tuple-identifier* refers to a tuple in the database and *I* is an d-dimensional rectangle which is the bounding box of the spatial object indexed: $I = (I_0, I_1, \ldots I_{d-1})$ where d is the number of dimensions and I_i is a closed bounded interval [a, b] describing the extent of the object along dimension *i*. Non-leaf nodes contain entries of the form (I, child-pointer) where child-pointer is the address of a lower node in the R-tree and *I* covers all rectangles in the lower node's entries. Figure 2.5 shows some rectangles organized into an R-tree.

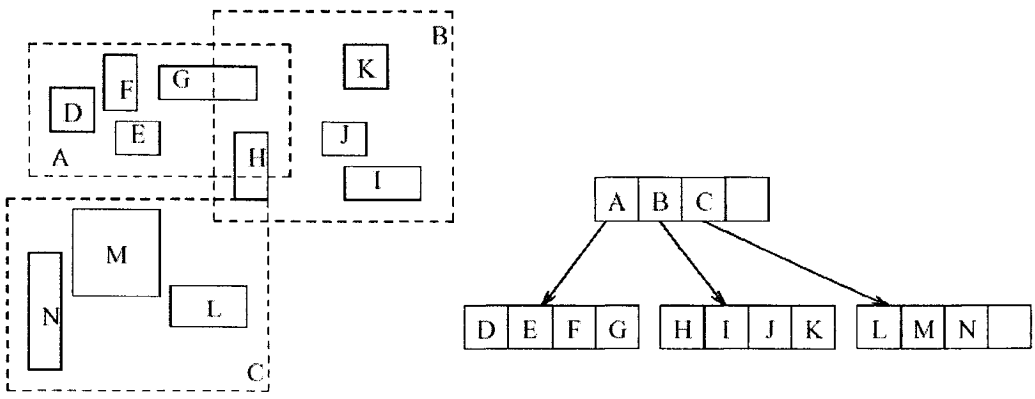

Figure 2.5: Rectangles Organized into an R-tree

Let M be the maximum number of entries that will fit in one node and let $m<=M/2$ be a parameter specifying the minimum number of entries in a node. An R-tree satisfies the following properties:

- Every leaf node contains between m and M index records unless it is the root.
- For each index (*I, tuple-identifier*) in a leaf node, I is the smallest rectangle that spatially contains the n-dimensional data object represented by the indicated tuple.
- Every non-leaf node has children between m and M, unless it is the root.
- For each entry (*I, child-pointer*) in a non-leaf node, I is the smallest rectangle that spatially contains the rectangles in the child node.
- The root node has at least two children unless it is a leaf.
- All leaves appear on the same level.

The search algorithm descends the tree from the root in a manner similar to a B+-tree. However, more than one sub-tree under a visited node may need to be searched because the bounding boxes at the same tree level may overlap. Even for point queries, there may be several intervals that intersect the search point. Hence, the search cost depends on the structure of the tree.

To insert a new object, the minimum bounding box and the object reference are inserted into the tree. In contrast to searching, only a single path from the root to the leaf is traversed. At each level, a child node is chosen so that its corresponding bounding box needs the least enlargement to enclose the data object's bounding box. If several intervals satisfy this criterion, one descendant is chosen whose rectangle needs the least enlargement to include the new entry. This guarantees that the object is inserted only

once, i.e., the object is not dispersed over several buckets. Once the leaf level is reached, the object is ready to be inserted. If this requires an enlargement of the corresponding bucket region, adjustment is done appropriately and the change is propagated upwards. If there is not enough space left in the leaf, the leaf is split and the entries are distributed between the old and the new page. Then each of the new bounding boxes needs to be adjusted accordingly and the split is propagated up the tree.

Similarly for deletion, first, an exact match query is performed for the object to be deleted. If the object is found, it is deleted. If the deletion causes no underflow, a check is performed whether the bounding box can be reduced in size. If so, this adjustment is made and the change is propagated upwards. On the other hand, if the deletion causes the node capacity to drop below 50%, the node contents are copied into a temporary node and the original node is removed from the tree. All bounding boxes have to be adjusted. Afterwards all orphaned entries of the temporary node are reinserted into the tree. In his original paper, Guttman [GA84] discussed various policies to minimize the overlap during insertion. Guttman suggests several algorithms, including a simpler one with linear time complexity and a more elaborate one with quadratic complexity.

2.1.4 R*-tree

The R*-tree [BKS90] is another variant of the R-tree. Based on a careful study of the R-tree behavior under different data distributions, Beckmann et al. identified several weaknesses of the original algorithms. In particular, the insertion phase is claimed to be critical for search performance. The design of the R*-tree introduces a policy called forced reinsert: if a node overflows, it does not split right away. Instead, about 30% of the

maximal number of entries are removed from the node first and then reinserted into the tree. In addition, to solve the problem of choosing an appropriate insertion path, the R*-tree not only takes the area parameter into consideration, but tests the area, margin and overlap parameters in different combinations. The R*-tree differs from the R-tree mainly in the insertion algorithms; deletion and searching are essentially unchanged.

2.1.5 SS-tree

The SS-tree [WJ96], as shown in Figure 2.6, is an improvement of the R*-tree and enhances the performance of nearest neighbor queries by modifying the following respects:

- It employs bounding sphere rather than bounding rectangles for the region shape.
- Its algorithms divide points into isotropic neighborhoods and enhance the performance on nearest neighbor queries.
- The advantage of using bounding spheres for the region shape is that it only requires nearly half the storage compared to bounding rectangles.
- Since a sphere is determined by the center and the radius, it can be represented with as many parameters as the dimensionality plus one. On the other hand, the number of parameters required for a rectangle is double the dimensionality, because a rectangle is determined by the lower and the upper bound of every dimension. This permits doubling the fan-out of nodes and reduces the height of trees.
- When a node or a leaf is full, the R*-tree reinserts a portion of its entries rather than splits it, unless reinsertion has been made on the same tree level. On the

other hand, the SS-tree reinserts entries unless reinsertion has been made at the same node or leaf. This promotes the dynamic reorganization of the tree structure.

- On the insertion of a point, the insertion algorithm determines the most suitable sub tree to accommodate the new entry by choosing a sub tree whose centroid is the nearest to the new entry.
- When a node or a leaf is full, the split algorithm calculates its coordinates variance (as in R*-tree) on each dimension from the centroids of its children and chooses the dimension with the highest variance for splitting it.

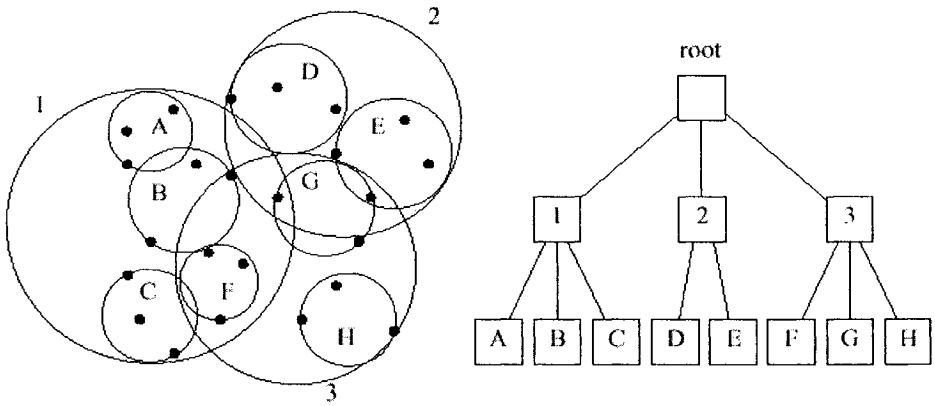

Figure 2.6: The SS-tree Structure

2.1.6 SR-tree

The SR-tree [KS97], as shown in Figure 2.7, is an extension of the R*-tree and the SS-tree. The distinctive feature of the SR-tree is the combined utilization of bounding spheres and bounding rectangles. This improves the performance on nearest neighbor

queries by reducing both the volume and the diameter of regions compared with the R*-tree and the SS-tree.

A region of the SR-tree is specified by the intersection of a bounding sphere and a bounding rectangle. Bounding rectangle divides points into smaller volume regions, but tends to have longer diameters than bounding spheres, especially in high-dimensional space. Bounding spheres divides points into short-diameter regions, but tends to have larger volumes than bounding rectangles. SR-tree combined the use of bounding sphere and bounding rectangle, as the properties are complementary to each other. SR-tree insertion algorithm is based on SS-tree's centroid-based algorithm; and deletion algorithm is similar to that of the R-tree.

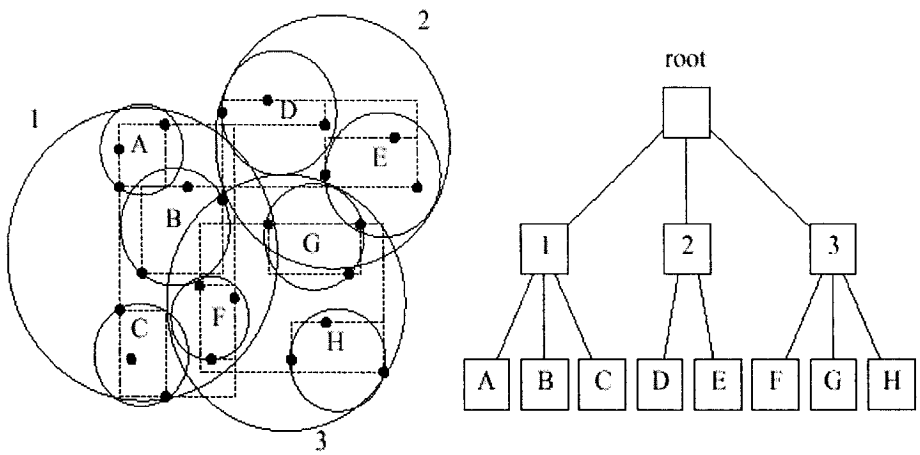

Figure 2.7: The SR-tree Structure

2.2 Generalized Search Tree (GiST)

Hellerstein et al [HNP95] introduces an index structure, called a Generalized Search Tree (GiST), which is a generalized form of an R-tree [GA84]. The GiST allows new data types to be indexed and supports an extensible set of queries. In addition, the authors claim that the GiST unifies previously disparate structures used for currently common data types. For example, both B-trees and R-trees can be implemented as extensions of the GiST.

In detail, a GiST is a balanced tree structure with tree nodes containing (*p, ptr*) pairs, where is a predicate that is used as a search key, and is the identifier of some tuple in the database for a leaf node and a pointer to another tree node for a non-leaf node. A GiST has the following properties:

1) Every node contains between *min* and *max* index entries unless it is the root.
2) For each index entry (*p, ptr*) in a leaf node, is true when instantiated with the values from the indicated tuple (i.e., *ptr* holds for the tuple).
3) For each index entry (*p, ptr*) in a non-leaf node, *p* is true when instantiated with the values of any tuple reachable from *ptr*.
4) The root has at least two children unless it is a leaf.
5) All leaves appear on the same level.

In principle, the keys of a GiST may be any arbitrary predicates that hold for each datum below the key. In practice, the keys come from a user-defined object class, which provides a particular set of methods required by the GiST. To adapt the GiST for different uses, users are required to register the following set of six methods:

- **Consistent(E,q):**given an entry $E = (p, ptr)$, and a query predicate q, returns false if $p \wedge q$ can be guaranteed unsatisfiable, and true otherwise.
- **Union(P):**given a set P of entries (p_1,ptr_1), ... (p_n,ptr_n), returns some predicate r that holds for all tuples stored below ptr_1 through ptr_n.
- **Compress(E):**given an entry $E = (p, ptr)$, returns an entry (pp, ptr) where pp is a compressed representation of p.
- **Decompress(E):**given a compressed representation $E = (pp, ptr)$, returns an entry (r, ptr) where r is a decompressed representation of pp.
- **Penalty(E1,E2):**given two entries $E_1 = (p_1, ptr_1)$, $E_2 = (p_2, ptr_2)$, returns a domain specific penalty for inserting E_2 into the sub-tree rooted at E_1, which is used to aid the splitting process of the insertion operation.
- **PickSplit(P):**given a set P of $M+1$ entries (p, ptr), splits P into two sets of entries P_1 and P_2, each of size kM, where k is the minimum fill factor.

The three methods of GiST provide algorithms for search, insertion and deletion operations:

- **SEARCH** can be used to search any dataset with any query predicate by traversing as much of the tree as necessary to satisfy the query. The general search algorithm is controlled by the user-specified **Consistent()** method, which returns true if the predicate is satisfied and false in vice versa. The same **Consistent()** method applies to both index node and leaf node. But usually the satisfied conditions are different for the index node and leaf node, no matter what kinds of searching, an exact matching or a range/window query. So further checking is needed after leaf entries are fetched according to the algorithm.

Another restriction is that only one type of query can be conducted in one program because function Consistent() is used in the search instead of a pointer to a function.

- **INSERT** guarantees that the GiST's tree structures are balanced. User defined key method **Penalty()** is used for choosing a sub-tree to insert; method **PickSplit()** is used for the node splitting algorithm; method **Union()** is used for propagating changes upward to maintain the tree properties.
- **DELETE** determines a leaf with specified key and removes entry (p, ptr) and, if required, maintains the balance of the tree using method **Union()** (as in **INSERT**). For underflow, it uses the B+-tree "borrow or coalesce" technique if there is a linear order; otherwise it uses the R-tree reinsertion technique.

2.3 Survey of Generalization of Search Trees

2.3.1 Tree Structure

A good survey of search trees is provided by Volker Gaede and Oliver Günther [GG98], though one-dimensional search tree B-tree and their variants, a variety of multidimensional search trees, such as R-trees [GA84] and their variants: R*-trees [BKS90] and R+-trees [SRF87], to other point search trees include K-D-B-trees [JTB81], and hB-trees [LS901,LS902]. Also GiST [HNP95] presents the essential nature of tree structures as observed in the GiST paper [HNP95]. Figure 2.7 shows the basic structure of such a tree. There are two major components: Index and Leaf. Both Index and Leaf are further composed of tree nodes, which usually have a fixed size (usually the page size). Within each tree node, there is a series of entries and an empty space if the node is not

full. The index nodes, also called interior nodes, are used as a directory to guide the search down the tree. The leaf nodes, also called data nodes, contain data entries or pointers to the actual data. The leaves may be further connected via a linked list to allow for partial or complete scanning. Figure 2.8 shows the typical structure of leaf nodes.

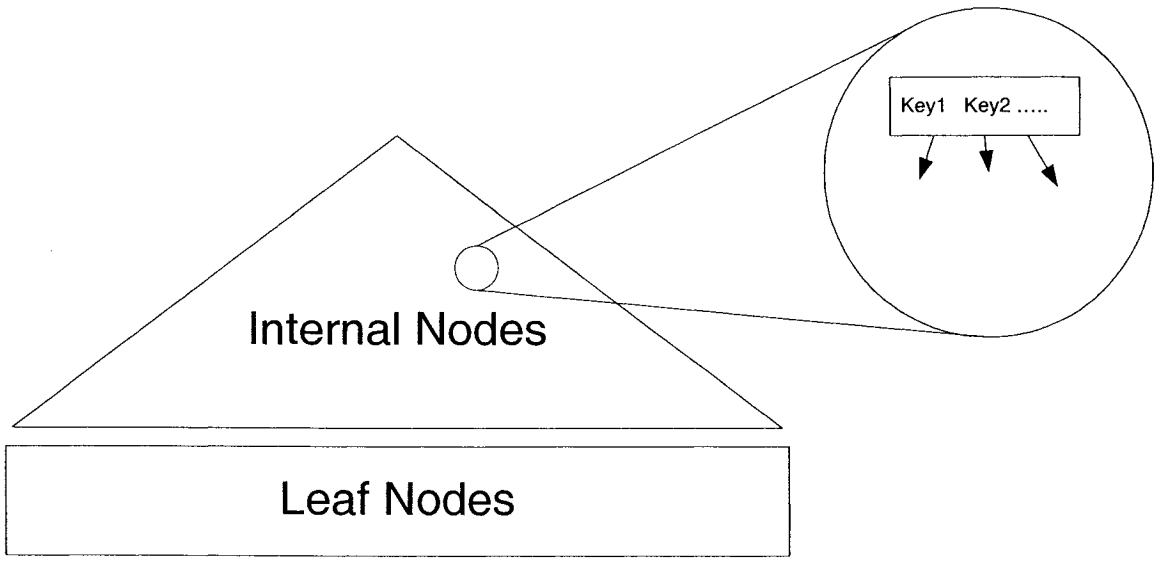

Figure 2.8: Sketch of a Database Search Tree

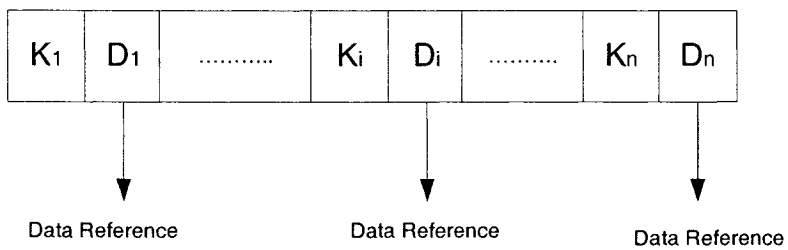

Figure 2.9: Typical Structure of Leaf Node

From the review of tree basic structure, it is possible to deduce that tree nodes of most database search trees, from one-dimensional B+-tree to multi-dimensional point and spatial search trees, satisfy the following common properties:

1. Each index node contains entries between min and max unless it is the root.

2. Each entry in a leaf node contains a key uniquely identifying the object and data object or a data reference to a data object.
3. Each entry in an index node contains a key and a pointer to a child node.
4. The root has at least two children unless it is a leaf.

Different search trees are used to solve specific problems in different domains, and therefore, have different structures for the keys in properties (2) and (3). Examples of key structures include integer values for data in a B-tree, and rectangles for regions in an R-tree. Index key structures may be different from the leaf key depending on the kind of tree. For example, in an R-tree, leaf keys may represent points and index keys represent regions. Basically, a tree structure is used to classify information by breaking a whole into parts, and repeatedly breaking the parts into subparts. In this sense, an index key logically matches all data stored in its subparts.

2.3.2 Tree Functionality

Different kinds of search trees are related in that they share the common concept of a search tree. That is, they have certain common operations such as insertion, deletion and search. Other possible functions include returning the size of the tree, creating an empty tree or clearing the contents of the tree, and so on.

The following algorithms are designed to capture the common parts in search, insertion and deletion operations.

- **Search:** Recursively descend all possible paths in the tree whose keys match with the search key.
- **Insertion:** An entry of a new key, with or without a new data object, is added to the leaf level. A leaf that overflows is split, and splits propagate up the tree.

- **Deletion:** Remove an entry from its leaf node. If this causes underflow, adjust tree accordingly by updating key in parent nodes to preserve search tree properties.

2.3.3 Trees Extension

An efficient implementation of search trees is crucial for any database system. To support the growing set of applications, search trees must be extended for maximum flexibility. This requirement has motivated three major research approaches in extending search tree technology:

1. Specialized search trees: A large variety of search trees have been developed to solve specific problems. Among the best known of these trees are spatial search trees such as R-trees [GA84]. While some of these works have had significant impact on particular domains, the approach of developing domain-specific search trees in problematic. The efforts, required to implement and maintain such as data structures, are high. As new applications need to be supported, new tree structures have to be developed from scratch, requiring new implementations of the usual tree facilities for search, maintenance, concurrency control and recovery.
2. Search trees for extensible data types: As an alternative to develop new data structures, existing data structures such B+-tree and R-tree can be made extensible in the data types they support. For example, B+-tree can be used to index any data with a linear ordering, supporting equality or linear range queries over that data. While this provides extensibility in the data that can be indexed, it does not extend the set of queries that can benefit from the trees, which are containing

equality and linear range predicates. Similarly in an R-tree, the only queries that can use the tree are those containing equality, overlap and containment predicates. This inflexibility presents significant problems for new applications, since traditional queries on linear orderings and spatial location are unlikely to be pertinent for new data types.

3. GiST [HNP95] gives us the third direction for extending search tree technology. GiST is an index structure supporting an extensible set of queries and data types. The GiST allows new data types to be indexed in a manner supporting queries natural to the types. In a single data structure, GiST provides all the basic search tree logic required by a database system, thereby unifying disparate structures such as B+-trees, R-trees, and RD-trees in a single piece of code, and opening the application of search trees to general extensibility.

2.4 Frameworks

As a maximum reuse technique at many levels, object-oriented frameworks are a very active issue for both the software industry and academia.

2.4.1 What is a Framework?

A framework [FS99] is an abstracted collection of classes, interfaces and patterns dedicated to solving a class of problems through a flexible and extensible architecture. Frameworks encapsulate critical design architectures specific to their purposes. In doing this, classes of frameworks can be utilized by developers to save time otherwise wasted reinventing common application problems.

Frameworks have been defined in many ways:

- "A framework is a collection of classes that provide a set of services for a particular domain; a framework thus exports a number of individual classes and mechanisms which clients can use or adapt" – Grady Booch, Object-Oriented Analysis and Design [BG93].
- "A framework is a set of prefabricated software building blocks that programmers can use, extend, or customize for specific computing solutions." ---Taligent, Building Object-Oriented Frameworks [TW94].
- "A framework is more than a class hierarchy. It is a class hierarchy plus a model of interaction among the objects instantiated from the framework" – Ted Lewis, Object-Oriented Application Frameworks [LRP95]

Framework technology has been studied for more than a decade. Frameworks have proven to improve productivity owing to their reusability. Frameworks can be reused in many levels such as domain knowledge, analysis, architecture, design and code. Unlike earlier reuse techniques based on class libraries, frameworks are domain specific. Frameworks are an object-oriented reuse technology. Normally, there are three kinds of frameworks [FS97]:

- ***System infrastructure frameworks*** -- These frameworks simplify the development of portable and efficient system infrastructure such as operating system and communication frameworks, and frameworks for user interfaces and language processing tools. System infrastructure frameworks are primarily used internally within a software organization and are not sold to customers directly.
- ***Middleware integration frameworks*** -- These frameworks are commonly used to integrate distributed applications and components. Middleware integration

frameworks are designed to enhance the ability of software developers to modularize, reuse, and extend their software infrastructure to work seamlessly in a distributed environment. Common examples include ORB frameworks, message-oriented middleware, and transactional databases.

- ***Enterprise application frameworks*** -- These frameworks address broad application domains (such as telecommunications, avionics, manufacturing, and financial engineering) and are the cornerstone of enterprise business activities. Relative to System infrastructure and Middleware integration frameworks, Enterprise frameworks are expensive to develop and/or purchase. However, Enterprise frameworks can provide a substantial return on investment since they support the development of end-user applications and products directly. In contrast, System infrastructure and Middleware integration frameworks focus largely on internal software development concerns. Although these frameworks are essential to rapidly create high quality software, they typically do not generate substantial revenue for large enterprises. As a result, it's often more cost effective to buy System infrastructure and Middleware integration frameworks rather than build them in-house.

2.4.2 Whitebox Frameworks and Blackbox Frameworks

Frameworks can also be classified by the techniques used to extend them, which range along a continuum from whitebox frameworks to blackbox frameworks.

Whitebox frameworks rely heavily on OO language features like inheritance and dynamic binding to achieve extension. Existing functionality is reused and extended by:

1) Inheriting from framework base classes

2) Overriding pre-defined hook methods using patterns like Template Method.

Blackbox frameworks support extensibility by defining interfaces for components that can be plugged into the framework via object composition. Existing functionality is reused by

1) Defining components that conform to a particular interface
2) Integrating these components into the framework using patterns like Strategy and Functor.

Whitebox frameworks require application developers to have intimate knowledge of the frameworks' internal structure. Although whitebox frameworks are widely used, they tend to produce systems that are tightly coupled to the specific details of the framework's inheritance hierarchies. In contrast, blackbox frameworks are structured using object composition and delegation more than inheritance.

As a result, blackbox frameworks are generally easier to use and extend than whitebox frameworks. However, blackbox frameworks are more difficult to develop since they require framework developers to define interfaces and hooks that anticipate a wider range of potential use-cases.

2.4.3 Common Terminologies in Frameworks

The following are three common terminologies in frameworks.

- Hot spots-----Hot spots [WP94] are the general areas of variability within a framework where placing hooks are beneficial. A hot spot may have many hooks within.

- Hooks-----Hooks are the places in a framework that can be adapted or extended to provide application specific functionality. A hook can be adapted in some way such as by filling in parameters or creating subclasses. Each hook description documents a problem or requirement that the framework builder anticipates an application developer will have, and provides guidance about how to use the hook and fulfill the requirement. They are the means by which frameworks provide the extensibility to build many different applications within a domain [FH97].
- Frozen spots-------Frozen spots [WP94] within the framework capture the commonalities across applications, in contrast to hot spots. They are fully implemented within the framework and leave no work to the framework user. Typically there are no hooks associated with them.

2.5 Design Pattern

A pattern is a way of doing something, or a way of pursuing intent. Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [HNP95] .

Design Patterns capture important relationships between abstract kinds of objects that can collectively be applied to accommodate a particular problem in object-oriented design. Experienced software developers are often confronted with the same types of problems in different projects. Applying a known solution to these problems makes them easier to solve and promotes reliability in their application. By naming and cataloging these patterns, developers have a common library of solutions from which to choose.

Patterns have three distinguishing characteristics:

- Patterns are reusable. A Pattern's only purpose is to be reused. And because a pattern is implementation-independent, it is suitable for inclusion in a framework. Patters typically provide micro-architectural solutions to common recurring problems. Many of these problems may be embedded in a framework, which addresses a larger-scale domain problem.
- Patterns are context-free. The context, or surrounding implementation, in which a pattern can be applied, is generally not a constraint. Given a specific type of problem, if a pattern exists to addresses that type of problem, it can be applied generally, meaning that the application specific objects or components of the pattern are supplied by the application developer and the behavior mechanics are dictated by the pattern.
- Patterns have granularity. Design patterns are fundamental structures and are not intended to scale into larger domain aspect types of problems. They represent useful reusable micro-architectures that developers can apply to a given problem of an equally minute granularity. This is not to suggest that design patterns do not exist at many levels within a complex system. Actually, a variety of patterns can be extracted during the development of large applications and then reapplied in similar situations. It would seem that the reusability of aggregated design patterns diminishes as the complexity and specificity of the pattern rises.

The Gang of Four [GHJ95] defines a pattern as having four essential traits, used to distinguish the patterns from one another. The four traits are:

- Pattern name --- the pattern name is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. Having a vocabulary for patterns lets us talk about them with our colleagues, in our documentation, and even to ourselves. It makes it easier to think about designs and to communicate them and their trade-offs to others. Finding good names has been one of the hardest parts of developing our catalog.
- Problem --- problem describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.
- Solution --- solution describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.
- Consequences --- consequences are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language

and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.

We simply introduce to the following design patterns associated with our design and implementation of the search tree index framework.

2.5.1 Composite Design Pattern

The intent of the composite pattern [GHJ95] is to compose objects into tree structures to represent part-whole hierarchies. The Composite Design pattern allows a client object to treat both single components and collections of components identically.

In Figure 2.10, create an abstract Component class to define the interface of all tree objects and implements default common behavior. A concrete Composite class implements behavior for storing and accessing child Component objects.

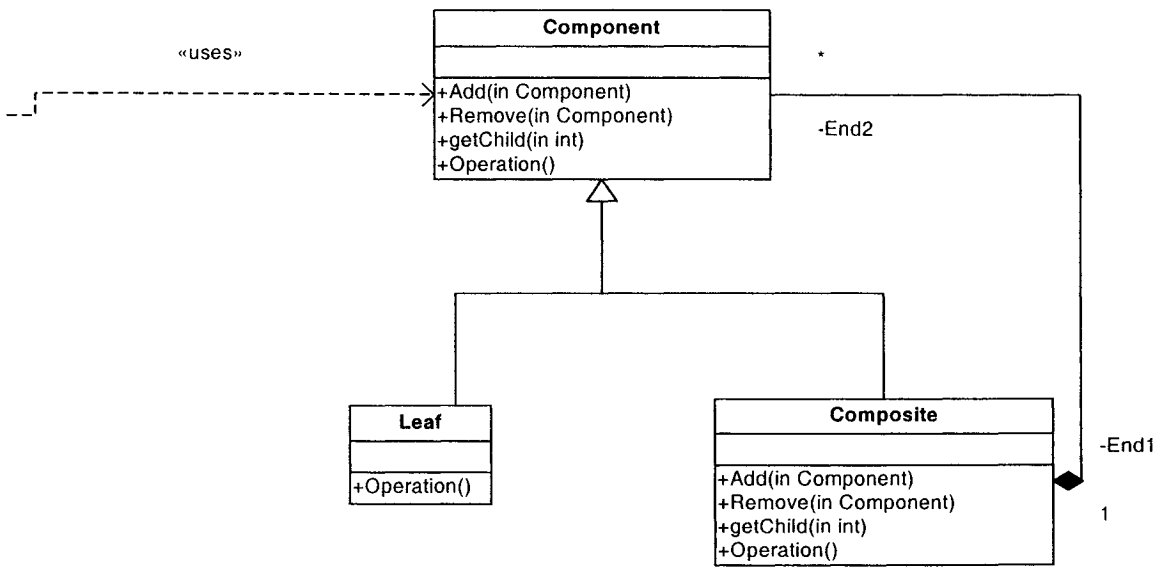

Figure 2.10: UML of Composite Design Pattern

The Composite design pattern enables the client to treat composite structures as well as leaf structures as if they are the same thing. As a consequence this simplifies the client and makes it easier to add new types of components, newly defined Composite of Leaf subclasses will work automatically with current structures.

2.5.2 Proxy Design Pattern

The intent of the Proxy design pattern [GHJ95] is to provide a surrogate or placeholder to control access to an object as shown in Figure 2.11 and Figure 2.12

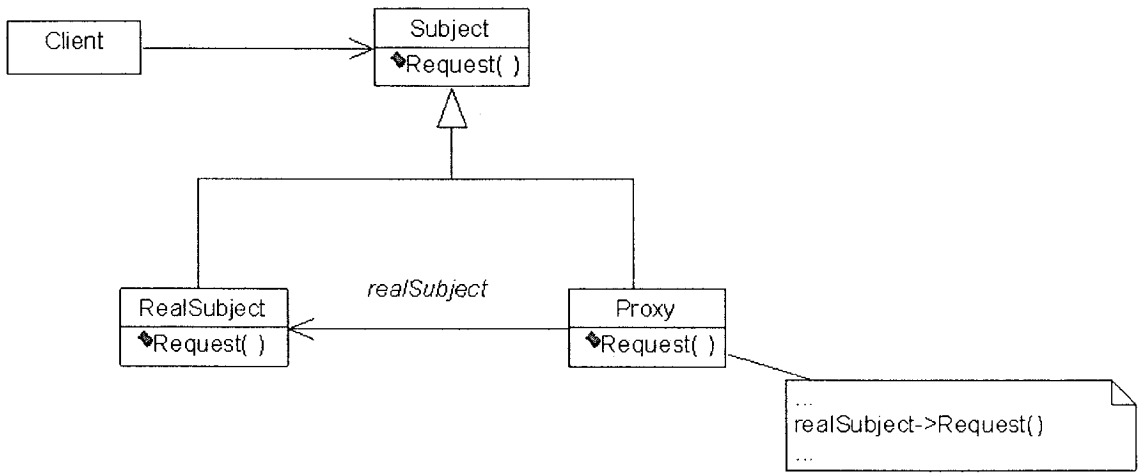

Figure 2.11: UML of Proxy Design Pattern

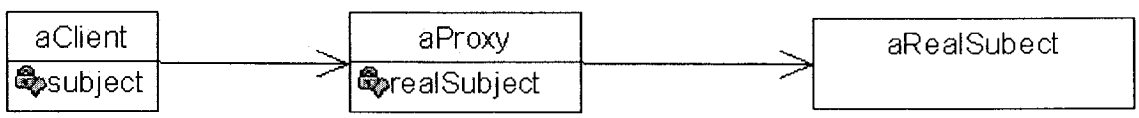

Figure 2.12: A Possible Instance Diagram of a Proxy Structure at Run-time

In the Proxy design pattern, real object and proxy implement the same interface, so that Proxy can handle some or all requests to the real object. Proxies provide a level of indirection to specific properties of objects, so they can restrict, enhance or alter these properties. Proxy is applicable whenever there is a need for a versatile or sophisticated reference to an object than a simple pointer. A proxy can be used in many ways:

- A remote proxy provides a local representative for an object in a different address space.
- A virtual proxy creates expensive objects on demand.
- A cache proxy can save resources by storing results temporary. When the client uses the same expensive operation of an object, the proxy supplies the result without calling the real subject's method.
- A count proxy can perform additional operation such logging counting before and after calling the real subject.
- A protection proxy controls access to the real object. Protection proxies are useful when objects should have different access rights.

A smart reference replaces a bare pointer that performs additional actions when object is accessed. Typical uses include counting the number of references to real object so that it can be freed automatically when are no more references, also called smart pointers, loading a persistent object into memory when it's first referenced, and checking that the real object is locked before it's accessed to ensure that no other object can change it.

A smart pointer [AA01] is object that look and feel like pointer, but is smarter. It is an application of Proxy design pattern. We implemented a smart pointer.

To look and feel like raw pointer, smart pointers need to have the same interface that pointers do: they need to support pointer operations like dereferencing (operator *) and indirection (operator ->). To be smarter than regular pointers, smart pointers need to do things that regular pointers do not. Probably the most common bugs in C++/C are related to pointers and memory management: memory leaks, allocation failures, loading a persistent object transparently, locking and others.

2.5.3 Singleton Design Pattern

The intent of the Singleton design pattern [GHJ95] is to ensure a class has only one instance and provide a global point of access.

Singleton design patterns help structure code. It hides the operation that creates the instance behind a static member function. This member function, traditionally called Instance (), returns a pointer to the sole instance. Clients access the singleton by calling the static instance function to get a reference to the single instance and then using it to call other methods.

2.5.4 Abstract Factory Design Pattern

The intent of the Abstract Factory design pattern [GHJ95] is to provide an interface an interface for creating a family of related or dependent polymorphic objects.

Abstract Factories can be an important architectural component because they ensure that the right concrete objects are created throughout a system. The most typical form of Abstract Factory is an interface that declares a protocol of factory methods as shown in Figure 2.13. One or more concrete classes may implement this interface to

create the family of classes that are appropriate for the application. This design pattern provides a pluggable architecture that allows the application programmer to easily replace a family of components that provide the same functionality.

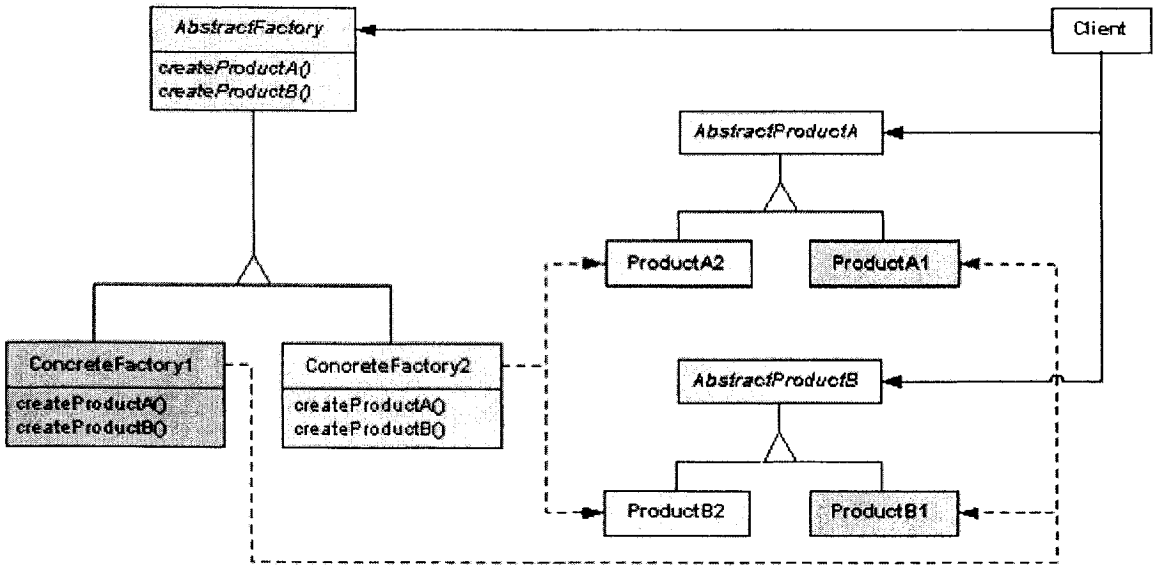

Figure 2.13: UML of Abstract Factory Design Pattern

2.5.5 Serializer Design Pattern

The intent of the Serializer design pattern [RSB98] is to read arbitrarily complex object structures from, and write them to, varying backends efficiently, such as flat files, relational databases, and RPC buffers.

The Serializer pattern, shown as Figure 2.14, can efficiently stream objects into data structures as well as create objects from such data structures. Figure 2.14 shows the structure of the Serializer Pattern. The Reader part of the pattern builds an object structure from a backend. The Writer part of the pattern writes an existing object structure as a data structure to a backend. Both parts together constitute the Serializer pattern.

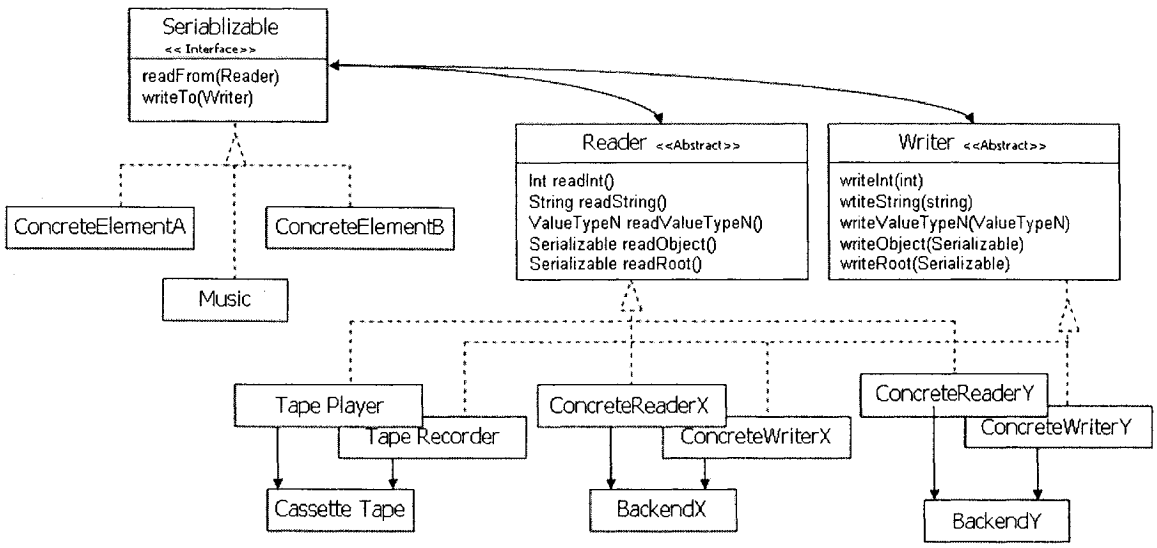

Figure 2.14: Structure of the Serializer Pattern

Serializer pattern makes application easy to add new data representation formats for object and to take knowledge about external data representation formats out of the objects to be streamed. Object structures can be written to and read from a new and unforeseen backend simply by introducing a new Reader/Writer pair, and by using the Read/Writer interface of simple read and write operations, so the objects are effectively shielded from any data format of their external representation.

2.6 Generic Programming Techniques

Generic programming is about representing domains as collections of highly general and abstract components, which can be combined in vast numbers of ways to yield very efficient concrete programs. The term generic programming has at least four different meanings:

- Programming with generic parameters.
- Programming by abstracting from concrete types.

- Programming with parameterized components
- Programming method based on finding the most abstract representation of efficient algorithms.

In C++, class and function templates are particularly effective mechanisms for generic programming because they make the generalization possible without sacrificing efficiency. By using templates, we can design a single class that operates on data of many types, instead of having to create a separate class for each type.

2.6.1 Template

Templates are functions or classes that are written for one or more types not yet specified. A class template allows the complier to generate multiple versions of a class type by using type parameters. A function template allows us to define a group of functions that are the same except for the types of one or more their arguments or objects. When a template is used, programmers pass the types as arguments, explicitly or implicitly. Because templates are language features, you have full support of type checking and scope. Templates also allow us to parameterize behavior, to optimize code, and to parameterize information.

The introduction of templates to C++ added a facility whereby the compiler can act as an interpreter. This makes it possible to write programs in a subset of C++, which are interpreted at compile time. Language features such as for loops and if statements can be replaced by template specialization and recursion. These programs do not have to be executed -- they generated their output at compile time as warning messages. For example, one program generated warning messages containing prime numbers when compiled.

2.6.2 Template and Code Reuse Techniques

There are some of the generic programming techniques used in our design.

- Traits: a traits class provides a way of associating information with a compile-time entity (a type, integral constant, or address).
- Adaptors: an adaptor is a class template that builds on another type or types to provide a new interface or behavioral variant. Example of standard adaptor is *std::reverse_iterator.*
- Type Generators: a type generator is a template whose only purpose is to synthesize a new type or types based on its template argument(s). The generated type is usually expressed as a nested *typedef* named, appropriate type. A type generator is usually used to consolidate a complicated type expression into a simple one.
- Object Generators: an object generator is a function template whose only purpose is to construct a new object out of its arguments. Think of it as a kind of generic constructor. An object generator may be more useful than a plain constructor when the exact type to be generated is difficult or impossible to express and the result of the generator can be passed directly to a function rather than stored in a variable.
- Policy Classes: policy classes are implementations of punctual design choices. They are inherited from, or contained within, other classes. They provide different strategies under the same syntactic interface. A class using policies is templated having one template parameter for each policy it uses. This allows the user to

select the policies needed. The power of policy classes comes from their ability to combine freely. By combining several policy classes in a template class with multiple parameters, one achieves combinatorial behaviors with a linear amount of code.

2.6.3 Standard Template Library (STL)

The Standard Template Library (STL) [SL95] is a general-purpose C++ library of algorithms and data structures, originated by Alexander Stepanov and Meng Lee. The STL based on generic programming, is part of the standard ANSI/ISO C++ library.

The STL is implemented by means of the C++ template mechanism and provides a powerful set of components for generic programming. What this means in layman terms is that the STL contains a standardized library of data structures that have already been developed and debugged. The C++ programmer does not need to create his own doubly linked-list and then debug it prior to use, but may utilize the STL List container (data structure). Additionally, the STL provides virtually all container functions and algorithms that one would need in manipulating the data in the STL data structure. Using the STL, the burden of repeatedly creating and supporting needed data structures is lifted from the shoulders of the programmer.

Another advantage of STL provided data structures is that they are optimized and the associated container methods and algorithms have performance guarantees. Compiler manufacturers are constrained to meet STL performance requirements established by ANSI standards. The programmer is afforded the confidence to choose an STL container

not only based on its structure but also based on the performance characteristics that his/her program requires.

The Standard Template Library is also generic and modular. Because the components that make up the STL use C++ template functions and classes, generic programming is possible. STL containers and algorithms support all C++ data types as well as user defined data types and structures. When using an STL data structure and associated algorithm, the programmer does not need to worry about the type of data they are inserting into the containers. Likewise, converting an STL supported C++ program from one data type to another can be a very trivial task.

The Standard Template Library is made up of six basic components: Containers, iterators, Generic Algorithms, Function Objects, Adaptors, and Allocators. (Namespaces are a recent addition to the ANSI C++ standard, and though not technically an STL component, they are used by the STL.) Each of theses components, in turn, is made up of several smaller components or structures. The STL Container component, for example, is made up of seven different data structures: Vectors, Deques, Lists, Sets, Multi-sets, Maps, and Multi-maps; the Generic Algorithm component contains over eighty different algorithms.

As one might expect in a modular system, all the components and their associated sub-components or structures fit together in a modular way. The Standard Template Library allows the programmer to build data structures and functionality within a program by "plugging together" STL components.

Chapter 3 Tree Index Framework Design

This chapter discusses the conceptual and architecture design of general search tree access methods.

3.1 Use Cases

Use cases capture the functionality of a system. A use case defines a goal-oriented set of interactions between external actors and the system under consideration.

We consider the index as a part of the database with the following properties:

- Built as an auxiliary data structure intended to help speed up the retrieval of records.
- Be customized to suit different applications.
- Be used to look up data in the database (read-only access).
- Be used to insert or delete data in the database (read-write access).

Figure 3.1 shows the detailed use cases of index framework. The diagram presents the main actors of an index system.

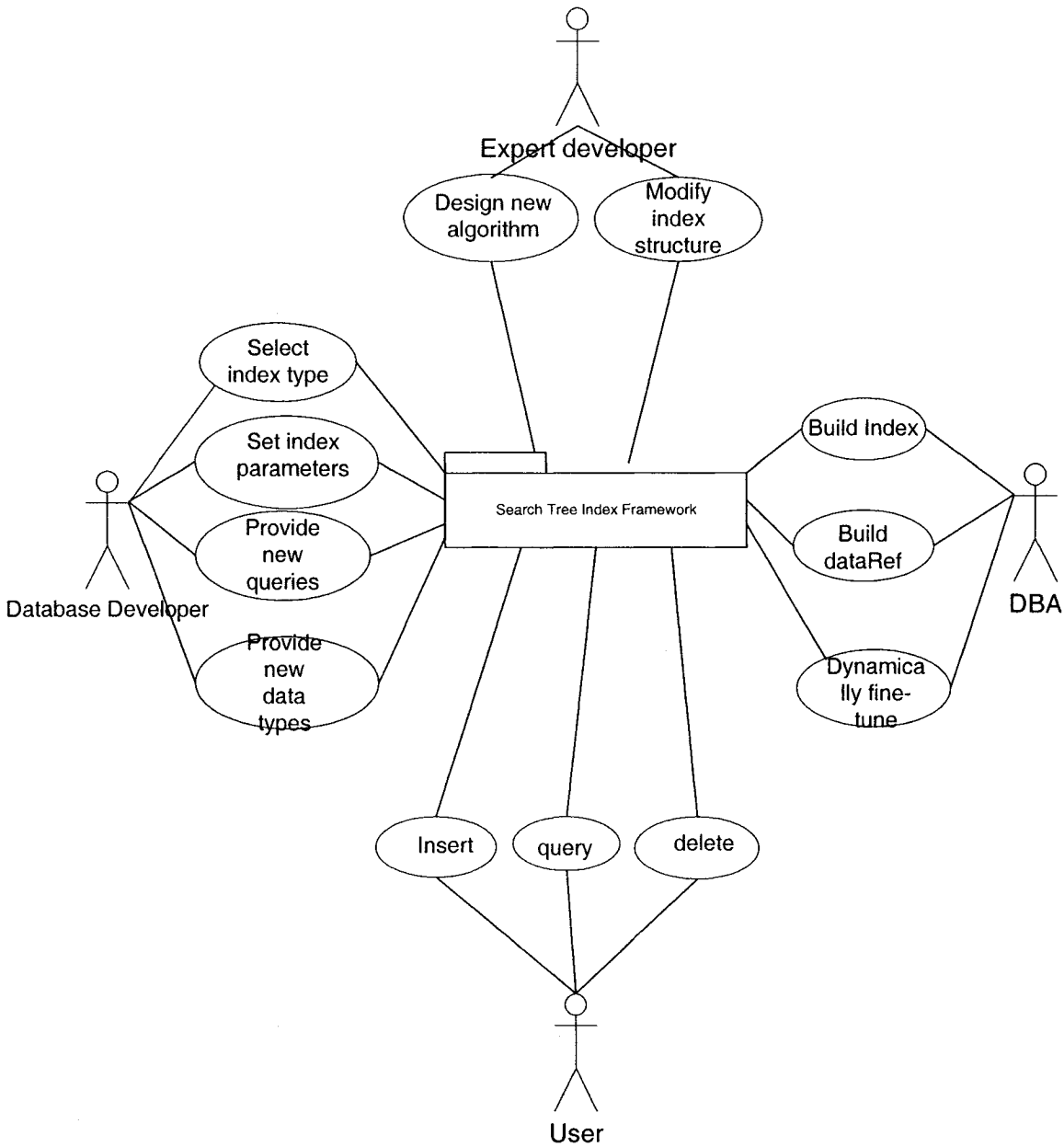

Figure 3.1: Use Case of Index System

Expert Developer

The expert developer is an experienced programmer who designs the database system to satisfy experts' needs in certain domains and then implements the design using a programming language.

The expert developer should put a complete design for a database system not only to meet the functional need of database domain, such as organizing new index structures and algorithms to solve some special problems or defining new methods to traverse the index by replacing some parts of an existing index in order to produce a new index that is using a new access, but also to satisfy the nonfunctional needs like storage, retrieval issues and platform mounting.

Database Developer

The database developer is a knowledge domain expert, responsible for choosing an existing index access method to satisfy some domain requirements.

The database developer is responsible for setting the index parameters. Examples include trees order, the page fill factor, page size to suit the system platform and some application variables.

The database developer should customize an existing database system by modifying some of its parts (components) or by replacing some parts with others to be able to support new data or query types.

Database administrator

The database administrator is the person responsible for building the database system, which includes building schemes, physical tables and indexes used to access them.

After the database system is set up and running, the DBA needs to dynamically fine-tune it by adjusting its parameters to achieve the optimal performance under typical workloads.

User

The user is the software that is using the system to search for, insert or delete data from the database.

3.2 Index and Database Data

Each index is built from the physical data indirectly, through a data reference file call index file. The index file is composed of key and data reference pairs, and constitutes the data level of each index. This separation between physical data and data references allows use to build multiple indexes on the same data set. Notice that data is not included in the index and the index does not directly provide data, but it provides information about where the data is. This means that data stored in the index is different from the typical database information, such as tables, records etc.

3.3 The Conception Design of Index Framework

The Standard Template Library [SL95] is a recent addition to the C++ language (1998) that supports good programming practices and provides a wealth of building blocks that can satisfy the needs of many sophisticated modern systems. STL's components are interoperable and extensible. All STL components are written to conform to precisely specified requirements. Abstract concepts, algorithms and data structures in terms of abstract concepts are the essence of generic programming. Gaffar [GA01]

designed a STL style B+-tree structure in his thesis. We will use his concept design to our framework.

The Know-It-All index framework applies the STL modularity concept in analysis, architecture, design and interface, which is used for primary-memory data structure libraries both to classify functionality and reuse implementation. We attempt to start a similar taxonomy but for search tree access methods manipulating data on secondary storage.

The adoption of STL approach promotes code reuse, increases readability and user friendliness, and reduces time and money overheads incurred during the application development process. Like STL, our index framework is designed as a STL container so that each component of the index framework belongs to a group such as container group, iterator group, and so on. These different groups are designed to carry out a different part of a system so that in the end we can have a complete system from these groups. Moreover, all search tree algorithm functions are unified to the same interface that acts on containers through iterator. For example, the all components of index framework, such as tree, page, index page, leaf page, cache, cursor shown in Figure 3.2 conform to a set of abstracts container concepts provide by STL. All searching functions have the same interface *find()* and *find_if()*. Figure 3.3 shows a general interface of index framework for database developer, which is very similar to STL container.

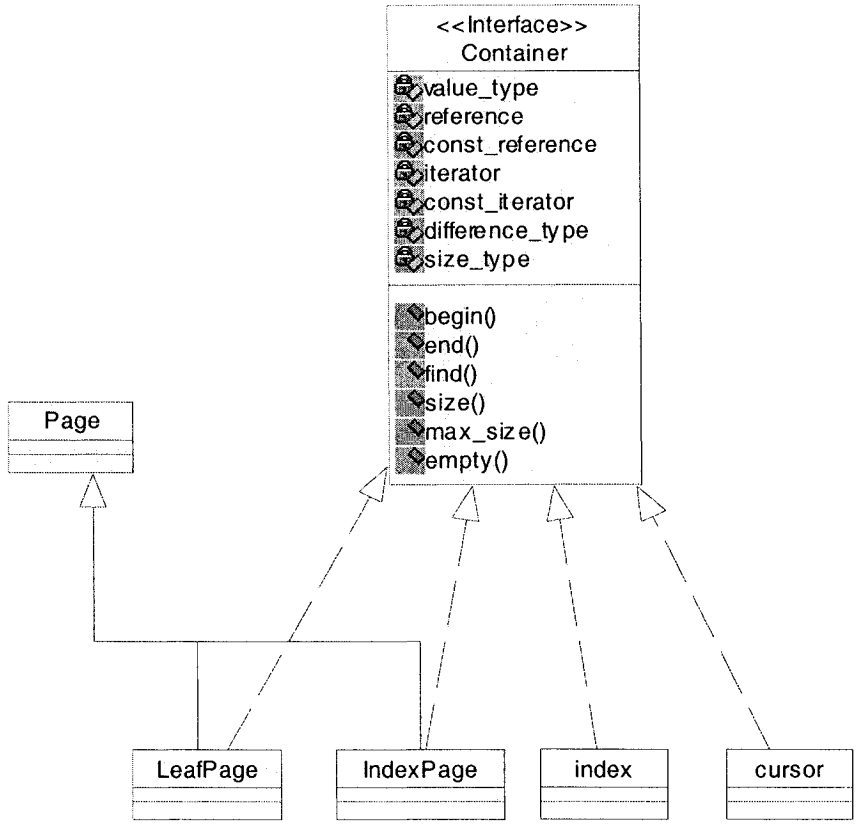

Figure 3.2: The Basic Interfaces of the Containers

The framework also has a flexible design by adopting a complete building block replacement policy. The design can be adapted to the needs of any application by simply changing some of the building blocks. This was made easier since the building blocks are highly decoupled. The interface of these blocks is carefully designed to seamlessly connect to other blocks of the same or a different group. It means that when the existing access methods do not cover the domain, the users can add any specialized access methods with the same interface to replace an existing one. This allows a minimal impact on the system since the user-defined block will seamlessly integrate into the other blocks, so it eliminates the need to do a completely new system.

```
template <
            class DataRef,
            class PageType,
            class IndexKeyType,
            class LeafKeyType,
            class IndexPageType,
            class LeafPageType,
            class Predictor,
            class CacheType,
            class GiSTExtension,
            template< class, class, class, class, class, class, class,class > class SearchTree
           >
class SearchTreeIndex: public SearchTree < DataRef, PageKeyType, PageType,
                    LeafPageType, IndexPageType, Predictor, CacheType, GiSTExtension>
{
private:
          ......
public:
          typedef SearchTree  IndexType;
          typedef SearchTree::value_type value_type;
          typedef SearchTree::key_type key_type);
.....................

          iterator insert(const value_type& aPair) {IndexType::insert(aPair); };
          iterator find(const key_type& key, Predictor_type predictor)
                    {IndexType::find(key,predictor);};
          iterator find(const key_type& key, Cursor* cur, Predictor_type predictor)  {
                              IndexType::find(key,cur,predictor);};
          iterator delete(const key_type& key){IndexType::delete(key);}
.....
}
```

Figure 3.3: The General Interface of Index Framework

3.4 Transient Data and Persistent data

In general, when database programmers develop a new access method, they face the problem of dealing with two different views of structured data: transient data and persistent data. Transient data is in primary storage and ceases to exist after the creating process terminates, whereas persistent data is stored in secondary storage and outlives the programs that create and manipulate it. In common, traditional programming languages provide facilities for the manipulation of transient data. For example, data structures in

primary storage are usually organized using pointers, which are used directly by the processor's instructions. However, it is generally impossible to store and retrieve data structures containing pointers to/from disk without converting at best the pointers and at worst the entire data structure into a different format.

Our framework provides a uniform view of pointer, smart pointer [AA01], for efficiently and transparently constructing low-level access methods between primary and secondary storage. It unifies the transient data and persistent data. The smart pointer eliminates the work of converting structured data between primary and secondary storage. Using smart pointer, data on secondary storage is accessible in the same way as data in primary storage so that building powerful and flexible data structure on secondary storage directly is convenient to programmer. A smart pointer looks like a normal pointer that can be operated directly to secondary storage. Memory container can subsequently retrieve and manipulate persistent data by only replacing raw points with smart pointers without having to modify the code that manipulates them. Data structures for an index file, such as a B+-tree, are significantly simpler to build, test, and maintain than traditional file structures. Hence, using smart pointer, developing a new search tree access is becoming very easily, as simple as developing a STL container.

3.5 Architecture of Index Framework

Layering provides a design technique to break down a complex system by decomposing functionality. It reflects the division and conquest principle and helps to minimize subsystem dependencies. Our system is designed as an integrated set of layers including Container Presentation layer, Proxy layer and Physical Storage layer. These layers provide all the necessary parts needed for implementing a complete index

framework system. Each of these layers, referred to as a building block, is well defined and has a clear set of functionality. A lower layer provides services to the layer above it. A layer has no knowledge about other layers except the layer that provides services to it. A layer can be easily understood in isolation and substituted easily.

A Search Tree Index Framework can be broken down into three layers: Container Presentation Layer, Proxy Layer and Physical Layer, Figure 3.4 shows its architecture.

- Container Presentation is a subsystem that provides services to the end user. It is used to represent any specialized tree as a STL style container. Examples are those data structure based on the hierarchical approach such as B-tree, B*-tree, R-tree, R+-tree, SS-tree and so on. There are two abstraction packages in this subsystem. B-tree container package is for linearly ordered search tree such as B-tree, B+-tree application, and R-tree container package is for spatial search tree such as R-tree, R*-tree, SS-tree, SR-tree. Container layer is not concerned about memory management, physical storage management in the secondary storage of index search tree and access control between physical storage and memory, but organization techniques or data structure, and leave these functions to other lower layers.
- Proxy layer is a subsystem that is responsible for maintaining a repository for search tree page. It handles object caching, memory management and access control page objects between memory and physical storage.
- Physical layer is a subsystem that is mainly responsible for managing and controlling persistent data of search tree.

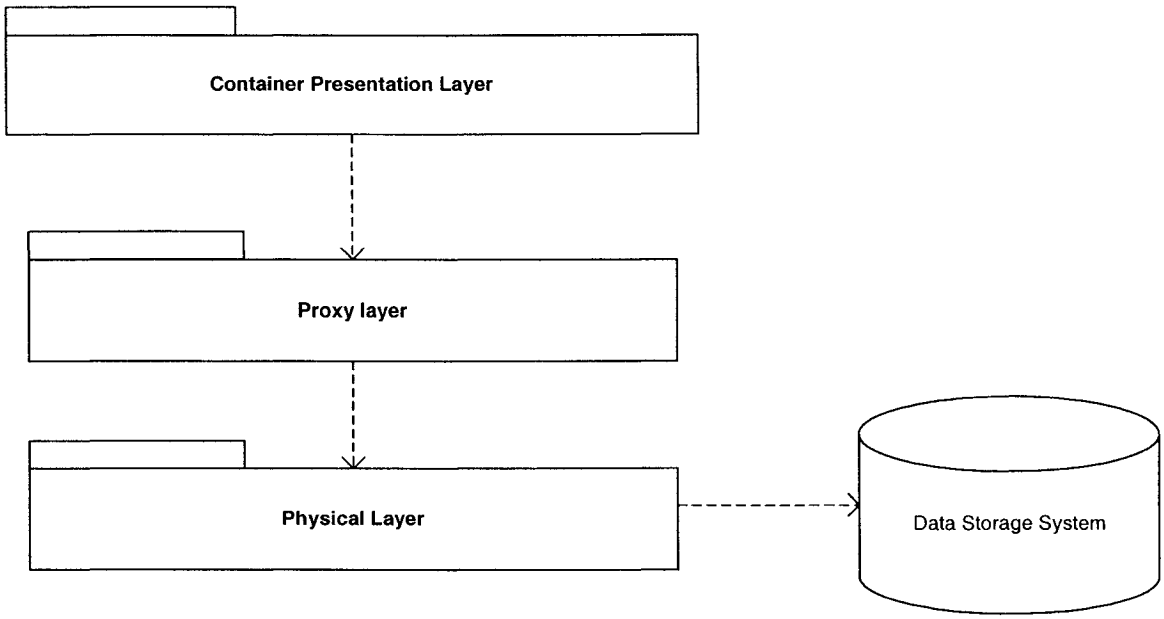

Figure 3.4: Layered Architecture for Search Tree Index Framework

The architecture that we presented above gives expert developer and database developer the freedom to isolate and replace any one or more of these components with no or minimal impact on the system and can be adapted to the needs of any application easily, when they develop a new access method.

We use template parameters to connect with each layer; lower layers are passed to the upper as template type in the implementation phase. For example, for cache layer, storage layer is some class parameters of cache class. As well, parameterization design allows for code reuse by making the same layer code usable many times and increase readability, which greatly improves reliability and reduces development cost. For example, cache layer and storage layer can be implemented and tested once and can be used to any tree containers many times.

3.5.1 Container Presentation Layer

The Container Presentation layer refers to some Search Tree STL containers, which implement some specialized index access methods to satisfy some domain requirements. There are some issues related to how to build a search tree container in Container Presentation Layer.

The Search Tree Presentation Issue

From the survey of search trees, it is possible to capture the essential nature of a database search tree: it is a hierarchy of categorizations, in which each categorization holds for all data stored under it in the hierarchy. They have certain common operations such as deletion, insertion and search. Based on this idea, any specialized trees can be represented as STL style containers.

A search tree access method can be implemented by plugging STL tree container into the framework. The only way to interact with the container is through its iterator. Iterators provide access to the elements of these data structures without having to use or access the particular data structure's implementation. Iterators can be used to perform general queries on search tree structures, while providing a clean interface for these queries. The basic component index and leaf page are also designed as containers. They will be retrieved from the hard disk into memory through a proxy provided by the Proxy layer. If users use STL standard operation functions: *insert*(), *erase*() or *replace*() to index entry in the containers, changes can be automatically committed to the persistent data. Figure 3.5 shows the perspective of general search tree container.

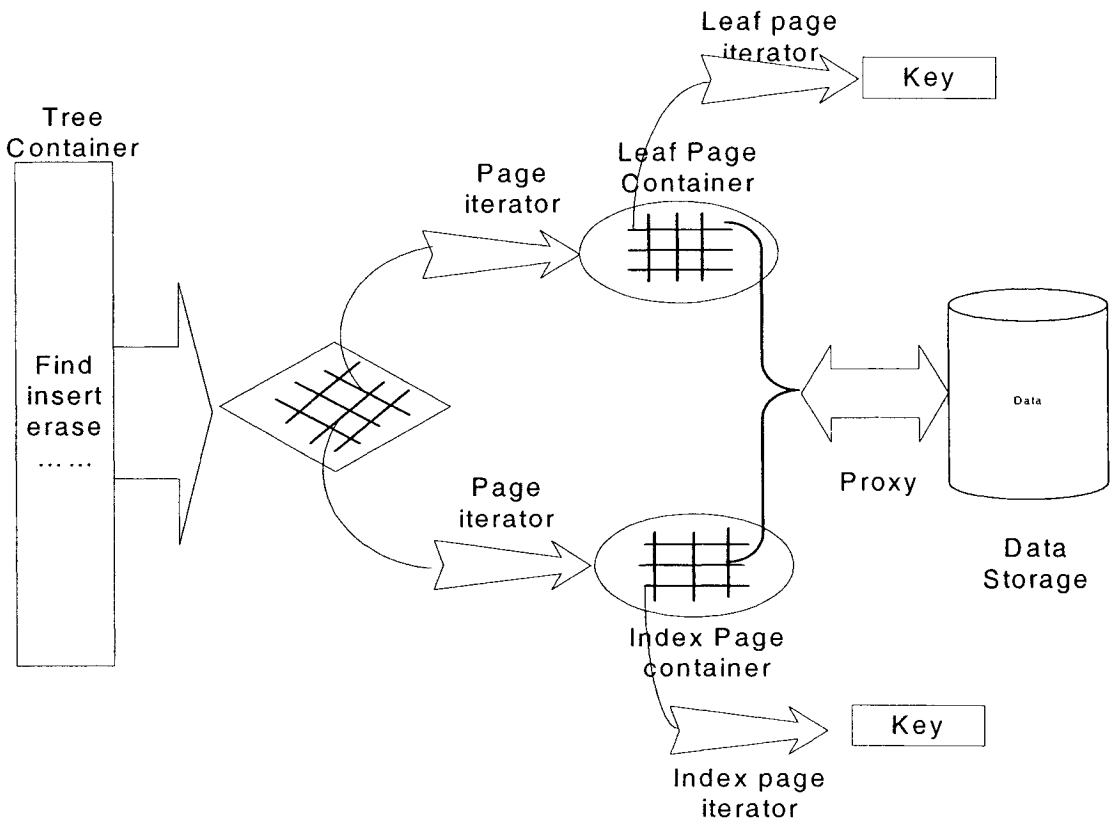

Figure 3.5: Perspective of General Search Tree Container

Extension Architecture Issue

The Container Presentation layer refers to some STL tree containers that provide interactive interfaces to end-users and main access method functions to build a database system. The tree containers may cover one-dimensional tree structures such as B+-tree, point access structures, such as K-D-B-Tree, hB-Tree, spatial multi-dimensional structures, such as R-tree, R*-tree, SS-tree, X-tree, also may fit sequential queries, exact match queries, range queries, approximate queries, and similarity queries. However, designing a special search tree is still a complex work, because developers of search tree access method require a very good understanding of search tree structure, search strategy, concurrency and recovery protocols.

To make the search tree assess method extension as easily as possible, a STL container of generalization of database search trees is designed based on the general notion of search keys and the essential nature of tree structures as observed in the GiST paper [HNP95] . It has two abstraction tree container packages, shown as Figure 3.6, that captures the basic search tree operations: B-tree container package that for extending linearly ordered search trees such as B-tree, B+-tree application, and R-tree container that for extending spatial search trees such as R-tree, R*-tree, SS-tree, SR-tree. Two base packages provide the common parts of the tree operation algorithms and require a programmer to give specific tree behavior routines to complete the operations of individual search tree.

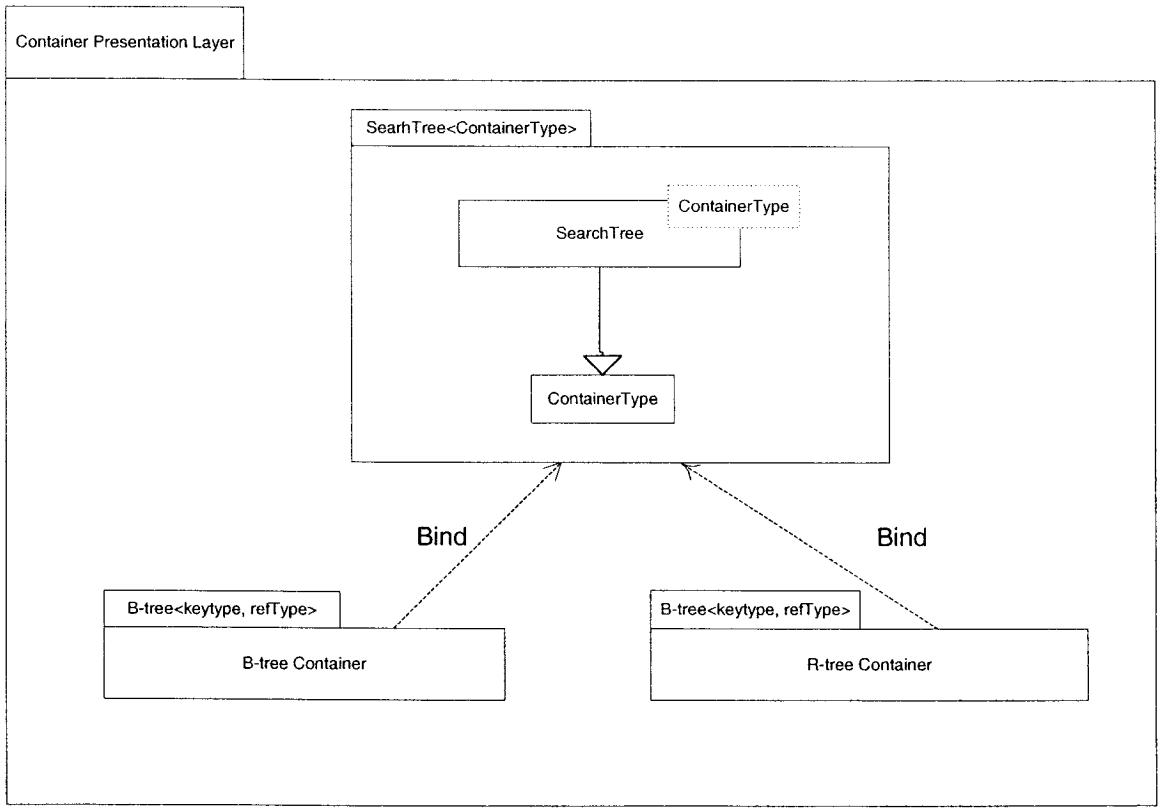

Figure 3.6: Architecture for Container Presentation Layer

We use multi-policy, a generic programming technique, to design tree container class. Two base tree containers are designed as two policy classes that are implementations of search tree functional design choices. They are not intended for standalone use; instead, they are used as a parameter to be plugged into generalized search tree container.

Database developers can choose the kind and the number of containers and the specific algorithms for particular operations, as well customize data types or query types of container to satisfy the domain requirements.

The Container Structure Issue

From the survey of generalization of search trees in the section 2.3, we know that a search tree can be designed as a STL container with certain common operations such as deletion, insertion and search. When programmers implement a certain access method, some particular behaviors and algorithms of a search tree can be plugged into the container by using the generic programming techniques with the STL iterator and container standards. Figure 3.7 shows the class diagram of B-tree or R-tree container package in our design. B-tree package and R-tree package have the same structures but different algorithms. Because tree, IndexPage and LeafPage classes are designed to be STL-like containers, they must follow STL style. To handle heterogeneous collections of page objects and make our implementation safer and more elegant, a page class is added as a base class of IndexPage and LeafPage. In practical application on non-volatile storage, the container will use a proxy (smart pointer) that takes responsibility of retrieving the page from the storage into memory for access.

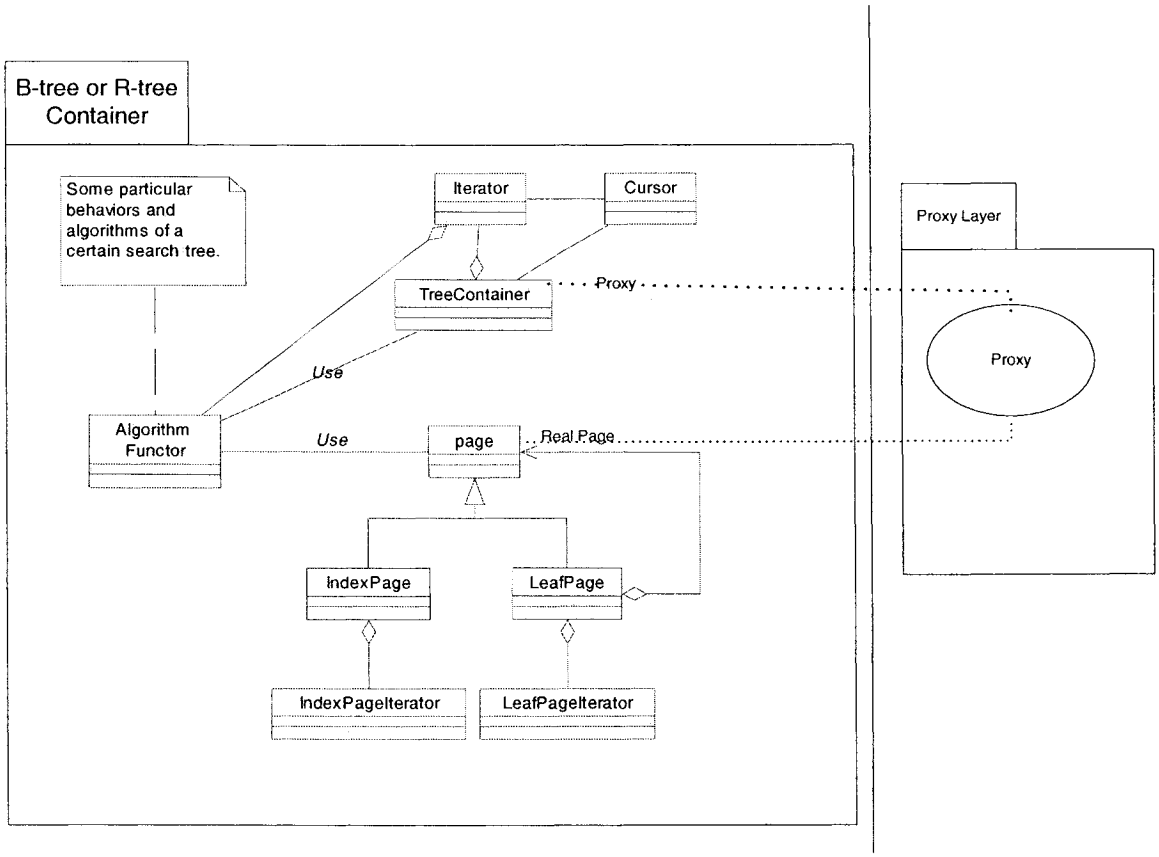

Figure 3.7: Class Diagram of R-tree or B-tree Package

Cursor and Iterator Issue

A cursor object represents a set of pairs of index key and reference for a specified index search. Our cursor design has a small difference from the traditional approach. To make our system more compatible, the cursor is designed as a leaf page container that contains the leaf pages, which may have the searching results satisfied with predicate algorithms.

The cursor class provides a common interface with search tree iterator for iterating through containers. In this way, the internal structure and implementation of a container becomes transparent.

The index is designed to be a container providing an iterator to its contents. The only way to interact with the container is through its iterator. An iterator is an abstraction of the notion of a pointer to an element of a container, such as a vector, an array and a linked list or a tree [Str97]. Iterators provide access to the elements of these data structures without having to use or access the particular data structure's implementation. Iterators can be used to perform general queries on search tree structures, while providing a clean interface for these queries. For example, iterators providing range queries on data in an ordered domain, such as a B+-tree, can include the following:

- Retrieve all records (sequential scan);
- Sequential scan from Key1 to Key2;
- Sequential scan from Key for count amount;
- Sequential scan for count amount before Key and count amount after;

Examples of iterators for window queries such as a R-tree, with a similar interface as that for range queries, include the following:

- Overlap: given an object O, find all objects with at least one point in common with O;
- Equal: given an object O, find all objects equals O;
- Containment: given an object O, find all objects enclosed by O;
- Adjacency: given an object O, find all objects adjacent to O;

All these query result can be traversed through iterator operator ++ () or operator -- (). And operator * () can de-reference the result and return to the pair of key and reference that iterator is pointing to.

Figure 3.8 shows the activity of search progress through cursor and iterator. The cursor is a container of leaf pages that may contain the search results satisfied with predicate algorithms. When a search tree calls the *find()* function, it builds a cursor object, and calls predicate algorithms to find and save all leaf pages that may contain the result. Then, iterator of trees would consider the first page of cursor as current leaf page, and find the first result through it using its iterator of leaf pages and predicate algorithms return the result. If users call the ++ operator of tree iterator, the tree iterator will go through the current leaf page and find the next result. If the iterator meets the end of the current leaf page, it will continue to the next leaf page. When the tree iterator meets the end of the last leaf page in the cursor container, it will return the end flag as the result.

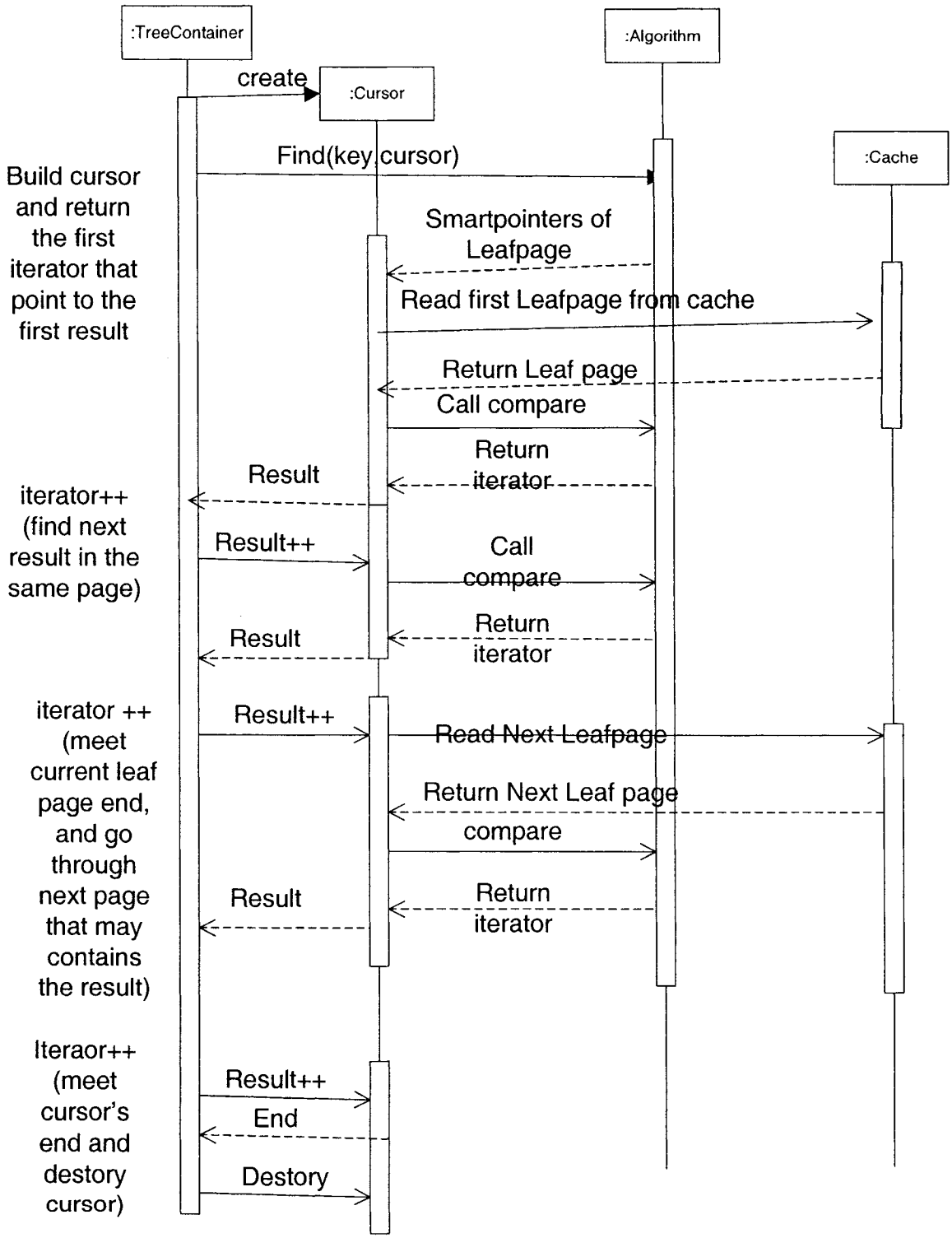

Figure 3.8: Activity of search progress through cursor and iterator

3.5.2 Proxy Layer

The Proxy Layer is the middle tier in the search tree index framework architecture. It bridges the Container Presentation Layer and the Physical Layer. Figure 3.9 shows the high-level design of the Proxy Layer. Proxy Layer is used to manage access control page objects between secondary storage and memory. There are some important issues related to page object access control including access control between memory and physical storage, cache strategy, lock policy and object instance management.

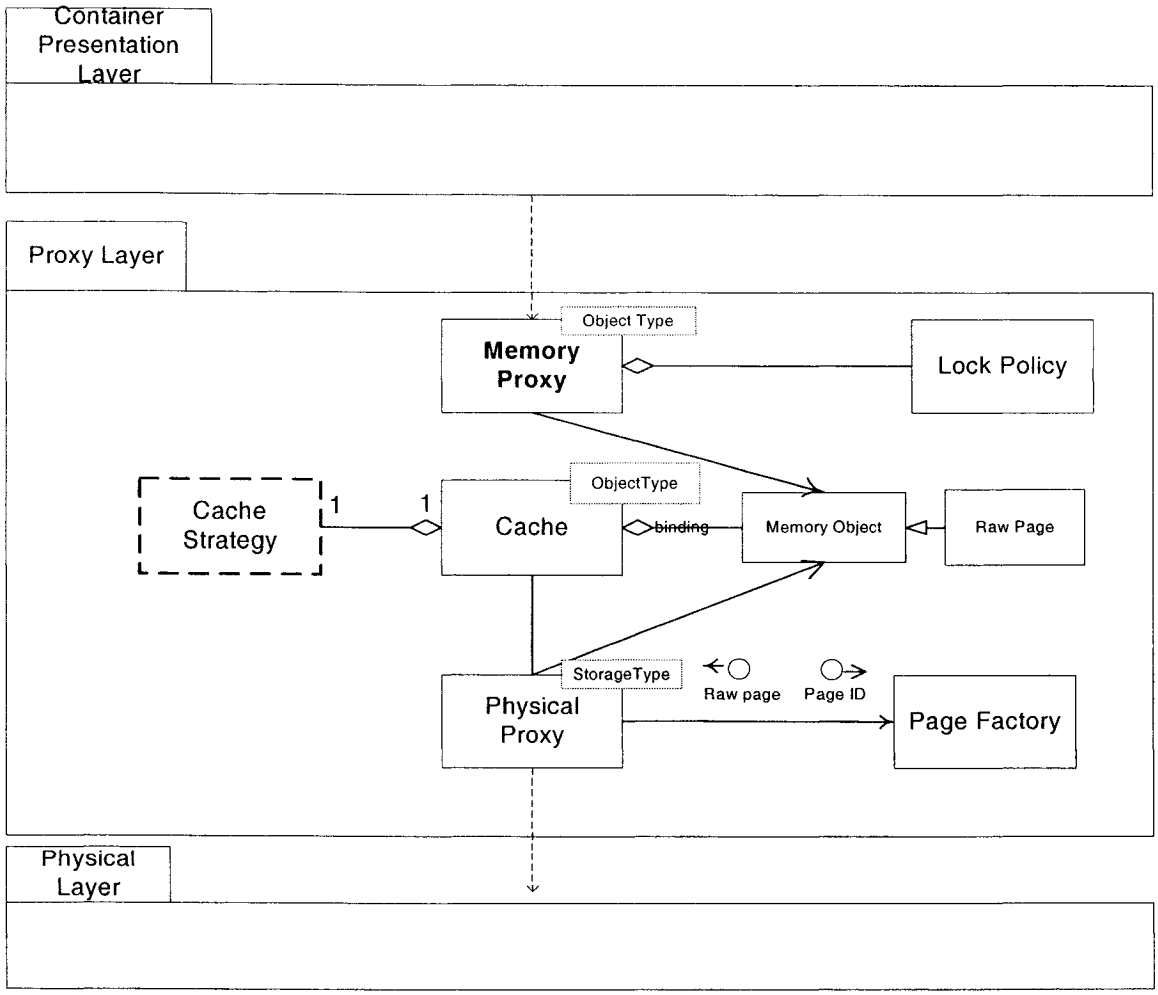

Figure 3.9: High-level design of the Proxy Layer

Access Control Issue

In general, a tree index container does not have to fit completely in memory. However, certain index pages and leaf pages, which is designed to be small enough to easily fit into memory, should be loaded in memory to allow the tree operations access at run-time. That means some page objects have to be loaded on demand. Therefore, we introduce a proxy strategy. At run-time, a proxy strategy is chosen and objects are loaded implicitly on demand under unawareness of the end user. Two kinds of proxy are implemented in the index framework, memory proxy (smart pointer) and physical proxy.

In the index framework implementation, a smart pointer is a replacement for a bare pointer that performs additional actions when object is accessed. A smart pointer is used as a memory proxy object for many purposes: counting the number of references to real object so that it can be freed automatically when they have no more references; loading a persistent object into memory when it is first referenced; and checking that the real object is locked before it is accessed to ensure that no other objects can change it.

Physical proxy class encapsulates the both the proxy pattern for page objects and the abstract factory pattern for page object instance creation. Physical proxy is used to manage memory allocation and de-allocation for tree pages, transparently serialize or de-serialize index search tree from physical storage layer to page cache or from page cache to physical storage Layer.

To further understand the behavior of proxy, we illustrate three scenarios. The first one, shown as Figure 3.10, is a scenario that tree container uses memory proxy (smart pointer) to access a non-root page. The memory proxy (smart pointer) will check

if the page is in memory. If yes, the memory proxy returns a smart pointer that contains a raw page pointer to the tree container.

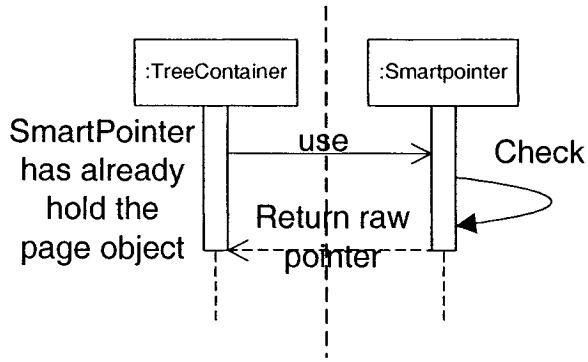

Figure 3.10: Sequence diagram for smart pointer has hold the page object

The second scenario is illustrated in Figure 3.11. When a tree container uses memory proxy access to a non-root page object, it first checks if the page object is in memory. If not, the memory proxy will check if the cache has a reference to the page. If the page reference is in the cache buffer, the cache can return the page object to smart pointer and return the raw pointer of the page object to the tree container layer.

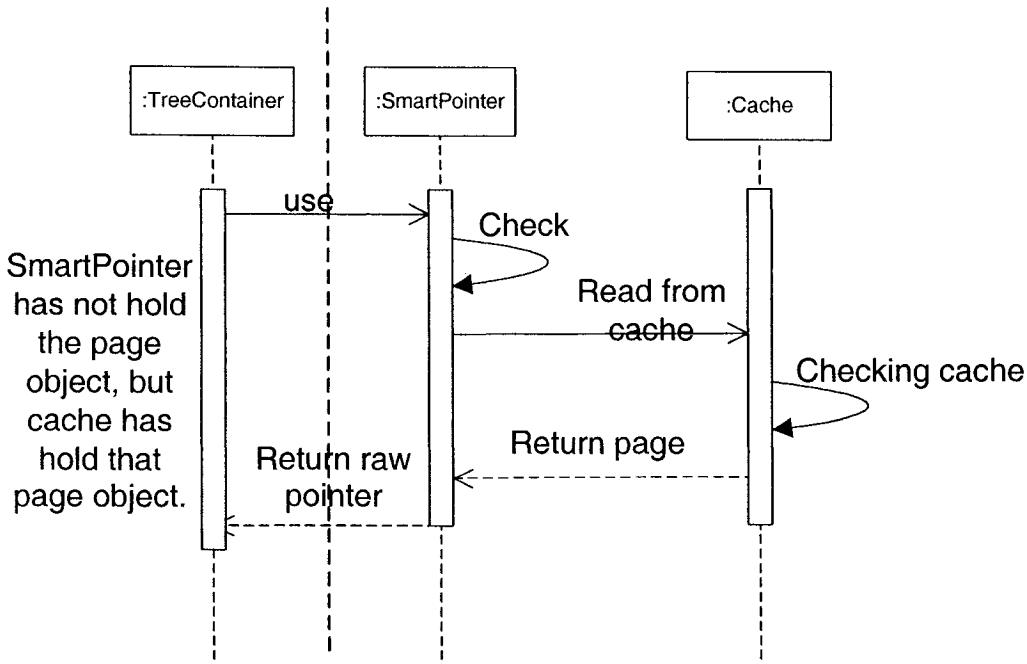

Figure 3.11: Sequence diagram for smart pointer does not hold the page object, but cache does.

The third scenario is illustrated in Figure 3.12. When a tree container uses memory proxy access to a non-root page object, it first checks if the page object is in memory. If not, the memory proxy will check if the cache has a reference to the page. If not, the cache will call the physical proxy to create a page object and serialize the page object from physical storage. Then, the physical proxy will put the page object into the cache, and the cache will return the page object in a smart pointer and finally return the raw pointer of page object to the tree container.

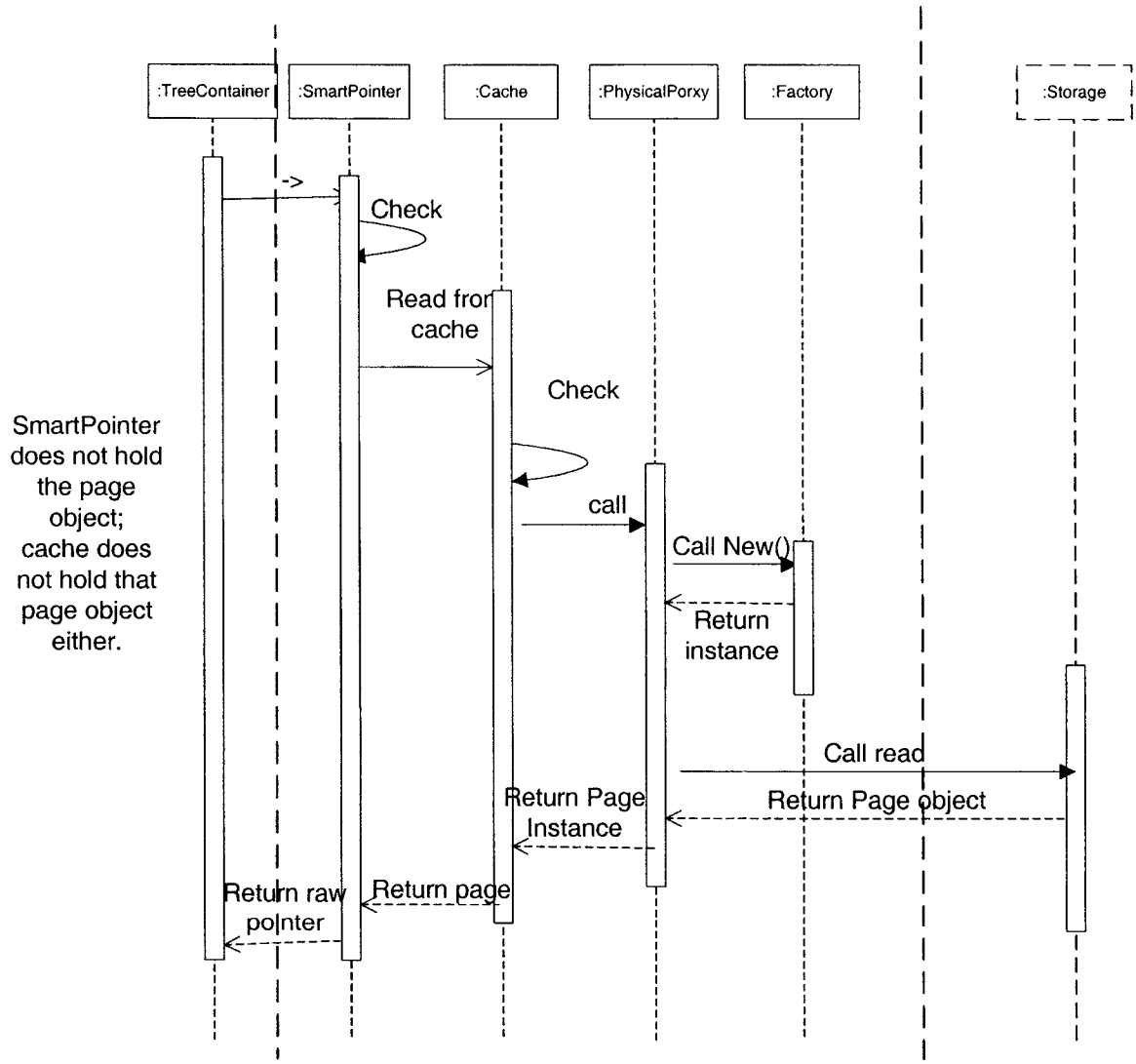

Figure 3.12 Sequence diagram for neither smart pointer nor cache holds the page object

Cache Strategy Issue

As we discussed before, on the one hand, objects can be created/loaded on demand; on the other hand, the memory (cache) has a limited capacity. It is therefore impossible to store all page objects in memory. So we introduce a Cache manager to solve this problem. The cache manager will implement a certain number of caching strategies. So at run-time, a cache strategy decides which page objects should be pre-stored and which page objects should be cached out and destroyed.

Lock Policy Issue

Since page objects are stored in the in-memory cache buffer, it is risky that multiple concurrent access operations on the same page object in the concurrency environment may corrupt the page. A locking mechanism is necessary. In the proxy layer, we provide a page lock mechanism. The smart pointer is integrated with the lock class for ensuring that the page object is correctly locked in the access. When a page object is accessed through the smart pointer, the page is locked automatically. After access, the page lock will also be released automatically.

Object Instance Management Issue

Because our system is a framework to extend many kinds of search tree access methods, each search tree may have different page object types. In order to reduce our system complexity of extension, an abstract factory class is provided to ensure that the right concrete page objects of search tree are created throughout our system.

3.5.3 Physical Layer

The index data should be in memory to allow the applications to access at run-time. But since any in-memory data is transient, we need to introduce a persistency mechanism to protect transient data from loss. The Physical Layer is such a subsystem to handle data persistency in the search tree index framework. Figure 3.13 shows a common high-level design of the Physical Layer. The Physical Layer contains two levels of data abstraction. The top level, Page Mapping management, is the one for mapping in-memory page object to persistent page object. The bottom level, Persistent Data management is for managing format of persistent page object. There are two issues about how to manage index persistent data of physical storage and how to map in-memory page object to physical storage.

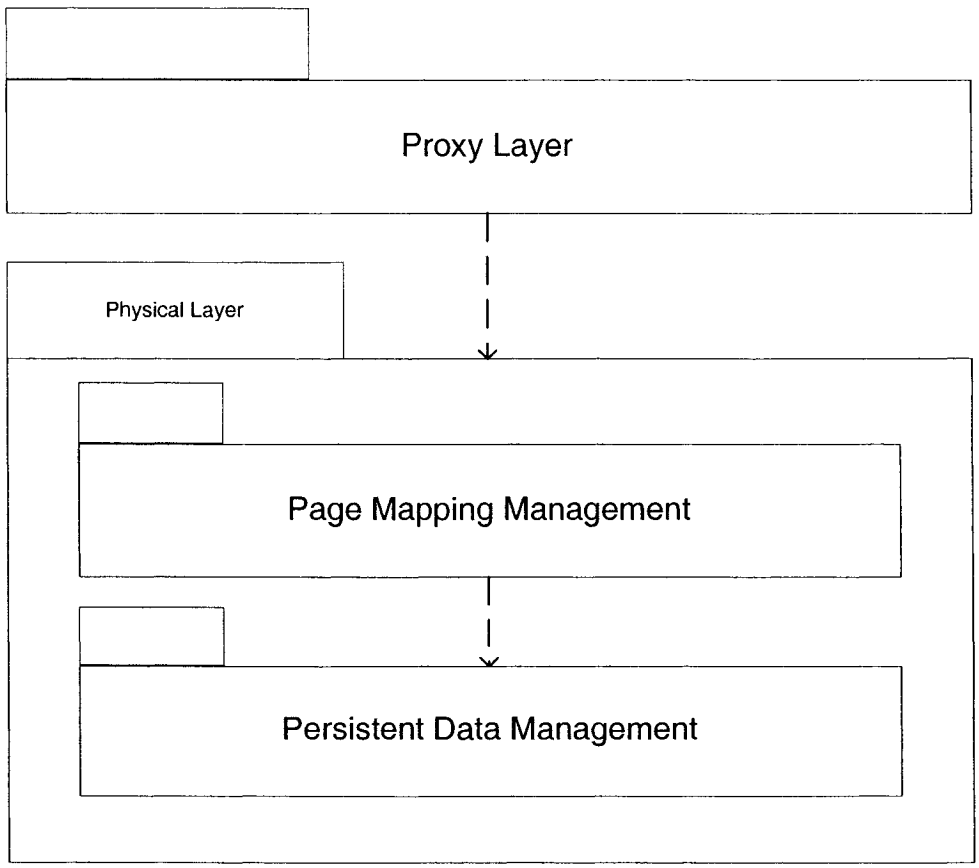

Figure 3.13: High-level design of the Physical Layer

Page Mapping Issue

In the design of the physical layer of the framework, a stream is an ordered collection of objects of a particular type, stored in external storage, and accessed in sequential order. Streams can be thought of as fundamental physical objects, which map volatile, typed index entry data elements in internal memory to persistent, untyped data elements in external physical storage. Streams are read and written like file in Unix, and support a number of primitive file-like operations such as *read()*, *write()*, *fseek()*, etc. Since different search trees have different page types, how to read/write different complex page object structures from/to varying data structures is a problem. We introduce a serializer to let page objects to be serialized for particular backend and external representation formats.

Persistent Data Management Issue

Since all index operations are based on in-memory index page and leaf page objects, in storage implementation, search tree index also is divided into page and then each is stored in a block of contiguous location on disk. A block is the basic unit for I/O operation. The size of block is usually determined by many factors such as the characteristics of the hard disk drive, and the amount of memory available. In common, the size of a page is equal to a block size of hard disk.

When a new page object is needed to write into the index files, Storage class allocates a block for it on the physical storage. When a page object is deleted from the files, storage collects the block used by the page and reallocates it.

Chapter 4 Implementation of Search Tree Index Framework

This chapter discusses the design methodology to achieve generalization at multiple levels and presents the structure and components of the search tree index framework in detail.

4.1 Container Presentation Layer Implementation

The modularization mechanisms and layer strategies discussed in last chapter are used to develop generalized search tree that provides the basis for common tree access methods used in database systems. The module can provide a search tree implementation by plugging STL tree containers into the framework. User can move through tree containers using standard STL iterators; and if using *insert ()*, *erase ()* or *replace ()* index entry in the containers, changes can be automatically committed to the persistent data.

To make the search tree assess method extension as easily as possible, we designed an STL container of generalization of database search trees based on the general notion of search keys and the essential nature of tree structures as observed in the GiST paper [HNP95] .

The search tree is designed to be a container based on the STL style that provides an iterator to its contents. The only way to interact with the container is through its iterator. Iterators provide access to the elements of these data structures without having to use or access the particular data structure's implementation. Iterators can be used to perform general queries on search tree structures, while providing a clean interface for

these queries. The basic component index and leaf page are also designed as containers. They will be retrieved from the hard disk into memory through a proxy.

4.1.1 Tree Structure Implementation

The Composite design pattern is a good way to implement the tree structure. Figure 4.1 shows the structure of Tree class diagram. To implement the Composite design pattern, we add a Page class as the base class of IndexPage and LeafPage, which is an interface that declares some common virtual functions. For every function in the Page interface, the IndexPage and LeafPage classes implement it respectively in their own classes. The Composite design pattern uses the page pointer to access both the IndexPage and LeafPage without distinguishing between them.

The search tree index container will be built on leaf and index page components, so the elements that the index stores are of type IndexPage and LeafPage. These components will provide the necessary index structure, and manage the access and storage of the index through a proxy mechanism. Tree structure holds a pointer to a page but it can get references to the index page and leaf page through this page using the *getindexpage()* or *getleafpage()* function, and then can invoke the class-specific functions such as *begin()*, *end()* and *insert()* through their references. They are passed to the index as a template type in the implementation phase. This makes the index independent of the page design and work with any page type. The index is a container that provides an iterator to its contents, which provides controlled access to the elements of the container. The index iterator can iterate through pages, one page at a time. This is exactly the database concept of indexes: paged indexes to facilitate access and improve performance.

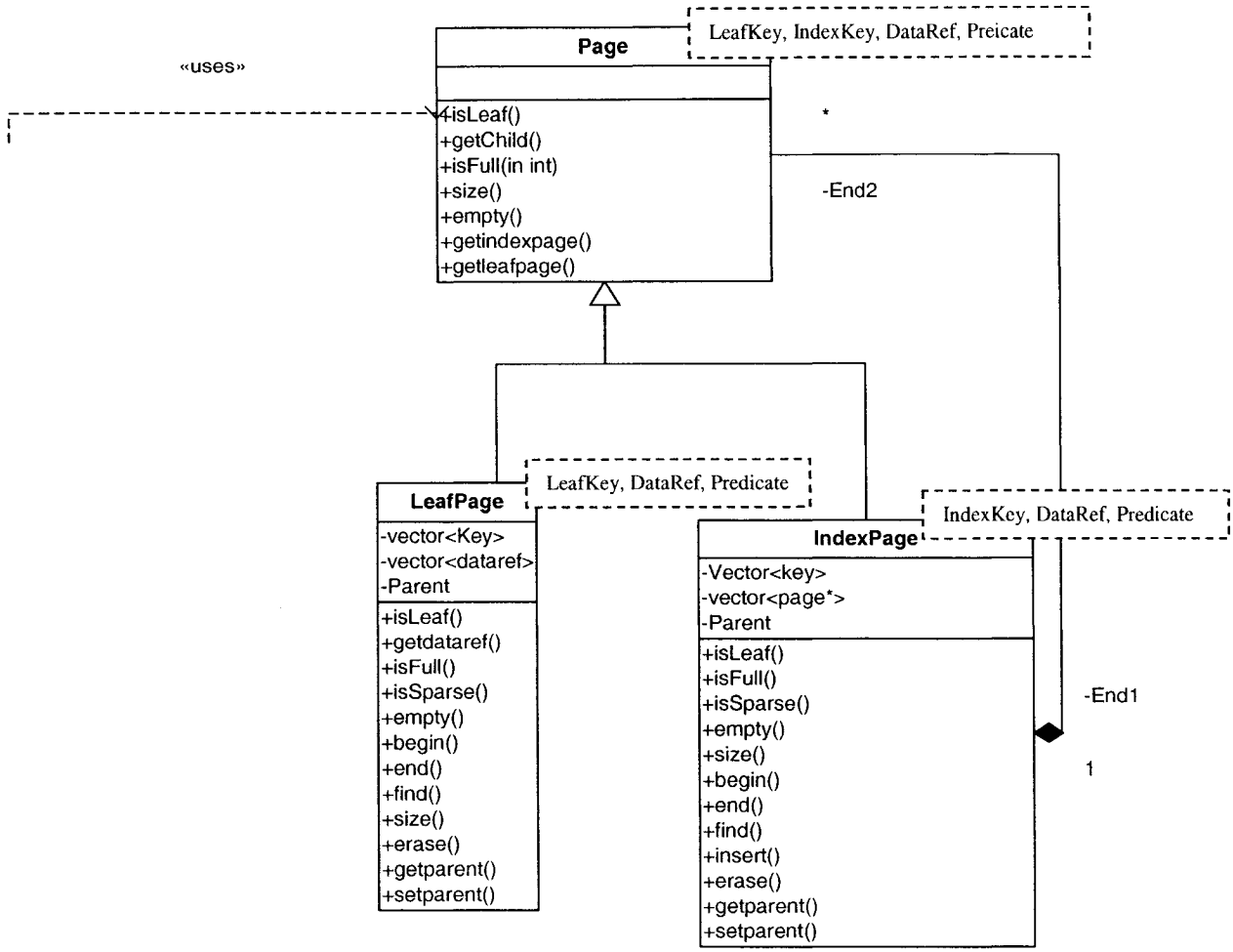

Figure 4.1: Tree Structure Using Composite Design Pattern

4.1.2 Basic Components Design

The index and leaf page are also designed as containers of STL style and generic programming on a small scale. While the index typically resides on hard disk, a page is small enough to fit in memory. Whenever a page is needed, it is retrieved from the hard disk into memory through a proxy. At this point the page can perform its tasks of searching for a key in its contents, accepting new entries, deleting some existing ones, and so on. This design produces a simple system structure without affecting its complexity or extensibility.

The elements of a leaf page container shown in Figure 4.2 is a pair with entries of the form (Key, Data object/Data reference) where key refers to a field in database, and data reference is the physical reference of tuple. An index page shown in Figure 4.3 is also a container that has entries of the form (key, child-pointer) where a child-pointer is the address of low page or a sub-tree and an index key logically matches all data stored in its sub-tree. Notice that children pointer and data reference are different type.

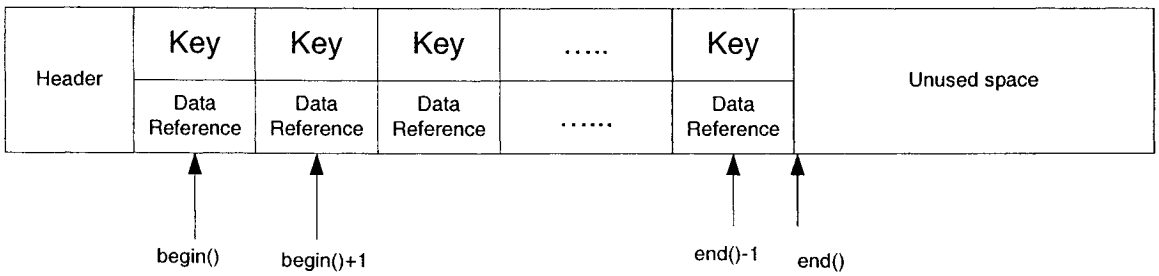

Figure 4.2: LeafPage Structure

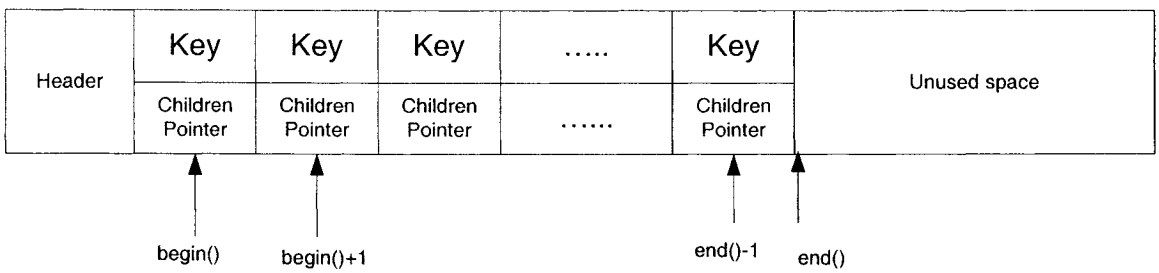

Figure 4.3: IndexPage Structure

Different search trees are used to solve specific problems in different domains, and therefore, have different structures for the keys in the page. Leaf keys are used to uniquely identify the data objects in the database. Examples of key structures are integer values for data in a B-tree, coordinate values for points in a K-D-B-tree and rectangles for regions in an R-tree. Index key structures may be different from the leaf key depending

on the kind of tree. For example, keys of K-D-B-tree are rectangles in index page, and are points in the leaf page.

Although these two kinds of pages are containers, which have almost the same interfaces following STL container standard, the interfaces are not programming but conceptual interfaces. Iterators of the IndexPage and LeafPage class have the same conceptual name, but they are completely different types essentially. An iterator in LeafPage points to a pair of Key and Data Deference, and an iterator in IndexPage points to a pair of Key and Page pointer. Moreover, IndexPage and LeafPage classes are containers that follow the STL style. There must be a number of functions such as: *iterator begin()*; and *iterator end()*; which are the same conceptual name but different meanings and types. Figure 4.4 and Figure 4.5 shows the interface of the LeafPage class and the IndexPage class.

```
template<
            class LeafKey,
            class IndexKey,
             class DataRef,
            class Predicate
            class streamable=streamtype
      >
class LeafPage: public Page<LeafKey, IndexKey, DataRef, streamable, Predicate>
{
 public:
.....
      typedef  LeafKey key_type;
      typedef  std::pair<Key,DataRef> value_type;
      typedef  std::vector<Key>  KeyContainer;
      typedef  std::vector<DataRef> RefContainer;
      typedef  std::vector<WeakPointerType> PageContainer;
      typedef  typename KeyContainer::iterator iterator;
      typedef  typename KeyContainer::reverse_iterator reverse_iterator;
      typedef  typename RefContainer::reverse_iterator reverse_RefIterator;
      typedef  typename RefContainer::iterator RefIterator;

 public:
      LeafPage(long Max=MaxElement){}
      ~LeafPage() { };
      void clear(){ }
      size_type erase(const key_type key) { }
      iterator find(const key_type key, Predicate pre){}
      iterator find(iterator first, iterator last, const key_type key, Predicate pre) {}
      void erase(iterator pos) {}
      iterator insert(const value_type value) {}
      Iterator insert(const key_type key, const DataRef value){}
      size_type size();
};
```

Figure 4.4: The Interface of LeafPage Class

```
template<
            class LeafKey,
            class IndexKey,
            class DataRef,
            class Predicate
            class streamable=streamtype
      >
class IndexPage : public Page< LeafKey, IndexKey, DataRef, Predicate, streamable
>
{
 public:
     typedef  std::vector<Key> KeyContainer;
     typedef  std::vector<WeakPointerType> PageContainer;

     typedef  typename KeyContainer::iterator iterator;
     typedef  typename PageContainer::iterator PageIterator;
     typedef  typename KeyContainer::reverse_iterator reverse_iterator;
     typedef  typename PageContainer::reverse_iterator reverse_PageIterator;
public:
      IndexPage(IndexPage&) {};
      IndexPage(long Max=MaxElement){}
      ~IndexPage(){}
      void clear(){};
      size_type erase(const key_type key) {}
      iterator find(const key_type key, Predicate pre) {}
      void erase(iterator pos)  {}
      iterator insert(const key_type key, const PointerType value){}
      ..............................
};
```

Figure 4.5: The Interface of IndexPage Class

4.1.3 Tree Container Class

The generic programming techniques with the STL iterator and container standards means we can plug our abstractions into a wide variety of algorithms for searching and manipulation into our search tree container.

The B+-tree may have a different deletion algorithm and an R-tree may use a different node splitting algorithm. Hence, we must design two base tree containers: B-tree container for extending linearly ordered search tree such as B-tree, B+-tree application, and R-tree container for spatial search tree such as R-tree, R*-tree, SS-tree, SR-tree. It is

now possible to write a general function like *size()* that can get the number of components of either a B-tree or an R-tree. A specific search tree may extension from specific ones.

We use multi-policy, a generic programming technique, to design the tree container class. Two base tree containers are designed as two policy classes that are implementations of search tree functional design choices. They are not intended for standalone use; instead, they are inherited from search tree container, for example, the search tree can be defined as:

```
template <
            class DataRef,
            class PageType,
            class IndexKeyType,
            class LeafKeyType,
            class IndexPageType,
            class LeafPageType,
            class Predictor,
            class GiSTExtension,
            template< class, class, class, class, class, class, class > class SearchTree
           >
class SearchTreeIndex: public SearchTree < DataRef, PageKeyType, PageType,
LeafPageType, IndexPageType, Predictor, GiSTExtension>
{
.....
}
```

When we implement common linearly ordered search tree extension, we use BtreeIndex policy class as SearchTree template parameter of SearchTreeIndex class; likewise, when we implement spatial search tree extension, we pass Rtreeindex policy class to SearchTree parameter of SearchTreeIndex.

Two search tree policy classes BtreeIndex and RtreeIndex not only define a tree function interface, but also they implement that interface. They all implement common functionalities of either common linearly ordered search tree, or spatial search tree, which

are general enough to apply to most search tree and are independent of any implementations of inner structures. The following algorithms are designed to capture the common parts in search, insertion and deletion operations and stub routines are used within them so that the specific parts related to a particular tree structure can be specified by user.

Search

Recursively descend all possible paths in the tree whose keys match with the search key.

- S1. [Search sub trees] if the root is a leaf, go to step S2, else check possible sub trees that match the search condition (predicate).
- S2. [Search leaf] check the entries in the leaf for a match with the search key.

This search algorithm basically can be used to search any search tree with any query predicate. The search condition can be an exact match (equality), range query, window query or any possible query predicate corresponding to the key type.

Insertion

An entry of a new key, with or without a new data object, is added to the leaf level. A leaf that overflows is split, and splits propagate up the tree.

- I1. [Find leaf L for the new record] if the root is a leaf, root is L, else choose a sub tree where the record should go until a leaf is reached
- I2. [Add record to the leaf L] if the leaf has room for another entry, add the record to the leaf, else split the old records in L and the new record into both L and a new leaf LL

- I3. [Propagate changes upward] adjust the tree according to new L to preserve search tree properties, passing LL if a split is performed
- I4. [Grow tree taller] if split propagation causes the root to split, create a new root

Choosing a sub-tree in step I1 and splitting a tree node in step I2 are application dependent. In steps I3 and I4, where changes propagate upward, to calculate the new parent key according to its new sub-tree is also dependent on the specific application.

Deletion

Remove an entry from its leaf node. If this causes underflow, adjust tree accordingly by updating key in parent nodes to preserve search tree properties.

- D1. [Find node containing entry] invoke search to locate the leaf node L containing the entry, stop if the entry is not found
- D2. [Delete entry] remove the entry from leaf L
- D3. [Propagate changes] adjust the tree according to new L to preserve search tree properties
- D4. [Shorten tree] If the root node has only one child after the tree has been adjusted, make the child the new root.

Our search tree container provides the common parts of the tree operation algorithms and requires a programmer to give specific tree behavior routines to complete the operations of individual search tree. The following tree behavior routines are defined and are derived from the GiST work, but have been modified.

- **Consistent:** Consistent takes a query predicate and a page entry as arguments and returns true if the entry matches the predicate. This routine is used by the search and deletion algorithms.
- **Union:** Union computes the expanded predicate as the union of the old sub-tree predicate and new key. This routine is used for adjusting the tree in insertion and deletion algorithms to maintain the tree property.
- **PickSplit**: PickSplit determines the split strategy by specifying which of the entries on a page move to the new right sibling page during the split. This routine is used by the insertion algorithm.
- **ChooseSubTree**: ChooseSubTree is used to choose a sub tree to insert a new record by using equality and range containment queries. This routine is used by the insertion algorithm.

These routines specify particular behavior and implantation of a search tree, so that they are external parts of search tree container, which are used to extend a general search tree container to a special search tree container. These functions are part of implementation of a tree operation. Tree behavior routines cannot be called directly, as can tree operations, only indirectly from within tree operations.

For extending a special search tree container, user must implement these special tree behavior routines as static function in a class, which these routines can be passed as a single argument to either a common linearly ordered search tree, or a spatial search tree. A set of static methods in the class provides behavior routines for an individual search tree corresponding to a particular implementation. Because the actual implementations of these methods depend on the implementation of the containers, so we need pass the key

type and page type to the policy class as its type parameters. The Figure 4.6 shows the interface of the behavior routine class.

```
template <
            class PageType,
            class IndexKeyType,
            class LeafKeyType,
            class IndexPageType,
            class LeafPageType,
            class Predicate
            >
class GISTExtension
{
     typedef SmartPtr<PageType> PointerType;
     typedef WeakPtr<PageType> WeakPointerType;
public:
      static bool Consistent( IndexKeyType, LeafKeyType );
      static IndexKeyType Union( PointerType );
      static PointerType ChooseSubTree(PointerType, LeafKeyType);
      static void PickSplit( PointerType, PointerType);

};
```

Figure 4.6: Interface of the Behavior Routine Class

4.1.4 Predicate of Search Tree Container

Search is the fundamental operation when using the index. The application uses the index to locate data by searching its contents to find where the physical data resides. As mentioned before, the index does not provide the data we are looking for, but tells us where it is stored, so the data returned by the index is typically a reference to a location. The other index use cases, inserting and deleting, also depend on the search operation. In order to delete an element, we must find its location first. Also to insert an element, we must search for the best location to insert it. For unique elements, we must search before inserting an element to make sure it does not already exist.

The generic programming techniques compliance with the STL iterator and container standards means we can plug our abstractions into a wide variety of algorithms for searching and manipulation into our search tree container. STL style uses *find()* and *find_if()* functions to do any search algorithms, which use a function object—predicate as a function parameter and adapt the searching algorithms. A predicate is an expression that returns a Boolean value. Similarly, a function object that returns a Boolean value is a predicate object.

The search tree find function first uses the index page internal find function to go through entries beginning from the root page if root is a index page, and pass predicate along with key to find the sub-tree root page that matched the query algorithm in the predicate. Recursive doing that will find the leaf page, which may contain the elements. Then, search tree find function will pass predicate to leaf page internal find function and find the result.

However, for some searching tree, some searching tree strategy algorithm is different between leaf page and index page, A R-tree is a well-known example with this properties: The R-tree access method uses the four required strategy functions, overlap, equal, contain, and within in a variety of combinations when searching in an R-tree index, as described by the following table in Table 4.1. For example, R-tree equal strategy function is used to give an object O and find the exactly match O in the R-tree, which uses contains algorithm through index page iterator to locate the leaf page that may contain O, and uses equal algorithm through leaf page iterator to get the result. Our predicate, shown as Figure 4.7, is designed to adapt two different algorithms. The integer

variable tag in the predicate is a flag that is used to tell the page container, index page and leaf page, to identify which Functor should be used.

Strategy Function	Algorithm called on an index page	Algorithm called on a leaf page
Overlap	Overlap	Overlap
Equal	Contains	Equal
Contains	Contains	Within
Within	Overlap	Contains

Table 4.1: R-tree basic strategy functions

```
template<class LeafKey, class IndexKey, class value_type>
class Predictor : public std::binary_function<value_type,value_type,bool>
{
private:
        typedef FunctorHolder2<bool, LeafKey, value_type>  LeafFunctor;
        typedef FunctorHolder2< bool, IndexKey, value_type> IndexFunctor;
        int tag;
        value_type* value;
public:
        LeafFunctor leaffunctor;
        IndexFunctor indexfunctor;
        ~Predictor(){};
        Predictor(value_type* key)  {};
        bool operator () (value_type* rhs) { };

};
```

Figure 4.7: Interface of Predicate

Furthermore, predicate design gives our search tree index a good ability of customizing user-defined function, a key ingredient of ORDBMS.

4.2 Proxy Layer Implementation

The Proxy layer responses for memory management and manage controlled access between memory and physical storage. Proxy layer is used to manage memory allocation and de-allocation for tree pages, transparently serialize or de-serialize index

search trees from physical storage to memory or from memory to physical storage, also provide cache technology to efficient system.

4.2.1 Proxy Mechanism

The index framework uses a proxy mechanism to manage controlled access between secondary storage and memory. Two kinds of proxy are implemented in the index framework, memory proxy (smart pointer) and physical proxy.

Only the root of a tree is loaded initially, and resides in the memory until the index tree is destroyed. Each access to a non-root page checks if the page is in memory. If yes, the memory proxy returns a smart pointer that contains a page raw pointer to the tree methods; otherwise, the memory proxy will check if the Cache has a reference to the page. If the page reference is in the Cache, the Cache can return the page object from the cache. If not, the Cache needs to call physical proxy to create a page object and serialize the page object from physical storage.

4.2.2 Memory Proxy/ Smart Pointer

In the Index framework implementation, a smart pointer is used as a memory proxy object for many purposes including counting the number of references to real object so that it can be freed automatically when there are no more references, loading a persistent object into memory when it is first referenced, and checking that the real object is locked before it is accessed to ensure that no other object can change it.

A smart pointer is like a bridge between index tree memory container and cache. It is important for STL-style containers to consider the specific garbage collection scheme used. A non-intrusive reference counting smart pointer in Figure 4.8 is suited for

algorithms layers of the index framework to connect with Proxy layer seamless and access physical layer transparently. When we need the index algorithms to connect with storage, one thing to do is that all the raw pointers are simply replaced with smart pointer, and do not need to change any other parts.

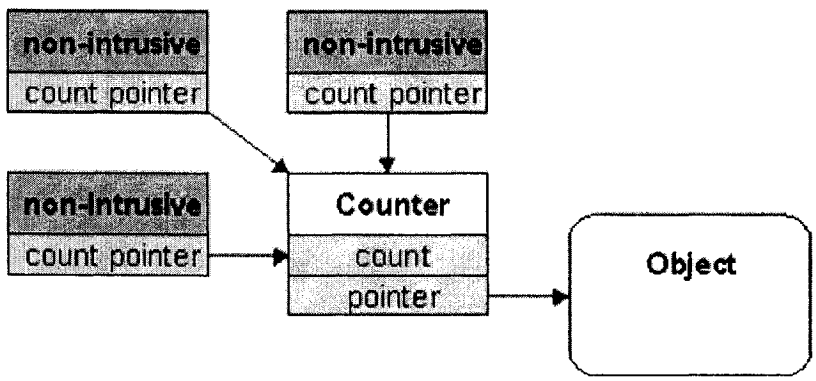

Figure 4.8: Non-intrusive Reference Counting Smart Pointer

The number of object references is stored in a counter that maintains a count of the smart pointers that point to the same page object. The smart pointer will delete the object when this count becomes zero. The disk reference holds a page in the index files, and a raw pointer maintains a reference to a loaded page in cache.

Figure 4.9 shows the interface of smart pointer class. The template parameters T and Cache stand for a reference object type and cache type respectively. By using smart pointers, a page is only loaded on demand. The need of loading is determined by using * and -> operations of a smart pointer. When a smart pointer calls * or -> operation, it first checks if the page object is already in the memory. If not, it will call the cache manager to load the page from the storage; otherwise, it will check if the page is dirty. If the page is dirty, it will call the cache manager to reload the page.

```
struct RefCounts
{
   RefCounts () ;
   bool dirty_;
   long totalRefs_;
   long strongRefs_;
};
template <class T,class Cache>
class SmartPtr
{
public:
   typedef Cache CacheType;
   typedef T element_type;
   typedef T* PointerType;
   typedef T& ReferenceType;
   SmartPtr ()  { }
   SmartPtr (const SmartPtr& other)  { }
   SmartPtr(T* obj) { }
   template <typename O>
   SmartPtr(const SmartPtr<O,CacheType>& other)  { }
   template <typename O>
   SmartPtr(const WeakPtr<O,Cache>& other)  { }
   SmartPtr& operator = (const SmartPtr<T,Cache>& other)  { }
   template <typename O>
   SmartPtr& operator = (WeakPtr<O,Cache>& other)  { }
   SmartPtr& operator = (Null)   { }
   ~SmartPtr ()  { }
   T& operator * ()  { }
   T* operator -> ()  { }
   void swap (SmartPtr<T,CacheType>& other) { }
   void SetID(long id) { }
   long GetID() { }
   bool is_null() { }
   long GetCounter(){ };
   void dirtied(){ };
private:
   T* ptr_;
   RefCounts* refCounts_;
   long id_;
.....
}
```

Figure 4.9: Interface of Smart Pointer Class

Because the implementation of smart pointer uses reference counting, cycles of smart pointer instances will not be reclaimed. For example, an object A holds a smart pointer to an object B. Also, object B holds a smart pointer to A. These two objects form a cyclic reference; even though you do not use any of them anymore, they use each other. The reference management strategy cannot detect such cyclic references, and the two objects remain allocated forever. The cycles can span multiple objects, closing circles that often reveal unexpected dependencies that are very hard to debug. We use a weak pointer to "break cycles."

A weak pointer, as shown in Figure 4.10, will be just like regular reference counter pointers; only their references will not count. If there are a smart pointer and a weak pointer pointing to the same object, and the smart pointer is destroyed, the weak pointer immediately becomes null.

We use the smart pointer to replace the pointer that points from the index page to their children pages and use weak pointer replace the pointer that points from the children page to its parent's page.

```
template <typename T,typename Cache>
class WeakPtr
{
public:
   typedef Cache CacheType;
   typedef T element_type;
   typedef T* PointerType;
   typedef T& ReferenceType;

   WeakPtr () { }
   WeakPtr (const WeakPtr& other) { }
   WeakPtr(T* obj) { }

   template <typename O>
   WeakPtr (const WeakPtr<O,Cache>& other){ }

   template <typename O>
   WeakPtr (const SmartPtr<O,Cache>& other) { }

   WeakPtr& operator = (const WeakPtr<T,Cache>& other) { }

   template <typename O>
   WeakPtr& operator = (SmartPtr<O,Cache>& other) { }

   WeakPtr& operator = (Null) { }
   ~WeakPtr () { }
   T& operator * () const { }
   T* operator -> (){ }
   void swap (WeakPtr<T,CacheType>& other) { }
   void SetID(long id) { };
   long GetID() { };
   bool is_null() { };
   long GetCounter(){ };
private:
   T* ptr_;
   RefCounts* refCounts_;
   long id_;
.....
}
```

Figure 4.10: Interface of Weak Pointer Class

4.2.3 Physical Proxy

Physical proxy class encapsulates the both the proxy pattern for page objects and the abstract factory pattern for derived classes. Physical proxy is used to manage memory

allocation and de-allocation for tree pages, transparently serialize or de-serialize index search tree from physical storage to page cache or from page cache to physical storage.

When a smart pointer needs access to a page, and the cache does not has a reference to the page, the cache will call the Physical Proxy to read the page object from storage. The Physical Proxy first locates the physical position of the page object in the physical storage by using the page reference and read page type information; then, it calls the page factory and the memory allocator to build the page object and de-serialize the page object from storage. Figure 4.11 shows the interface of Physical Proxy class.

```
template < class TYPE, class storagetype, class typeidtype=long >
class PhyProxy
{
private:
    storagetype * storage_;
public:
    typedef TYPE* pointer;
    PhyProxy(){};
    PhyProxy( storagetype * storage):storage_(storage) {};
    PhyProxy( const PhyProxy& src ):storage_(src.storage_){};
    inline void operator=( const PhyProxy& src ) { };
    storagetype * GetStorage() {}
    ~PhyProxy() { };
    inline void DeleteObject(long objectid) { }
    inline void DeleteObject(pointer object) { }
    bool HasRoot() {}
    long GetRootID() { }
    pointer GetRoot() { };
   void SetRoot(long id_) { }
    inline pointer GetPointer(long id_) { };
    pointer CreateNew(typeidtype typeid_ ) { };
};
```

Figure 4.11: Interface of Physical Proxy Class

4.2.4 Page Cache Manager

The page Cache manager provides efficient access to storage files. To improve IO operation, the page cache is used for caching the most recently used pages and buffering

IO requests. In our design, a Singleton class is designed from Singleton design pattern to manage the Cache instance. A Singleton class is used when there must be exactly one instance of a class, and it must be accessible to users from a well-known access point. This ensures the Cache has only one instance and provides a global point of access.

Since the index has to get the index page object from storage for every accessed page object, the performance of the application mostly depends on the efficiency of the page object caching strategy. Usually a standard LRU algorithm is used for replacing entries in the cache, but for a database index this algorithm is not always a good choice.

When the cache is full and needs to replace the page objects, with a traditional LRU cache replacement algorithm, it will scan a large number of cached page objects, and find the page of lowest access rate to be replaced. But the newest cached page object always has the lowest access rate than other cached page object, which means it has the highest priority to be replaced. Moreover frequently scanning cached page objects makes the system performance significantly slow.

To avoid such undesirable behavior, a special modification First-In-First-Out algorithm is used in the Index framework. The page cache buffer is divided into two parts: an index pages part and leaves part. Both parts are controlled by an ordinary First-In-First-Out discipline using a vector. The non-leaf part has higher priority to stay in cache memory than leaves part, because for tree structure, index pages have much more access rate than leaf pages. When cache needs to load a page in to cache and the cache is full, it checks if the leaves part is empty. If not, it finds the first non-used page from the leaves part, then cache it out, and cache a new one in; otherwise, it finds the first non-

used index page from the index pages part. Figure 4.12 shows the interface of the Cache class.

When the Cache caches the page objects out the buffer, it checks if the page is modified. If yes, it will call physical proxy to serialize the page object to storage. However, the Cache class will not destroy the page object immediately, but instead; it puts a dirty flag on the page object. The dirty flag ensures smart pointer will not access that object page and will destroy it safely.

```
template <class T,  class StorageType >
class cache{
public:
   typedef long K;
   typedef cache < T,StorageType > CacheImpl;
   typedef PhyProxy < T,StorageType > PhyProxyType;
   typedef SingletonHolder<CacheImpl> CacheType;
   typedef SmartPtr<T,CacheType> PointerType;
   typedef WeakPtr<T,CacheType> WeakPointerType;
   typedef std::map<K,PointerType> BufferType;
   typedef std::vector < K > KeyIndexType;
   typedef K key_type;
   typedef T cachedType;
   typedef typename container_type::value_type value_type;
   typedef typename container_type::size_type size_type;
   typedef typename container_type::reference reference;
   typedef typename container_type::const_reference const_reference;
   typedef typename container_type::iterator iterator;
   typedef typename container_type::const_iterator const_iterator;
private:
   container_type container_;
   long MaxElement_;
   PhyProxyType *proxy_;
   KeyIndexType keyindex_;
   PointerType Root_;
public
   cache(long Maxe=512):MaxElement_(Maxe){Root_ = NullPtr;};
   void SetProxy(PhyProxyType * proxy){ proxy_ = proxy;}
   ~cache() throw() { };
   iterator begin() { }
   const_iterator begin() const {  }
   iterator end() { }
   const_iterator end() const { }
   size_type size() const {  }
   bool empty() const { }
   PointerType GetPointer(key_type key)  {}
   void insert(K key, PointerType ptr)  {  };
   PointerType CreateNew(defaultIDKeyType id) {  }
   void DeleteObject(long id)
   void erase(iterator pos) { }
   void clear() {}
   PointerType GetRoot(){}
.....
};
```

Figure 4.12: Interface of Cache Class

4.3 Physical Layer Implementation

In order to rebuild an index from index files, we need to maintain a copy of the index structure on the disk. The Physical storage layer is mainly responsible for managing and controlling access to the index files on disk.

4.3.1 Persistent Data Storage Implementation

In general, the indexes are too large to fit in the main memory and must therefore reside on disk. Thus communication between internal and external physical storage, instead of actual memory computation time often become the bottleneck in the index access. This is due to the huge difference in access time of fast internal memory and slower external physical storage such as disks. The most critical feature of secondary storage devices is that it takes a relatively long time to seek to a specific location, but once the read head is positioned and ready, reading or writing a stream of contiguous bytes proceeds rapidly. So if we access a large block of contiguous data elements at a time, and perform the necessary algorithmic steps on the elements in the block while in the high-speed memory, the speedup can be considerable.

Hence, in storage implementation, search tree index is divided into pages and then each is stored in a block of contiguous location on disk. A block is the basic unit for I/O operation. The size of the block is usually determined by many factors such as the characteristics of the hard disk drive, and the amount of memory available. In common, the size of a page is equal to a block size of hard disk.

When a new page object is needed to write into the index files, The Storage class allocates a block for it on the physical storage. When a page object is deleted from the files, storage collects the block used by the page and reallocates it. Figure 4.13 shows the

basic tree index structure on the disk. The index file header holds the basic information of index file such page size root page position in the index file. Following the root page position, we can get the root page, and children page's position in the index file. Figure 4.14 shows the interface of storage class.

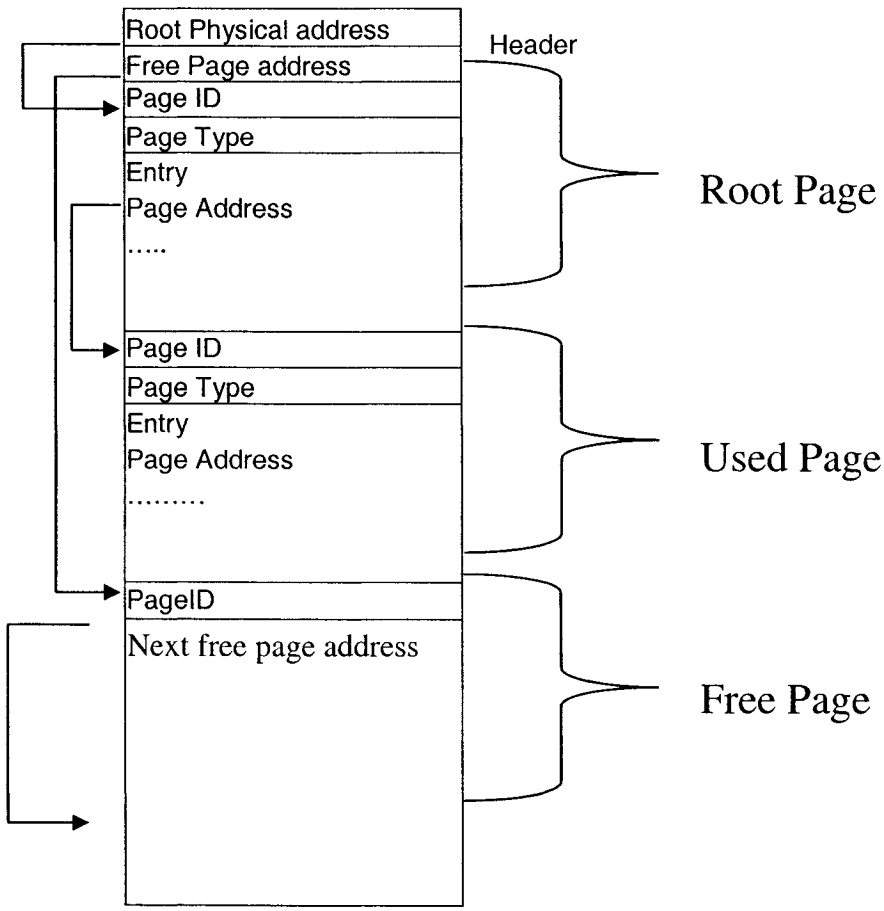

Figure 4.13: The Basic Tree Index Structure on the Disk

```
template <class streamable>
class BlockStorage :  public reader<streamable>,public
writer<streamable>
{
public:
  std::fstream  Block_file;
  std::fstream  Block_index;
  std::deque < long >  freeblock_deque_;
  char base_file_name_[FILE_NAME_LENGTH];
  Header<streamable> header_;
  size_t os_block_size_;
  size_t in_memory_blocks_;
  off_t file_pointer;
  char * place;
protected:
 long new_block_getid() { }
 void delete_block(long bid)  { }
 void set_block_location(long bid){ }
public:
 BlockStorage(const char *base_name, size_t
                  logical_block_factor=1)  { }
 void set_root(long bid) { }
 bool has_root() { }
 long get_root() { }
 long set_block(long bid=0){ }
 long free_block(long bid) { }
 size_t size() const  { }
 size_t file_size() const { }
 size_t block_size() const  { }
 size_t block_factor() const { }
  ~BlockStorage()  { };
.....
};
```

Figure 4.14: The Interface of Storage Class

4.3.2 Stream I/O Implement and Serialization

In the design of the Physical layer of the framework, a stream is an ordered collection of objects of a particular type, stored in external storage, and accessed in sequential order. Streams can be thought of as fundamental physical objects, which map volatile, typed index entry data elements in internal memory to persistent, untyped data elements in external physical storage.

The index framework contains a collection of stream classes that support reading and writing stream. We also introduce a serializer to define Serializable, Reader and Writer interfaces, and let page objects to be serialized to implement the Serializable interface and concrete Readers and Writers to be implemented for particular backends and external representation formats.

The Reader and Writer, as shown in Figure 4.15, are the classes for bytes streams in the framework. The Reader provides implementation for read streams that read 8-bit characters and The Writer provides implementation for write streams that write 8-bit characters. Furthermore, as with the Reader and Writer, most of container of STL can provide specialized I/O for stream. Figure 4.16 shows the interface for STL containers to serialize and de-serialize to stream by using the Reader and Writer.

```
template < class streamable >
class reader : public streamable::reader_type
{
    reader( const reader & ){ }; //no impl
    void operator=( const reader & ){ }; // no impl
 protected:
    virtual void read( serializable<streamable> & ) { }
 public:
    reader(){ };
    virtual long set_block(long bid)=0;
    virtual ~reader() { }
    reader < streamable > & operator>>( char & value ) { }
    reader < streamable > & operator>>( short & value ) { }
    reader < streamable > & operator>>( int & value ) { }
    reader < streamable > & operator >> (unsigned int & value ) { }
    reader < streamable > & operator>>( bool & value ) { }
    reader < streamable > & operator>>( long & value ) { }
    reader < streamable > & operator>>( std::string & value ) { }
    reader < streamable > & operator>>( float & value ) { }
    reader < streamable > & operator>>( double & value ) { }
    reader < streamable > & operator>>( long double & value ) { }
    reader < streamable > & operator>>( char* value ) { }
};
template < class streamable >
class writer : public streamable::writer_type
{
     writer( const writer & ){ }; //no impl
     void operator=( const writer & ){ }; // no impl
  protected:
     virtual void write( serializable < streamable > & obj ) { }
  public:
      writer(){ };
      virtual long set_block(long bid)=0;
      virtual ~writer() { }
      writer < streamable > & operator <<( const char & value ) { }
      writer < streamable > & operator<<( const bool & value ) { }
      writer < streamable > & operator<<( const int & value ) { }
      writer < streamable > & operator<<( const unsigned int & value ) { }
      writer < streamable > & operator<<( const short & value ) { }
      writer < streamable > & operator<<( const long & value ) { }
      writer < streamable > & operator<<( const float & value ) { }
      writer < streamable > & operator<<( const double & value ) { }
      writer < streamable > & operator<<( const long double & value ) { }
      writer < streamable > & operator<<( const std::string & value ) { }
      writer < streamable > & operator<<( const char * value ){ }
};
```

Figure 4.15: Reader Class and Writer Class Interface

```
template < class streamable, class T, class A >
inline writer<streamable>& operator<<( writer<streamable>& out, const std::deque< T, A > & cntr ) { }
template < class streamable, class T, class A >
inline reader <streamable> & operator>>( reader<streamable>& in, std::deque< T, A > & cntr ) { }
template < class streamable, class T, class A >
inline writer<streamable>& operator<<( writer<streamable>& out, const std::list< T, A > & cntr ) { }
template < class streamable, class T, class A >
inline reader<streamable>& operator>>( reader<streamable>& in, std::list< T, A > & cntr ) { }
template < class streamable, class T, class A >
inline writer<streamable>& operator<<( writer<streamable> & out, const std::vector< T, A > & cntr ) { }
template < class streamable, class T, class A >
inline reader<streamable>& operator>>( reader<streamable> & in, std::vector< T, A > & cntr ) { }
template < class streamable, class K, class T, class P, class A >
inline writer<streamable>& operator<<( writer<streamable>& out, const std::map< K, T, P, A > & cntr ) { }
template < class streamable, class K, class T, class P, class A >
inline reader<streamable>& operator>>( reader<streamable>& in, std::map< K, T, P, A > & cntr ) { }
```

Figure 4.16: Interface for Serializing and De-serializing STL Container

The Reader and Writer classes can decouple the primitive data type from the backend; however, for our index search tree, the Reader and Writer classes should not know the concrete index page classes. Thus, for a page class to be serialized, it must implement the basic serialization operations in the page.

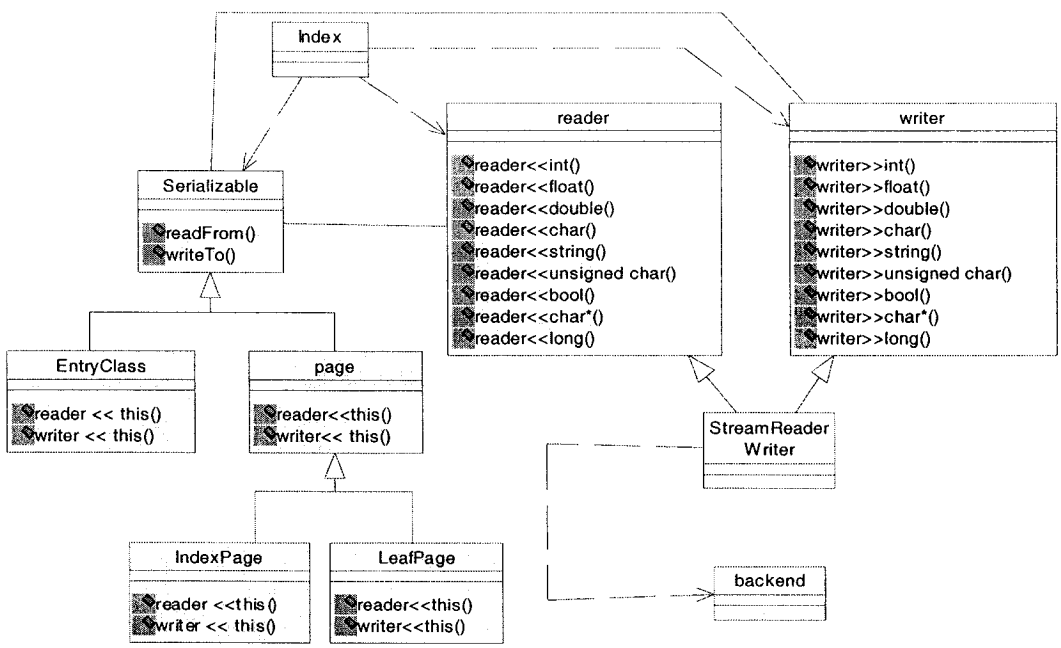

Figure 4.17: Class Diagram Related to Serialization

Figure 4.17 shows a class diagram related to serialization. In order to add this function to the index framework, we let the base class Page inherit from the Serializable class that declares serialize, deserialize and other related methods. Instead of directly reading and writing the storage, we serialize data to and from the Streamable physical storage. The Streamable uses overloaded insertion (<<) and extraction (>>) operators to perform writing and reading operations. IndexPage and LeafPage must implement these two methods: *serialize(Streamable &in)* and *deserialize(Streamable & out)*. We take the serialize operation shown in Figure 4.18 of the LeafPage as an example.

```
class LeafPage: public Page<Key, DataRef, streamable>
{
.....
    virtual void serialize(writer<streamable> & out)
    {
      serializable<streamable>::serialize(out);
      out << keycontainer;
      out << refcontainer;
      out << Parent_;
      .....
    }
    virtual void deserialize(reader<streamable> & in)
    {
      serializable<streamable>::deserialize(in);
      in >> keycontainer;
      in >> refcontainer;
      in >> refcontainer_;
      in >> Parent_;
      .....
    }
}
```

Figure 4.18: Serialization Method in Leaf page Class

4.4 Search Tree Index Framework Extension

As a sub-framework of Know-It-All, The search tree index framework is used to develop a generalized search tree that provides the basis for common tree access methods

used in database systems. To support the growing set of applications, the search tree framework must be extended for maximum flexibility.

4.4.1 Extensibility of Framework

C++ template gives us abilities to extension data type of search tree. By changing data type template parameters of a searching tree class, full data type extensibility of user defined access method can be achieved. For example, the B+-trees template can be used to index any data with a linear ordering, instead of developing new B+-tree structure for each new data types.

A modularization design with STL style offers our search tree framework good extensibility. There are two easy approaches to build a certain type of search tree. By understanding and generalizing the primary-memory data structure issue of searching tree from scratch, a developer can build a new search tree easily just as building a tree container, but is not required to care about memory management, physical storage management and access control between physical storage and memory. By understanding the core, reusable components of a series of search trees, a generalized search tree can be built and different search trees can be developed from the generalized components significantly faster than using the traditional approach.

The search tree index framework can cover one-dimensional tree structures such as B+-tree, point access structure, such as K-D-B-Tree, hB-Tree, special multi-dimensional structure, such as R-tree, R*-tree, SS-tree, X-tree. It can also fit sequential queries, exact match queries, range queries, approximate queries, and similarity queries.

Using our framework to implement a special tree is as simple as implementing a STL tree container; furthermore, to make tree container implementation as easy as possible, our framework offers two base tree policy containers and implement standard index operation: *find* which takes a predicate and returns all leaf entries satisfying that predicate; *insert* which adds a key/data reference pair to the tree; and delete, which removes such a pair from the tree. Two base containers implement these operations with the help of a set of external functions supplied by the access method developer. This set of external functions, which encapsulates the semantics of the data domain to be indexed and organization of predicates with the tree. So that implementing a new specific tree becomes as easily as implementing these external functions.

Specific search trees can be built from our index search tree framework significantly faster and more reliably than hand coding. As well, flexibility is provided for search tree developers to choose the kind and number of indices, the internal representation and implementation of the index page and leaf page, as well as the specific algorithms for particular operations.

Compared with the traditional approach, which is hand-coding from scratch or using GiST to implement a search tree access method, our framework have some significant advantages:

- **Easy to maintain and understand.** The search tree index framework applies the STL modularity concept in the analysis, architecture, design and interface. The search tree is designed as a STL container, which conforms to the standard interface of STL components. That makes our framework easy to maintain and understand because all designs have "standard" STL interface.

- **Have full data type support.** C++ template and generic programming techniques offers our index abilities of full data type support.
- **Extension is greatly simplified.** The framework can be adapted to the needs of any application by simply changing some of the building layer and is designed for maximum flexibility and the simplest extension. A generalized class library is created including a small set of specialized components so that a search tree file structure developer is not required to write all the components from scratch. By understanding the core, carefully selecting components from the library and only specializing components needed for a new algorithm/data-structure, a generalized search tree can be built. Also, different search trees can be developed from the generalized components significantly faster than by using the traditional approach.
- **Flexibility.** The framework has a flexible design by adopting a complete building block replacement policy. It isolates access control, memory and secondary storage management. The page container layout can be customized because our framework is designed by adopting STL style and each component of the Index Framework belongs to a group such as container group, iterator group and all functions of search tree are acted on container through iterator. Moreover, it has flexibility of user defined function extension, because containers use *find* and *find_if()* function doing search, any new user defined function can be integrated easily by providing predicate function.
- **Improved stability.** We build complete systems with the majority of its building blocks obtained from the STL library and take advantage of reuse capabilities in

modern programming languages such as generic programming and design pattern. Moreover, a search tree access method extension is implemented in term of a stable framework interface and need not rely on interfaces to some internal services, such as lock and log manager.

4.2 An Example of R-tree Extension

After constructing the generalized search tree and developing the specialization component library, the next goal is to demonstrate the feasibility of the generalization idea on database search trees. Here, we use our framework to extend an R-tree access method.

An R-tree can take advantage of the existing operations provided in R-tree policy container. First, we need to define the R-tree key class, and the data reference type. The data reference contains a value of the same type as the indexed data. We use long type as data reference type here. In the 2-dimensional domain, the keys in the tree are 4 numbers of type floats, representing the upper-left and lower-right corners of rectilinear bounding rectangles for 2d-objects. If keys are representing a point of 2d, the upper-left and lower-right has same value. Then, we can define the predicate class, leaf page and index page class by passing the Rectangle class as a parameter to them. The Predicate, LeafPage, and IndexPage class are defined as:

```
typedef Predicate<Rectangle> predicate;
typedef LeafPage<Rectangle,long,predicate> LeafPageType;
typedef IndexPage<Rectangle,long,predicate> IndexPageType;
```

Besides exact matching, the queries supported are point queries and window queries including containment, enclosure and intersection queries. We need provide these

predicate instances to the R-tree policy class. Moreover, we define some external function that need be accessed in the R-tree policy class.

The following is the implementation of the external extension methods:

- static bool Consistent(IndexKeyType, LeafKeyType): if key1 is contained in key2, then return true, otherwise return false;
- static IndexKeyType Union(PointerType): return a rectangle having the smallest lower-right corner and the biggest upper-left of all rectangles in the container;
- static PointerType ChooseSubTree(PointerType, LeafKeyType): It is used to choose a sub tree to insert a new record;
- static void PickSplit(PointerType, PointerType) splitting uses Guttman's quadratic splitting algorithm[GA84];

Performance Testing

Performance has always been a great concern for database index systems. Performance evaluation of an index system usually includes the following:

- Access Type: types of access that are supported efficiently, e.g., value-based search or ranges search.
- Search Time: time to find a particular data item or set of items.
- Insertion Time: time taken to insert a new data item or set of items (includes time to find the right place to insert.
- Deletion Time: time to delete an item or set of items (includes time taken to find item, as well as to update the index structure).
- Space Overhead: additional space occupied by an index structure.

We did the performance testing on a SUN SunFire 280R computer, with the Solaris 9 operation system, a 4G memory, 2 UltraSparc-III+ CPUs and the GNU g++ 3.2 complier. We used a data set provided by GiST. The data set contains 10,000 points as keys where each point is assumed to be in the unit square [0, 1000] that are randomly chosen. To do the test, we first set the size of a page to be 8Kb that is the size of a disk block of the testing computer. The cache buffer can hold at most 16 pages, which includes 14 index pages and two leaf pages. Then, we recompiled GiST v1.0 and our R-tree extension example. A page will contain at most 100 keys. The R-tree should have at least 101 pages in two levels. The following operations were involved in the test:

- Insert all the data of data set as keys into R-tree.
- Range query: find the data set with in [0,50]
- Exact match query: find an exact point.
- Delete all the element where in [0,50]

Table 4.2 shows the test results in average time (microseconds) of 20 tests under the same conditions. From the table, all experiments show that the R-tree developed from the Search Tree Index Framework has competitive performance compared with the GiST R-tree.

OPERATION TIME (Microseconds)	**KIA R-tree Index**	**GiST R-tree Index**	**Comparison (KIA/GiST)**
Insertion Time	3194332	5665792	0.56
Range Query Time	498.4	668.6	0.61
Exactly Match Query Time	158.6	394.4	0.40

Deletion Time	704.5	815.2	0.86

Table 4.2: Comparison of performance KIA R-tree and Gist R-tree

The KIA R-trees are slightly faster than GiST R-tree Index in insertion, deletion and query using GiST data set of points. One reason that our framework R-tree is faster than GiST R-tree is our cache strategy. We use a special modification First-In-First-Out cache algorithm in the Index framework. The page cache buffer is divided into two parts: an index pages part and leaves part. Both parts are controlled by an ordinary First-In-First-out discipline using a vector. The non-leaf part has higher priority to stay in cache memory than leaves part. Hence, many non-leaf pages may be cached in run-time and it will reduce access times significantly; whereas, GiST only caches one search path when it does any operation.

Chapter 5 Conclusions

In this thesis, we described our work, the Know-It-All search tree index framework for database management system. In addition, more general issues in framework based development were discussed.

The Know-It-All search tree index framework provides all the basic search tree logic required to build most tree based access methods. It applies the Standard Template Library modularity concept in the analysis, architecture, design and interface, and unifies distinct structures, such as B-trees and R-trees in to STL container. The design separates index, data, and data reference; uses iterators to perform general queries on search tree structures, while providing a clean interface for these queries to define both positions within indexes or files, as well as to refer to a collection of information. The tree predicates are able to support an extensible set of queries and provide a unified interface for the queries.

The framework covers one-dimensional tree structures such as B+-tree, point access structure, such as K-D-B-Tree, hB-Tree, special multi-dimensional structure, such as R-tree, R*-tree, SS-tree, X-tree. It also fits sequential queries, exact match queries, range queries, approximate queries, and similarity queries.

By using modularization design, the system is designed as an integrated set of layers including Container Presentation layer, Proxy layer and Physical Storage layer. A layer provides a mechanism to decompose functionality. Each of these layers, referred to as a building block, is well defined and has a clear set of functionality that is meant to carry out a specific task. The Container Presentation layer refers to some Search Tree STL containers, which implement some specialized index access methods to satisfy some

domain requirements. The Proxy Layer is used to manage access control page objects between secondary storage and memory. The Physical Layer takes responsibility to handle persistent data in secondary. The framework architecture gives expert developer and database developers the freedom to isolate and replace any one or more of these components with no or minimal impact on the system and can be adapted to the needs of any application easily, when they develop a new access method.

By taking advantage of reusability in modern programming languages such as generic programming and design patterns, a good quality framework of database search trees index is built. A framework library is created to include a set of specialized components so that a search tree file structure developer is not required to write all the components from scratch. By carefully selecting components from the library and only specializing components needed for a new algorithm/data-structure, a new tree access method can be created significantly faster than via the traditional approach.

Compared with the traditional approach of implementing a search tree access method, building a new search tree access method is greatly simplified by extending through our framework; moreover, the access methods extended from our framework are easier to maintain and understand.

An R-tree extension example are demonstrated by taking advantage of the existing operations provided in the R-tree policy container in the Container Layer of the index search tree framework The R-tree example's experiments demonstrate the feasibility and viability of the search tree index framework to achieve efficient implementation of a search tree access method.

Future Work

This thesis concentrates on the building a framework to extending search tree access methods. To further test the performance and viability of our framework, more search tree access method should be implemented, such as B+-tree, SS-tree. Moreover, more work is needed to explore the idea on other database access methods, such as hashing techniques. Furthermore, we need to build tools to help people monitor the size, and performance of indexes in order to help people design indexes for new applications.

Bibliography

[AA01] Andrei Alexandrescu, "**Modern C++ Design: Generic Programming and Design Patterns Applied**", Addison Wesley Professional, 2001

[BCC02] Greg Butler, Ling Chen, Xuede Chen, Ashraf Gaffar, Jinmiao Li, Lugang Xu, "**The Know-It-All Project: A Case Study in Framework Development and Evolution, Domain Oriented Systems Development: Perspectives and Practices**", Kiyoshi Itoh, Satoshi Kumagai, T. Hirota (eds), Taylor and Francis Publishers, UK, 2002.

[BEJ99] Grady Booch, James Rumbaugh, Ivar Jacobson. "**The Unified Modeling Language User Guide**", Addison-Wesley, 1999.

[BG93] Grady Booch "**Object-Oriented Analysis and Design with Applications**" Addison-Wesley Pub Co; 2nd edition, October 1993

[BKK96] Stefan Berchtold, Daniel A. Keim, Hans-Peter Kriegel, "**The X-tree: An Index Structure for High-Dimensional Data**", Proceedings of the 22nd International Conference on Very Large Databases 1996 pages 28--39.

[BKS90] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. "**The R*-tree: An Efficient and Robust Access Method for Points and Rectangles**", In ACM SIGMOD International Conference on Management of Data, pages 322–331, 1990.

[BM72] R.Bayer, E.McCreight, "**Organization and Maintenance of Large Ordered Indexes**", Acta Informatica. Vol. 1, Fasc. 3, 1972, pp173-189.

[DC79] D. Comer. "**The Ubiquitous B-tree**" ACM Computing Surveys, 11(2): 121–138, 1979.

[DK73] D. Knuth. "**Sorting and Searching**", In *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973.

[FH97] Gary Froehlich , H. James Hoover , Ling Liu , Paul Sorenson, "**Hooking into Object-Oriented Application Frameworks**", Proceedings of the 19th international conference on Software engineering, p.491-501, May 17-23, 1997, Boston, Massachusetts, United States

[FS97] Mohamed Fayad and Douglas C. Schmidt, "**Object-Oriented Application Frameworks**", Communications of the ACM, Special Issue on Object-Oriented Application Frameworks, Vol. 40, No. 10, October 1997.

[FS99] M. Fayad, D. Schmidt, and R. Johnson (eds). "**Building Application Frameworks: Object-Oriented Foundations of Framework Design**", John Wiley and Sons, New York, September 1999.

[FZ92] Michael J. Folk, Bill Zoellick, "**File Structure**", Addosion Wesley, 1992

[GA01] Ashraf Gaffar. "**Design of a Framework for Database Indexes**", Master thesis, Department of Computer Science, Concordia University, 2001

[GA84] Antonin Guttman. "**R-trees: A Dynamic Index Structure for Spatial Searching**", In ACM SIGMOD International Conference on Management of Data, pages 47–54, 1984.

[GG98] V. Gaede, O. Günther, "**Multidimensional Access Methods**", In ACM Computing Surveys, Volumn 30, Number 2, June 1998.

[GHJ95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. "**Design Patterns: Elements of Reusable Object-Oriented Software**". Addison-Wesley, 1995

[HNP95] J. M. Hellerstein, J. F. Naughton, and Avi Pfeffer. "**Generalized Search Trees for Database Systems**", Proceedings of the 21st International Conference on Very Large Data Bases, Zurich, Switzerland, 1995, pages 562-573

[HSW88] A. Hutflesz, H. W. Six, and P. Widmayer. **"Globally Order Preserving Multidimensional Linear Hashing",** Proceedings of 6th IEEE International Conference on Data Engineering, pages 572–579, 1988.

[HSW89] Andreas Henrich, Hans-Werner Six, and Peter Widmayer. "**The LSD Tree: Spatial Access to Multidimensional Point and Non Point Objects**", in Peter M.G. Apers and GioWiederhold, editors, Proceedings of the International Conference on Very Large Data Bases, Amsterdam, Netherlands, 1989, pages 45 53. 16

[HSW90] A. Hutflesz, H. W. Six, and P. Widmayer. **"The R-file: An Efficient Access Structure for Proximity Queries",** Proceedings of 6th IEEE International Conference on Data Engineering, pages 372–379, 1990.

[JTB81] J. T. Robinson. "**The K-D-B-tree: A Search Structure for Large Multi-dimensional Dynamic Indexes**". In Proc. ACM SIGMOD Int. Conf. on Management of Data, pages 10--18, 1981.

[KS97] N. Katayama, S. Satoh, "**The SR-Tree: An Index Structure for High Dimensional Nearest Neighbor Queries",** SIGMOD Conference, 1997, pages 369—380

[LRP95] Ted Lewis, Larry Rosenstein, Wolfgang Pree, Andre Weinand, Erich Gamma, Paul Calder, Glenn Andert, John Vlissides, Kurt Schmucker, "**Object-Oriented Application Frameworks**", Prentice-Hall, 1995

[LS901] David B. Lomet and Betty Salzberg. "**A Robust Multi-attribute Search Structure**", IEEE Conference on Data Engineering, pages 296–304, 1989.

[LS902] David B. Lomet and Betty Salzberg. "**The hB-tree: A Multiattribute Search Structure**", ACM Transactions on Database Systems, 15(4):625–658, 1990.

[MF87] M. Freeston. **"The Bang File: A New Kind of Grid File",** ACM SIGMOD International Conference on Management of Data, pages 260–269, 1987.

[NHS84] 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.

[RSB98] Dirk Riehle, Wolf Siberski, Dirk Baumer, Faniel Megert, and Heinz Zulighoven, "**Serializer",** Pattern Languages of Program Design 3, chapter 17. Addison-Wesley 1998

[SL95] Alexander Stepanov and Meng Lee, "**The Standard Template Library**", 1995, Incorporated in ANSI/ISO Committee C++ Standard.

[SRF87] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. "**The R+-tree: A Dynamic Index for Multi-dimensional Objects**", In Peter M Stocker and William Kent, editors, Proceedings of the Thirteenth International Conference on Very Large Data Bases, pages 507–518, Brighton, England, 1987.

[TW94] Taligent Inc., "**Building Object-Oriented Frameworks**", A Taligent White Paper, 1994.

[WJ96] D. A. White and R. Jain, "**Similarity Indexing with the SS-tree**", Proc. of the 12th Int. Conf. on Data Engineering, New Orleans, USA, pages 516--523, Feb. 1996.

[WP94] W. Pree. "**Design Patterns for Object-Oriented Software Development**", Addison-Wesley, 1994.