DESIGN OF A FRAMEWORK FOR DATABASE INDEXES

ASHRAF GAFFAR

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

AUGUST 2001

Contents

List of Tables v

List of Figures vi

CHAPTER 1 INTRODUCTION 1

1.1 THE PROBLEM 1
1.2 OUR WORK 3
1.2.1 The Domain 4
1.2.2 The Programming Language 5
1.3 CONTRIBUTION OF THE THESIS 7
1.4 LAYOUT OF THE THESIS 8

CHAPTER 2 THE STL 9

2.1 STL BUILDING BLOCKS 9
2.1.1 Containers 10
2.1.2 Container Adaptors 13
2.1.3 Template Parameters 13
2.1.4 Iterators 15
2.1.5 Iterator Traits 16
2.1.6 Algorithms 17
2.1.7 Functors 20
2.1.8 Function Adaptors 20
2.1.9 Allocators 23
2.2 BUILDING BLOCKS CONNECTIVITY 26
2.2.1 Iterator Categories 26
2.2.2 Container Categories 28
2.2.3 Algorithm Categories 30
2.2.4 A Blue Print for the Connectivity 33

CHAPTER 3 DATABASE INDEXES 35

3.1 INTRODUCTION 35
3.2 INDEX TREE APPLICATIONS 37
3.3 B+ TREE INDEX EXAMPLE 37
3.4 GENERAL DOMAIN APPLICATIONS 38
3.5 SIMILARITY SEARCH APPLICATIONS 41
3.5.1 Extracting keys: Feature Vectors 44
3.5.2 Comparing Keys: Distance Functions 45
3.5.3 Searching the Database: Query Processing 45
3.5.4 Building an Index to Speed up Query Processing 46
3.5.5 Traversing an Index 47

CHAPTER 4 CONTEXT ANALYSIS 49

4.1 CONTEXT USE CASES 49
4.1.1 Actors 50
4.1.2 Use Cases 50
4.2 DETAILED USE CASES 51
4.3 THE DATABASE DATA MODEL 55
4.3.1 The Index Data Model 55

CHAPTER 5 THE GENERAL DESIGN 58

5.1 DESIGN RATIONALE 58
System Modularization 58
Adopting the STL Concept 58
Code Reuse 58
Interface Reuse, and Combinatorial Composition 59
Implementation Independence 59
Efficiency 59
System Flexibility 60
Wide Rage of Complexity 60
Default Values for Simplicity 60
Favoring Templates over Inheritance 60
User-friendly, Readable Code 61
Ease of Modification 61
Sustained Code Quality 61
5.2 SYSTEM STRUCTURE 62
5.2.1 The Basic Components 62
5.2.2 The Class Diagram 65
5.3 THE DESIGN DYNAMISM: MODULES REPLACEMENT SCHEME 65
5.4 EXCEPTION HANDLING 68
5.5 THE GENERAL SYSTEM INTERFACE 70
5.5.1 The Page Container 71
5.5.2 The Leaf Page Interface 72
5.5.3 The Index Tree Container 73

CHAPTER 6 DESIGN FOR LINEARLY ORDERED DOMAIN 76

6.1 THE CONCEPT OF LINEARLY ORDERED DOMAIN 76
6.2 CLASS DIAGRAM 78
6.3 SYSTEM BEHAVIOR 80
6.3.1 Scenario for “search” Using an Internal Query 82
6.3.2 Scenario for “search” Using an External Query 87
6.3.3 The Cursor Behavior: Iterating Through the Search Result 90
6.3.4 The Activity Diagram for Insertion 92
6.3.5 The Activity Diagram for Deletion 93

CHAPTER 7 DESIGN FOR GENERAL DOMAIN 94

7.1 THE DESIGN 94
7.2 THE SYSTEM LAYOUT 95

7.3 SYSTEM FLEXIBILITY; USING STACK OR QUEUE 97
7.3.1 The Effect of Stack and Queue on Index Behavior................97
7.4 THE CLASS DIAGRAMS...98
7.5 THE BEHAVIOR OF THE SYSTEM 100
CHAPTER 8 SIMILARITY SEARCH APPLICATIONS......................104
8.1 THE ANALYSIS 104
8.2 A NUMERICAL EXAMPLE...107
8.3 THE ORDER OF READING LEAF PAGES.....................................108
CHAPTER 9 CONCLUSION 110
9.1 FUTURE WORK...111
BIBLIOGRAPHY 113
APPENDIX A FLOW OF EVENTS 118
APPENDIX B INTERFACES..120

LIST OF TABLES

TABLE 2.1: COMPARISON BETWEEN DIFFERENT ACCESS TYPES................12
TABLE 2.2: ITERATOR OPERATIONS 15
TABLE 7.1: COMPARISON BETWEEN INDEX TRAVERSAL POLICIES.............98

LIST OF FIGURES

FIGURE 2.1: THE CONTAINER AND TEMPLATES......................................14
FIGURE 2.2: THE ITERATOR INTERFACE...16
FIGURE 2.3: SOME STANDARD ITERATOR ASSOCIATED TYPES 17
FIGURE 2.4: AN STL FILL ALGORITHM 18
FIGURE 2.5: USING SAME ALGORITHM FOR DIFFERENT CONTAINERS.........19
FIGURE 2.6: FUNCTIONS WITH DIFFERENT FUNCTOR TYPES.....................21
FIGURE 2.7: PASSING MULTIPLE CONTAINERS TO A FUNCTION 22
FIGURE 2.8: USING THE FUNCTION MANIPULATECONTAINERS..................23
FIGURE 2.9: THE GENERAL SYSTEM LAYOUT 25
FIGURE 2.10: ITERATOR CATEGORIES 27
FIGURE 2.11: CONTAINER CATEGORIES...30
FIGURE 2.12: SOME ALGORITHMS 32
FIGURE 2.13: ALGORITHM CATEGORIES 32
FIGURE 2.14: A BLUE PRINT FOR CONNECTIVITY 34
FIGURE 3.1: B+ TREE 38
FIGURE 3.2: FINDING THE KEY FOR PARTITIONS OF POLYGONS................39
FIGURE 3.3: SEARCHING DATA USING A KEY TO FIND MATCHING KEYS 41
FIGURE 3.4: THE SUBJECTIVITY IN SIMILARITY DECISIONS 43
FIGURE 3.5: THE SS-TREE IN SPATIAL PRESENTATION............................48
FIGURE 4.1: CONTEXT USE CASE DIAGRAM 49
FIGURE 4.2: DETAILED USE CASES 53
FIGURE 4.3: THE DATA MODEL...56
FIGURE 4.4: THE REFERENCE FILE MODEL 57
FIGURE 5.1: THE COMPONENTS LAYOUT 63
FIGURE 5.2: DETAILED SYSTEM LAYOUT 64
FIGURE 5.3: THE SYSTEM MAIN CLASSES..65
FIGURE 5.4: COMPONENTS REPLACEABILITY...66
FIGURE 5.5: DETAILED VIEW OF THE MAIN INTERFACES 67
FIGURE 5.6: THE EXCEPTION CLASS DIAGRAM.......................................69
FIGURE 5.7: PAGE CLASS SIGNATURE 72
FIGURE 5.8: THE LEAF PAGE 72
FIGURE 5.9: THE QUERY, CURSOR, AND INDEX CLASSES 75
FIGURE 6.1: THE NUMBERING CONVENTION..81
FIGURE 6.2: HI-LEVEL COLLABORATION DIAGRAM 84
FIGURE 6.3: HI-LEVEL SEQUENCE DIAGRAM 84
FIGURE 6.4: USING INTERNAL QUERY 85
FIGURE 6.5: USING EXTERNAL QUERY 88
FIGURE 6.6: CURSOR ITERATION THROUGH RESULT 91
FIGURE 6.7: INSERTION ACTIVITY DIAGRAM...92
FIGURE 6.8: DELETION ACTIVITY DIAGRAM 93
FIGURE 7.1: THE INDEX TRAVERSAL FOR GENERAL DOMAIN...................95
FIGURE 7.2: SYSTEM LAYOUT FOR GENERAL DOMAIN INDEX 96

FIGURE 7.3: OBJECT DIAGRAM FOR USING INTERNAL QUERY 99
FIGURE 7.4: SEQUENCE DIAGRAM WHEN USING INTERNAL QUERY 102
FIGURE 8.1: SEARCH SEQUENCE IN SIMILARITY SEARCH........................ 109

Chapter 1 Introduction

A database is not about storing away archive data; it is more about searching and updating live, dynamic data. Database indexes are the search engines for the database, and therefore constitute an important part of any database management system. They are a means for the database management system to deal with the data. From the application perspective, however, the index is invisible.

1.1 The problem

An efficient index is a fundamental requirement for good performance in a database. An accurate, fast index will result in a good quality, quick-responding database. An investment in an efficient index to the database is always a good idea. That is why in many commercial systems, a specialized handcrafted index is the classical choice to achieve the maximum efficiency possible. A specialized index is an index that supports a specific database application in a specific domain using predetermined structures and access methods as well as data types and queries. A specialized index for each new application is generally better for code efficiency and performance. The tradeoff is a lot of development time, and cost, normally affordable only by large corporations. Another choice is to make 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. Sometimes, however, frameworks provide applications that have less efficient performance than a specialized application. J. M. Hellerstein explains in [HNP95] that the use of their framework for database indexes does not always provide an efficient lookup.

Early frameworks exist that apply this idea. They generate flexible code with some hot spots that can be easily adjusted to develop different applications but they result in a generic source code that is often complex and not easy to read or modify; characterized by many as having a steep learning curve. The reason is that in one source code, the framework is meant to satisfy all the possible needs of the future members of a family of application at one time. We can see this methodology as concentrating the complexity of future expansions in one stage; the source code. That is one reason why the resultant source code is often less user-friendly than a well-written specialized code. Cost-wise, however, it is a very good alternative.

In some cases, as part of their evolutionary lifecycle, frameworks themselves need some maintenance work. A typical case of modification is by adding new features to the framework to serve a larger family of applications than the original framework was meant to serve. That usually means adding further code even to the cold spots, or frozen spots in the source code, which were not meant to be touched in the original framework. This should lead to another framework that covers more applications than the original one. As a side effect, however, it also often leads to an even more generic code that is bigger and less user-friendly. The reason is the same, namely concentrating the complexity of evolution in one stage; the source code. As this trend progresses, the framework, going in its normal lifecycle, grows bulkier and less user-friendly with more patches added to it. In the end, time comes when it is necessary to recycle some ideas from this framework and use them to develop a new framework to replace it.

The Generalized Index Search Tree, *GiST* [HNP95] is an example. GiST is an existing framework of a generalized index system. It has friendly hot spots that can be adapted to different key types and access methods. The rest of the source code, however, is meant to be black box so it is not user friendly. GiST tried to address all possibilities in the same piece of code, so it ended up with a large source code that is 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. Despite the efforts to cover all cases, the original GiST did not cover the similarity search trees so it was added later to it [Aok98] as part of the framework evolution cycle. This, however, made the source code even more unreadable as the modification was spread all over the original code. Another addition to visually fine-tune the access method, the amdb [SKH99] was also added later on. This made GiST more useful, but it almost doubled the size of the already large code. In general, we can see that the additions to the framework, albeit useful, had disadvantages with them.

As the framework development methodologies improve, these problems are being recognized and addressed. The load on the source code should be diffused; i.e. the complexity of expansion should be spread to the earlier stages of framework development, following the motto *the earlier the better*. This would help getting closer to the goal of producing frameworks that achieve the advantages of both worlds: the specialized code with efficiency, understandability, and maintainability, and the framework-reusable code with its inherited advantages of efficient time and cost utilization. On the evolutionary side, being able to modify the early stages of the framework, like its architecture and design as well as the code, in a constant process, could mean a better evolution as the framework can be easily redeveloped instead of recycled.

1.2 Our Work

Frameworks have a good potential for improving the process of developing good quality software. They combine the experience of expert programmers (most likely not domain experts) with that of domain experts (most likely not expert programmers) into a mass production environment for applications. They allow for producing several applications using virtually the same time, money and experience investment, thus the cost per application is cheaper as the total development cost is divided between several applications.

While mass production greatly helps make the application development affordable, it limits the freedom of choice a developer has [BCC+02]. Ford, the father of mass production technique in automobile industry, reduced the cost per car several folds using early, ad hoc mass production methods. Nonetheless, they had their concept of freedom of choice as *you can have any color you want in your new car, as long as it is black*. As the mass production concept was researched and improved, it now has better options, better quality, and better products that rival the custom-made ones at a fraction of the cost per application.

1.2.1 The Domain

Butler is leading a research group in framework methodologies and development for scientific applications. It is aimed at adopting and improving frameworks as a technique to produce good quality software. He is interested in framework development and evolution methodologies that make for better frameworks.

A better framework would:

-Produce good quality software (by combining effort of domain experts with that of expert programmers)

-Allow non-experts to easily develop professional application (most of experience is already in the framework)

-Promote code reuse to reduce time and cost and enhance quality.

-Produce flexible applications that are user-friendly and modification-friendly, not only in final stage (source code) but rather in all development stages.

-Be user-friendly and modification-friendly with well-defined development, documentation, application and evolution lifecycle.

On the domain front, Butler sees that "*The research on software methodology in an academic setting needs a concrete case study in order to evaluate the methodology and models*" [BCC+02].

He is interested in the database domain as a case study to validate the methodology. This resulted in the ongoing development of the Know-It-All (KIA) project, a "*framework for DBMS that support a variety of data models of data and knowledge, the integration of different paradigms and heterogeneous databases*" [BCC+02].

The KIA project involves multiple subframeworks that are integrated together in an adaptable DBMS context. It started by supporting the traditional relational database model, and it is expanding to support other types of database applications using different data models. Eventually, it will be applied to advanced applications in bioinformatics.

1.2.2 The Programming Language

The language of the source code is essential in the development process. In the end, the framework needs to be implemented using a programming language. Designing for the interface rather than for the implementation is always a good choice to follow, making the design implementation independent which allows for more freedom in the design. The programming languages are nonetheless more than just implementations. Concepts and features of some programming languages like inheritance and polymorphism can and should be reflected on the design stages to improve the quality. In other words, selecting a suitable programming language to do the job can certainly affect the quality of the software. Dr. Butler selects the C++ as one of the languages to write the implementations of different subframeworks of the KIA project.

As Stroustrup [Str94] quotes Alexander Stepanov summarizing the experience of writing and using a major library of data structures and algorithms:

> "*C++ is a powerful enough language – the first such language in our experience – to allow the construction of generic programming components that combine mathematical precision, beauty, and abstractness with the efficiency of non-generic hand-crafted code.*"

It is worth mentioning that this *major library of data structures and algorithms* mentioned by Stroustrup was the STL, designed by Alexander Stepanov and Meng Lee. The ANSI/ISO committee later recognized it as part of the standard library. STL is a library of high quality and efficiency with great emphasis on code reuse, certainly good credentials influenced by the language of implementation of that library, the C++.

The ever-evolving programming languages are introducing new features to improve their capabilities. The C++, one of the popular languages is no exception. The recent adoption of an ANSI/ISO standard for the C++ language with the STL library as part of the standard library has many advantages for the programming community. STL makes extensive use of templates, and favoring them over the traditional inheritance, casting and void* has brought advantages with it regarding code modularity and performance efficiency.

Templates

Collins dictionary defines template as a piece of metal or plastic cut into a particular shape and is used to reproduce the same shape many times. Webster's defines it as a mold used as a guide to the form of a piece being made. While this concept can apply to frameworks themselves, being seen as templates for design ruse, templates can also be adopted to the lower stages like source code, allowing for code reuse. Instead of writing the same code several times with several similar pieces, we can factor out the similarity of these pieces into one *template*, and write the code once only using this template. Later we can use the same code many times by replacing the template with actual pieces of the same shape. This saves on the amount of code to be written (code reuse) and improves run-time performance (efficiency).

As Stroustrup [Str94] explains

"The template mechanism is completely a compile-time and link-time mechanism. No part of the template mechanism needs run-time support".

1.3 Contribution of the Thesis

This thesis introduces a design of a generalized index framework, a subframework of the KIA project. The generalized framework is capable of producing tree-based indexes that are adaptable to different data / key types, different queries, and different database application domains. We produce several models of the index framework; analysis, architecture, design and interface, and we apply them to different index applications and show that the development of each of these applications is reflected in all models. Developing a new application using this framework starts at the earliest stage; the requirements analysis, by reusing the existing analysis to determine the requirements for the new application. This will determine the necessary changes or replacement in the system architecture. We obtain the new architecture from an existing one by applying the necessary changes mapped from the analysis. The new architecture will help us identify the necessary changes in the design. We map these changes to the corresponding design models (structural and behavioral) and obtain a new design. As we get a new design, it will help us map the changes from it to the interface.

To follow these steps, we build a modular framework by applying the STL modularity concept in the analysis, architecture, design and interface stages. We develop a new set of models for an index in different application domains; linearly ordered domain, general domain, and eventually similarity search domain. This will produce the reusable models we need for all the development stages of the framework, not just for the source code. Using coherent, decoupled building blocks allows us to locate and replace certain blocks in each model to obtain a new model with no major impact on the rest of the model. This makes modifications limited to specific regions of the system. A wealth of off-the-shelf STL modules can be used in the replacement process. The framework also allows for introducing new compatible modules as needed.

In the end we can have a new index (and thus a new source code) for each new application. Each code would be a pseudo-specialized source code that is more adjusted to its specific application than one generic code meant to serve all the possible applications.

1.4 Layout of the Thesis

Chapter 1 gives an introduction with an overview of the work of the research group. Chapter 2 presents a development environment analysis of the STL modularization concept and how to adopt its building blocks approach to decouple system modules. Chapter 3 gives an example of database indexes. Chapter 4 provides a context analysis of using an index system in a DBMS environment, and the relation between an index and both its upper level user; the application, and its lower level server; the database. Chapter 5 includes the design rationale and the general analysis, architecture, and design of the index system. We then provide examples of adapted systems for each of linearly ordered domain (chapter 6), general domain (chapter 7), and similarity search database (chapter 8). Chapter 9 gives a summary and future work.

We assume the reader is familiar with the Unified Modeling Language (UML) [BRI99], and the concepts of object-oriented design [Bch94].

Chapter 2 The STL

The Standard Template Library, STL, is not just another C++ library among many libraries available today; it is a rather important addition to the C++ programming community. One reason is in the word “Standard”. The ANSI/ISO standard committee accepted the STL as part of the C++ standard library (ISO/IEC 1998). It is “A particularly carefully constructed library”, [Bry00]. Another technical reason is in the word “Template”. The STL is using the template mechanism of the C++ language and extends it to new dimensions by using interoperable components concept. As Matthew Austern [Aus99] states, “*Computer programming is largely a matter of algorithms and data structures*”. The STL is based on separating algorithms from data structures as two different types of generic building blocks. It also introduces other generic building blocks (e.g. iterators, functors, container adaptors) to work as adaptors or glue to allow for building a large system using a combination of these blocks. This emphasizes code reuse. STL allows for new building blocks to be written and put to work with the existing ones, thus emphasizing flexibility and extendibility. This makes the STL particularly adaptive to different programming contexts including algorithms, data structures, and data types.

Part 2.1 introduces some of these building blocks with brief examples. Part 2.2 shows how to connect them together to build a useful system.

2.1 STL Building Blocks

The STL building blocks are meant to be largely independent of each other. They have different types, and a block of one type can connect and work correctly with other blocks of the same- or different types, giving an endless number of possible correct combinations.

This section introduces some of these blocks with brief examples to demonstrate their context.

2.1.1 Containers

A container is a common data structure that stores a group of similar objects, each of which can be a primitive data type, a class object, or –as in the database domain- a data record.

The container manages its own objects: their storage (see Allocators) and access (see Iterators). The stored objects belong to the container and are accessed through it. Each container provides a set of public type members, data members and member functions to provide information, and facilitate the access to its elements. Different container types have different ways of managing their objects. In other words they offer different sets of functionality to deal with their objects. Storing an object requires it to have some identity to help access it back. In computer science, objects are identified by their physical existence, so two exact copies of the same object are considered as two different objects occupying two different areas in the memory with two different identities. Object database follow the same concept by concentrating on objects as the elements, thus allowing for two similar copies to be stored as two different objects. In relational database, on the other hand, objects are identified by their contents, so two exact copies of the same object are considered as one object occupying one area in the storage with one identity. This is mainly because relational model is concerned with storing data, rather than objects, so it is considered redundant to store two copies of the same data. For simplicity, we will consider the first storage concept only (the object concept) as a general concept. The relational database concept can be considered in a larger, more specialized discussion. Containers can be classified according to the way they store and allow access to their objects into sequential containers and associative containers. They allow access to their elements either in a sequential or a random access method.

Tree containers are more complex data structures that can be build using some of the STL containers. Tree containers have internal structure that is controlled by the design of the tree, and they are built in multiple levels. Access methods in tree containers are also more complex than the other, one-level containers. Each access request by the user typically involves more than one element of the tree. We can consider this type of access as automatic access since it progresses internally from one tree element to the other without user intervention. We can build database indexes as tree containers. Details of these structures are provided in the design of database index containers, chapter 5.

Table2.1 shows a brief comparison between different access types for the containers.

Access Category	**Container Category**	**Number of Levels**	**Access Mechanism**
Sequential access	Sequence container	One	Container knows physical ID of one element. Each element knows physical ID of one other element (next element). No logical ID. User can advance to next element only. Current element translates it to one physical ID.
Indexed random access	Sequence container	One	Container knows physical ID of all elements. Each element is donated logical ID equals its position. Container binds positions with Physical ID's. User can directly go to any position. Container translates it to one physical ID.
Keyed random access	Associative container	One	Container knows physical ID of all elements. Each element has intrinsic logical ID (a key). Container binds keys with physical IDs. User asks for any key. Container translates it to one/more physical ID.
Automatic access	Tree container	Multi level	Each element has logical ID; a key. Container knows physical and logical ID of some elements. Each element knows physical and logical ID of some more elements. User asks for any key. Container translates key to some elements, each of which further translate same key to other elements. This continues until reaching terminal elements.

Table 2.1: Comparison between Different Access Types

2.1.2 Container Adaptors

Containers have some characteristics (associative, or sequential containers with random or sequential access, etc.), which will determine the functionality offered by the container to the user (how to access its elements, how to insert or delete an element, and even how to name an element).

Sometimes the user might be interested in only one part of this functionality; the rest is not to be used at all. In this case we might want to use the original container while leaving the unused functions in it, or we might want to mask them from the user, it accidental use of them could cause problems. In this case we may use a container adaptor to help mask the unused functions. It will have an interface to the user offering only those functions the user is allowed to use; the adaptor then simply translates them to the ones offered by the original container. The rest of the unused functions will never be called, as the adaptor does not allow the user to access them. Another use of container adaptor is when the functions used by the user have different names that those offered by the container. Instead of changing the signature (the name or argument or both) of the functions used by one party (the user or the container) to accommodate the other, an adaptor can be used for this purpose. It takes the user call to a particular function and delegates the call to the suitable function(s) in the container.

2.1.3 Template Parameters

Containers are made independent of the types of the objects they store by using template parameters. Each container type is implemented only once using a general template parameter as the type of objects stored in it, allowing the programmer to completely implement the desired functionality of the container without knowing the type of objects that will be stored in it.

```
template <class T> class vector {
                                        //implementation in terms of template T as
                                        //the generic type for the stored elements
                                          }
```

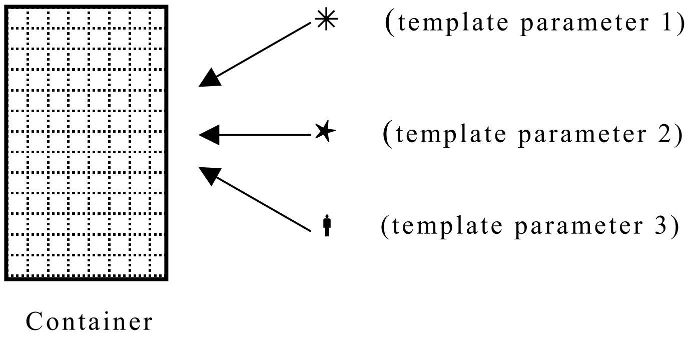


Figure 2.1: The Container and Templates

Later on when the container is to be used, an instance of it is created and passed the type of objects to be stored in, be it an *int*, a *double*, or a complex user defined type. This parameterization allows for code reuse by making the same container code usable many times. Different container object of the same class can be instantiated and passed - as template parameters- different types of objects to store.

```
vector <int> intVec ;
vector <student> studentListVec;
```

The programmer has different container types (with different functionality) to select from, according to the application needs.

2.1.4 Iterators

Containers do not allow direct access to their stored objects, but rather through another class called an Iterator. The Iterator is an object that can reference an element in a container. It allows programmers to visit each element in a sequential or random (indexed) way depending on the type of the container.

X++ ++X X-- -- X	Increment / decrement
X + n X – n	Move n elements forward / backward

X = Y
X+=n
X - =n
	Assign to / from an iterator

X-Y	Distance between two iterators
X = = Y X != Y X>Y X <Y X>= Y X <=Y	Logical comparisons
X [n] *X	Access

Table 2.2: Iterator Operations

This is another technique that adds to the concept of code reuse. Programmers do not need to know the exact interface to different containers to use them. They only need to know one common interface; that of the iterator. Different container types are readily accessible through the same piece of code, since all containers support iterators that have the same interface. This allows for changing the container type without significantly changing the code that is using the containers.

Iterators support some or all of the following operations, depending on the container they are accessing. Table 2.2 lists these operations associated with two iterators X, Y and an integer n.

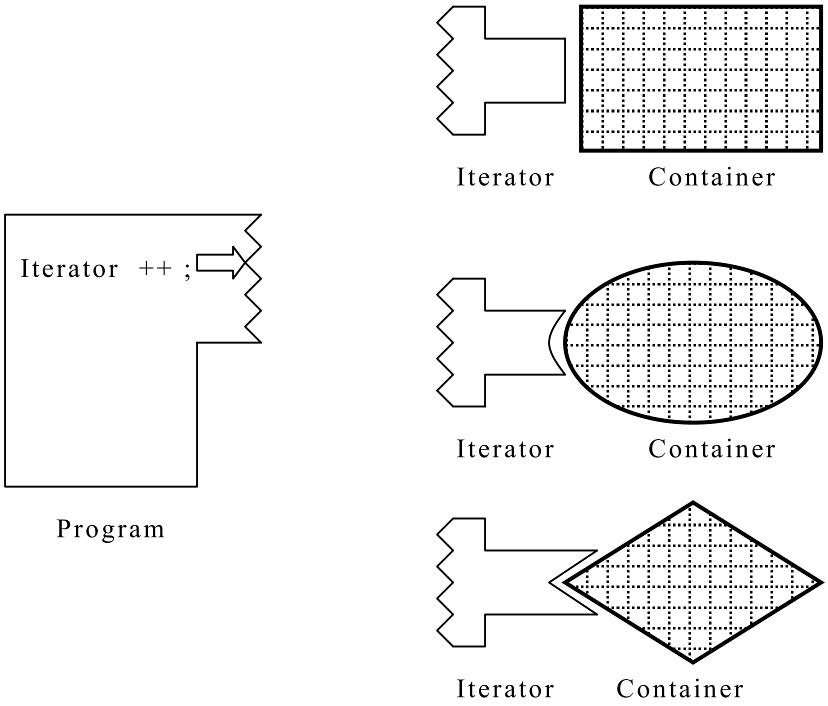


Figure 2.2: The Iterator Interface

2.1.5 Iterator Traits

The STL iterators have become a standard way of dealing with containers. Each iterator type posses a standard set of characteristics that completely describe it. Some of these characteristics are a set of types associated with each specific iterator type. A function, when passed any iterator through its argument, should be able to extract the different type information associated with that particular iterator. For example, an

iterator points to an element, and a function that receives an iterator should be able to find out the type of that element.
Some of these types are shown in figure 2.3.

```
value_type;          //The type of the element the iterator points to.
difference_type;     //A signed value indicating the distance between
                     //two iterators
pointer;             //Pointer to the type_value of the iterator.
reference;           //Reference to the type_value of the iterator.
iterator_category    //A tag telling the category of the iterator.
```

Figure 2.3: Some Standard Iterator Associated Types

2.1.6 Algorithms

Iterators separate the logic from the container types and element types stored in them. The same algorithm, written once, can be applied to different containers with different stored elements by using iterators in the algorithm code. This allows for a complete implementation of generic algorithms without knowing the exact type of the container (its functionality) or the type of elements stored in it.

This is achieved by writing the algorithms to deal with a standard Iterator interface common to all containers. Later in the program, these algorithms can be imported and added to the existing code and used directly without modifications, assuming that the existing code is using the Iterator-Container-Template building concept. This is a further contribution to the code reuse in importing and using already written algorithms in our code.

The algorithm shown in figure 2.4, which is already implemented in STL, will fill any container with any value of type T. All it needs is a standard iterator (of type ForwardIterator) that refers to some container, an object value of type T to use in filling the container, and a compatibility relationship between T and the type stored in the container.

```
template <class ForwardIterator, class T>
void fill (ForwardIterator   first,
          ForwardIterator   last,
          Const T&          value) ;
```

Figure 2.4: An STL fill Algorithm

Later in our code we can use this algorithm to fill a specific container with a suitable type of elements. By suitable we mean that the type T of the value must be assignment-compatible with the type of elements the container stores. This is because the algorithm will perform the assignment operation from value into container elements. For an algorithm to access the elements of a container, it needs to find its beginning and end positions (in fact the beginning and the past-the-end positions, an STL standard, denoted by the [a,b) notation which means the range from a to b including a and excluding b) or more generally any starting and ending position of a sub range within the container where the algorithm will be allowed to work. In the above algorithm, we see that it is allowed to work within the range [first, last). For some data structures, like an array object *Arr* of n elements, it is easy to find its beginning address, which is the array name *Arr* itself, and the past-the-end address, which is the address *Arr+n*. For STL containers, however, these positions cannot, in general, be easily found. Therefore each container must provide two methods begin( ), and end( ) that return an iterator to locate its first and past-the-end positions respectively. They can then be assigned to other iterators or passed directly to the algorithm when using it as in the following code.

```
vector<int> myVector(100); //creates a vector of 100 integers
fill (myVector.begin ( ), myVector.end( ), 1);
```

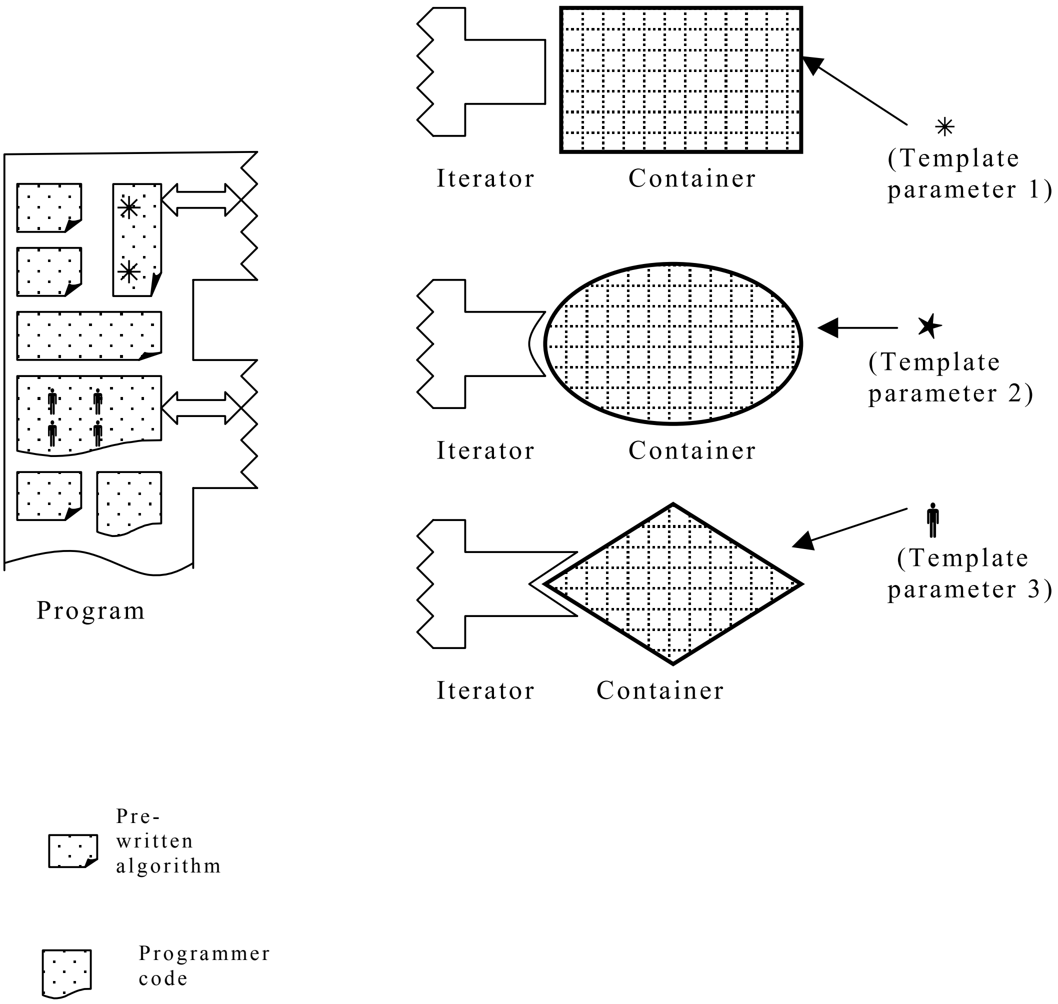


Figure 2.5: Using Same Algorithm for Different Containers

By following the STL standard process for templates, containers, iterators, and algorithms when building a database index, it will be easy to extend the code to support new data types, new queries, new access methods and produce a readable standard code with some off-the-shelf modules easily integrated into the code. The STL comes with a large number of generic algorithms that can be used in implementing an index to provide the needed functionality for searching or updating the database. More algorithms can then be added to increase its capabilities. Figure 2.5

sketches the concept of using different algorithms with different containers.

2.1.7 Functors

Function objects, or *functors,* are generalization of function pointers [Rob00], the same way as iterators are generalization of pointers. They add more functionality to them, making them much more powerful and conformant to STL specifications for building blocks. Function pointers can be bound to one of many functions at compile- or runtime depending on a condition (like user input for example). They are then used to perform the action they were bound to by overloading the *operator ( )* (the call operator) on them.

2.1.8 Function Adaptors

Some functions accepted two parameters in their argument and a compatible binary functor (that represent a binary operation). The function will call the functor, pass its two parameters to it as operands, and do something with the returned value from the functor. For a function that accepts a functor of type BinaryPredicate, an implementation would look like the one shown in figure 2.6.

There are other types of functors defined in the STL, like UnaryFunction, (which take one parameter as input and return a result) and UnaryPredicate (actually called “Predicate”, and takes one parameter as input and return a Boolean). What can we do if we have a function that takes only one parameter in its argument and we want to pass to it a binary functor, like less ( ). How can the function obtain the second operand to pass it to the functor? Also assume that we want to pass a binary functor to a function that is expecting a unary functor in its argument instead?

```
less <double> P;
plus <int> Q;

int Fn1 (int a, int b, Q ( ) ) { return  Q (a, b) ;};
//Fn1 expects a binary functor that returns an int,
//a functor of type BinaryFunction.

bool Fn2 (double a, double b, P ( ) ){return P (a, b) ;} ;
// Fn2 expects a binary functor that returns a bool,
// a functor of type BinaryPredicate.
```

Figure 2.6: Functions with Different Functor Types

The STL provide adapters that can adjust this kind of irregularity when combining the building blocks, so that we can still put some blocks together that seem incompatible at first sight.

Passing sequences as input parameters.

The main advantage of the functors and adapters is that they can be applied repeatedly on a sequence of elements; a container. We can write (or reuse) our functions and pass them a container (or more) and functors with adapters if needed.

The function F4 ( ) can be passed a complete sequence of elements (a container) to test for the elements that are less than 5 , and do something to those elements (e.g. replace them with value 10). So we pass to F4 ( ) a container of elements to test (by passing an iterators to first- and past-the-end positions) and a functor less ( ) which is bound to 5 to be compatible with F4( )

```
F4 (iterator first, iterator last, bind2nd ( less <int> ( ), 5 );
```

F5 accepts two sequences and a binary functor that applies a binary operation to them, element by element and produces a new sequence of the results. Note that we need to pass three containers to the function: the two input containers and the output container to be filled with the results. Note also that we need the size of one input container only (the smaller of the two, passed before the larger one, if they were not equal in size) since the algorithm will stop as soon as any input container is exhausted. We pass the beginning and past-the-end positions of first container, and the beginning of second input container, and the beginning of the output container

```
Template <class InputIterator, class OutputIterator, class BinaryOperation >
OutputIterator manipulateContainers (InputIterator       begin1,
                                     InputIterator       end1,
                                     InputIterator       begin2,
                                     OutputIterator      result,
                                     BinaryOperation     bin);
```

Figure 2.7: Passing Multiple Containers to a Function

And we can use the function as shown in figure 2.8. Note that the container list provides two functions begin( ) and end( ) that returns iterators to the first and past-the-end locations of the container respectively. STL containers must provide these standard functions. What the function does is, it selects every two corresponding elements from the two input containers, passes them to whatever binary functor object it has in its argument, gets the output from the functor, and puts it in the output container. The function repeats the process until the first input container is exhausted. STL has allowed the function to be independent of the container, the container elements, and the operation applied to them.

```
void main ( )
{
int in1[10], in2[12], out[10];
int* in1_end = in1+10    //need the location of past-the-end element of in1

//… code to fill the two input containers in1 and in2

manipulateContainers (in1, in1_end, in2, out, plus<int> ( ) );
//the array name is a pointer to its first element

// … other code

list<double> list1, list2, list3, list4;
//we use same function on a different container with a different element type and
//different functors

// … code to fill list1 and list2

//list3 will have element-by-element subtraction of list1 and list2

manipulateContainers            (list1.begin( ), list1.end( ),
                                 list2.begin( ),
                                 list3.begin( ),
                                 minus<double> ( ) );


//list4 will have element-by-element multiplication of list1 and list2
manipulateContainers            (list1.begin( ), list1.end( ),
                                 list2.begin( ),
                                 list4.begin( ),
                                 multiplies<double> ( ) );
}
```

Figure 2.8: Using the Function manipulateContainers

2.1.9 Allocators

Containers are responsible for managing the access and storage of their elements. The access is allowed by providing suitable iterators and a set of functions to provide information about the elements, return a reference to- or add/ delete an element. The storage is an essential part to the container, allocating memory to new elements, and freeing memory from the deleted elements. Normally the user need not worry about storage management when using a container. This is done behind the scene without any user intervention and this greatly simplifies the use of containers. On the other hand, some special applications require a special kind of storage management and cannot simply work with the standard one. This would mean that those special applications will have to discard the STL containers altogether and write their own special containers that have a special storage management. This means a lot of efforts, most likely without the efficiency and elegance of STL code. To be able to serve those special cases as well, while still keeping the containers simple for everyone else, the STL made a further separation of building blocks. The Container is not exactly responsible for the storage. An allocator class was made responsible for the storage of the container elements, and each container handles storage by simply asking an allocator object to do that. To keep things behind the scene for users, each container uses a default allocator for its storage needs, unless given a specialized allocator to handle the job. For example, the container "vector" has the format:

```
template <class T, class Allocator = allocator<T>  > class vector ;
```

We can see that a vector of integers can be instantiated by simply writing

```
vector <int> intVec ;
```

and a default allocator object will be used, without user intervention. On the other hand, the same container can be passed a specialize allocator, written by the user, or imported from another library as follows:

```
vector <int, my_allocator> myVec ;
```

In this case we made full use of the standard container class in a special application without writing a special container. In writing a specialized allocator, they must have the standard interface of the STL allocators to be integrated seamlessly into any STL container. This is very useful in database, where we need a special storage on the hard disk, where files are too large to fit in memory. We can now put all the building blocks together on one diagram. These building blocks satisfy the standard interface provided by STL for components. They are compatible with each other, and we can replace on or more of these blocks with a specialized block provided that it has the same interface of the corresponding STL block. Figure 2.9 shows a general system Layout for this system.

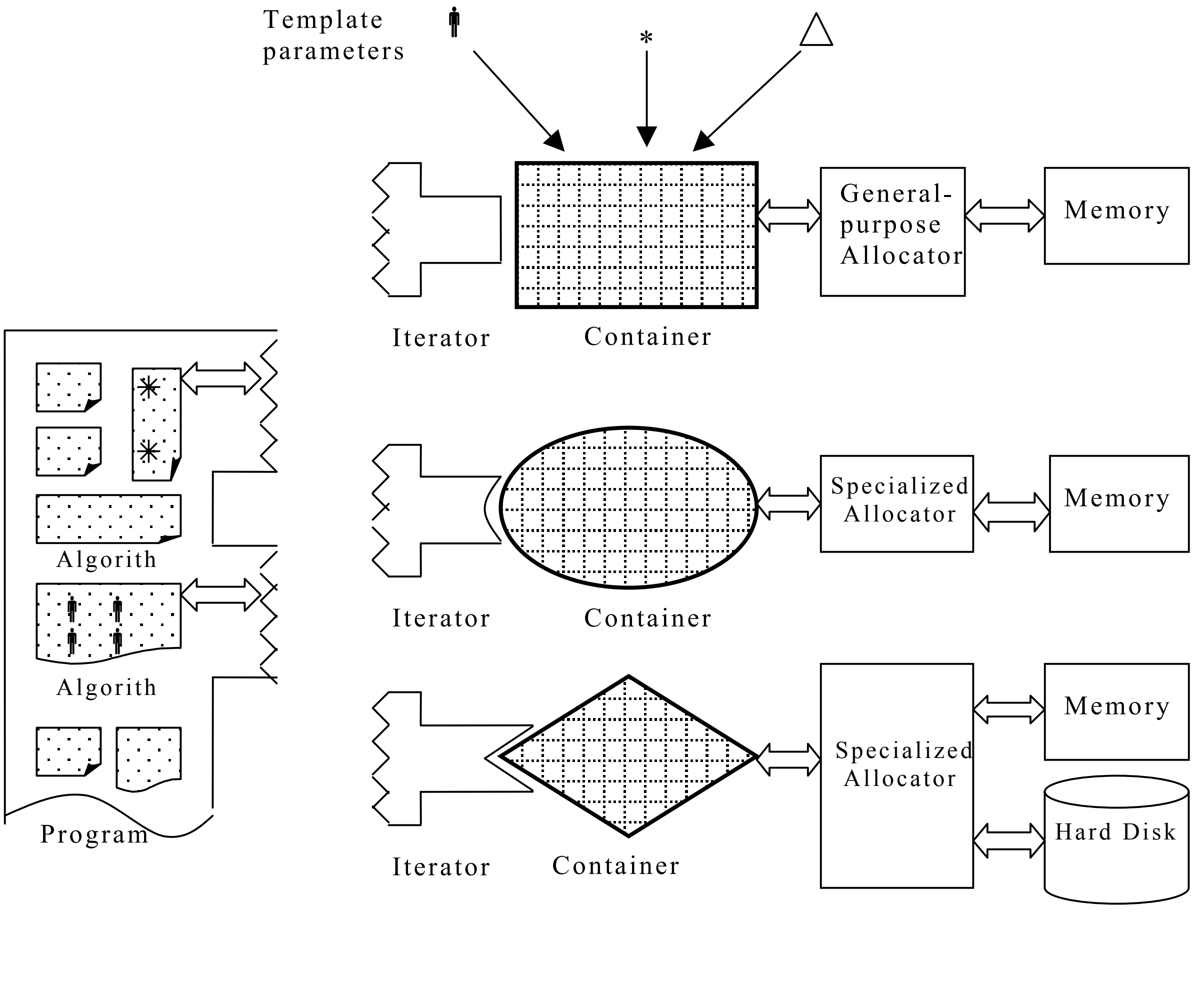


Figure 2.9: The General System Layout

2.2 Building Blocks Connectivity

The STL provides us with a complete set of compatible building blocks that can be connected together to build complex systems. Using these blocks, we can produce a blueprint design of the system, which can be drawn as a directed graph. The nodes would be STL blocks and the edges would be the connections between them. Semantically, a connection between two nodes would represent a use relationship between the two STL blocks represented by the nodes. Connecting these blocks is not, however, done at random. We cannot simply connect any one block of our choice to another and put them to work. There are certain rules that apply when trying to connect two blocks together. In order to show these rules easily, we will divide the three fundamental building blocks, Containers, Iterators and Algorithms, into categories. Then we can show how these categories connect together properly.

2.2.1 Iterator Categories

We can categorize the iterators according to their movement ability. Some iterators have more movement freedom than others. Figure 2.10 shows these categories.

Input Iterator

Input iterators have the least amount of movement; they can provide only one step forward. They are single-pass iterators, meaning that they cannot be used to access the same container twice; they might give different values each pass. They are used as rvalue to take input into program. Output iterators are similar to them, only they work as lvalue, taking output from the program to a container.

Forward Iterator

Forward iterators have the ability to move forward. They support the operator++ to step forward incrementally. They support multipass concept, where they give the same values every time they pass through the container. They do not support backward movement. To access the previous element, we increment the iterator all the way to the end of container, and start a new pass from the beginning until we get to that element.

Bidirectional Iterator

Bidirectional iterators have all the abilities of the previous two categories. They also support the backward (reverse) movement using the operator --.

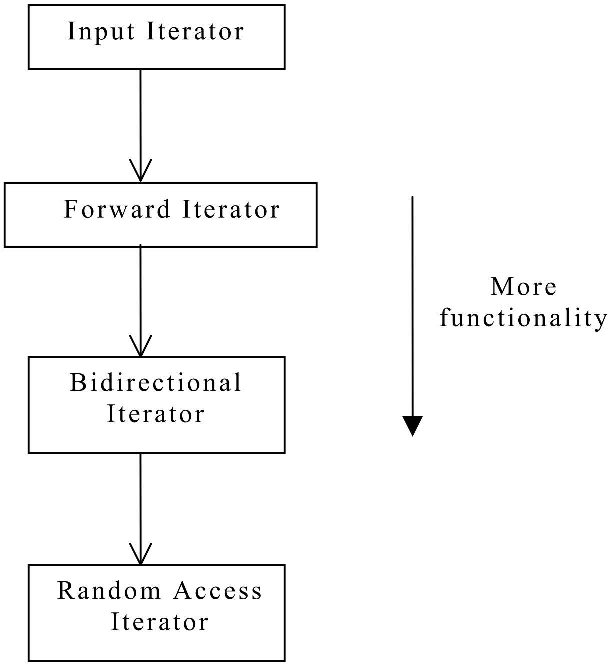


Figure 2.10: Iterator Categories

Random Access Iterator

Random Access iterators are the iterators with the highest degree of freedom. They can move forward, backward and also allow direct access to an arbitrary element using operator [ ] to specify the desired element by its index.

2.2.2 Container Categories

We can categorize the containers according to the movement freedom they offer to iterators trying to access their elements. Each Container can allow iterators to iterate through its element with certain degree of freedom, depending on how the container organized and stored its elements. Figure 2.11 shows these categories.

Compatibility between Iterators and Containers

If the iterator has more movement ability than what the container can afford, the extra iterator abilities cannot be used unless we upgrade the container. Using this extra functionality without upgrading the container category is incorrect and dangerous. It can crash the program, or at least provide incorrect information

Also if the iterator has less ability than what the container can offer, the extra container abilities cannot be used, unless we upgrade the iterator. This is, however, not a serious practice, and will cause no harm since we use iterators to access the container so we will have no access to any incorrect functionality; we will just lose some.

Internal and External Iterators

Iterator can communicate with internal and external iterators. For external iterators, they are built outside the container, but for a specific type of containers; meaning that the internal structure of the container must be known to the iterator. They can be used with any container of the same type by passing to them a particular element of the container and using them to move in the container. For internal iterators, the container provides them as methods that return an iterator. Each container can return iterators of different categories. The container is wise enough not to provide an internal iterator with higher movement ability than the container can afford, yet they can typically return a lower category iterator if that is all we need.

Input Container

Input Containers are conceptual containers capable of providing only input iterator. They would only provide input to the program in a single pass concept, like input stream object. An input container is conceptual as it does not really belong to the program itself, but it is more of an external component that provides input to the program.

The keyboard can be seen as an input container that provides input to the program. A second pass through this container (in the form of running the program code twice and accepting a set of input variables by a set of cin statements each time) may produce different elements as typed in by the user in every run. This set of cin statements can be seen as receiving their values from the "conceptual" input container; the keyboard.

Forward Container

Forward Containers are containers that are organized such that they can be accessed in a forward incremental direction; like single-linked-list data structure. No backward movement is allowed. This type of containers can only provide a forward iterator or an iterator with less functionality.

Bidirectional Container

Bidirectional Containers store their elements in a sequential way and allows access to them in a forward or backward direction, in incremental steps only. This is similar to the concept of doubly linked list data structure. They can therefore provide Bidirectional iterators (or less-capable iterators if needed). Also any external Bidirectional iterator can access this type of containers with no problems.

Random Access Container

Random Access containers store their elements such that they can be accessed in both directions, and can also support direct access to an arbitrary element without the need to incrementally access all elements before it. Obviously they can support a random access iterator or less.

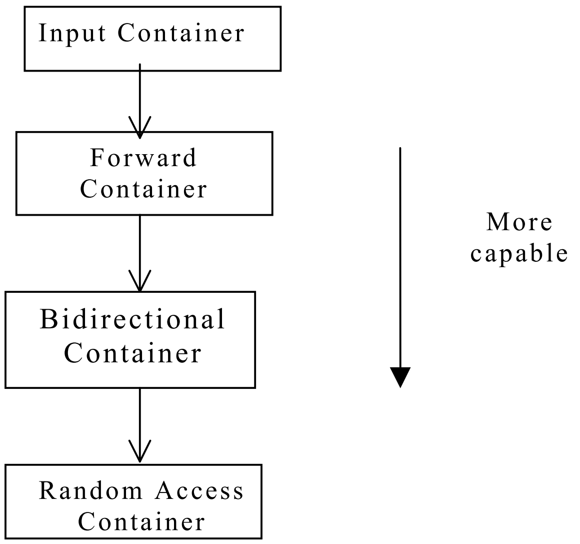


Figure 2.11: Container Categories

2.2.3 Algorithm Categories

A fundamental STL concept was to use iterators as an interface between algorithms and containers, allowing the same algorithm to be used on different containers and vise versa, promoting the cut-and-past concept on code; or code reuse. It might seem that we could say: “iterator allows the use of any algorithm with any container”. This is not exactly correct. A more exact statement would be: “iterator allows the use of many algorithm with many container”; hinting that some algorithms do not work with some containers. To make it clear, we can categorize algorithm using the same criterion. An algorithm will be categorized according to the iterator it uses. Figure2.12 shows these categories.

Compatibility between Iterators and Algorithms

For simplicity, we will consider the category of iterator needed for the algorithm, and ignore the category of the container as it can be inferred from iterator category. After all, this is why we use iterators; to make algorithms container-independent. Obviously the algorithm can connect to a higher-category iterator without any problems; it only will not make any use of the extra set of functionality provided by that iterator. Each algorithm will therefore be written with the minimum iterator category required to work correctly. Connecting an algorithm with a lower category iterator is, however, dangerous. The algorithm will be using some functionality that is not provided by the iterator, which will result in an incorrect performance, or even a program crash.

Input Algorithm

Input Algorithms require an input iterator to work properly. Their functionality would obviously be to input some elements to a container in an incremental unidirectional way, in a single pass concept.

Apparently this type of algorithms works fine with the lowest category of the containers (input container) and needs only the lowest category of iterators (input iterator).

For example the algorithm *accumulate* in figure 2.12 needs an input iterator to work correctly.

Forward Algorithm

Forward Algorithms need forward iterators to work correctly

For example, the algorithm *remove* in figure 2.12 needs a forward iterator to work correctly.

Bidirectional Algorithm

Bidirectional Algorithms needs at least a bidirectional iterator. This kind of algorithms will need to move forward and backward to work correctly.

For example, the algorithm *reverse* in figure 2.12 needs a bidirectional iterator to work correctly.

```
template < class InputIterator, class T >
T                       accumulate (InputIterator first,
                                        InputIterator last,
                                        T init) ;

template < class ForwardIterator, class T >
ForwardIterator         remove (ForwardIterator first,
                                ForwardIterator  last,
                                const T&  value ) ;

template  < class BidirectionalIterator >
void                    reverse (BidirectionalIterator first,
                                  BidirectionalIterator last);

template  < class RandomAccessIterator >
void                    sort (RandomAccessIterator first,
                              RandomAccessIterator last);
```

Figure 2.12: Some Algorithms

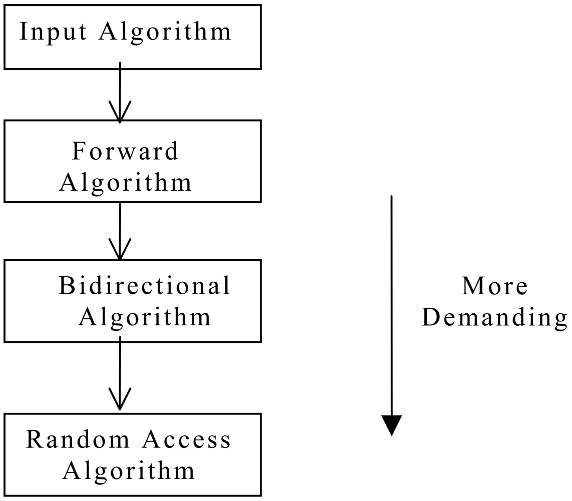


Figure 2.13: Algorithm Categories

Random Access Algorithm

Random Access Algorithms are the most demanding algorithms of all. They need the highest iterator category, a random access iterator. This kind of algorithms will need to move forward and backward and access any arbitrary element in the container to work correctly.

For example, the algorithm *sort* in figure 2.12 needs a random Access iterator to work correctly.

Downgrading an Algorithm

It is worth mentioning here that most algorithms can be downgraded to a lower category and still work correctly. This will, however, not be an efficient algorithm anymore. The minimum category for an algorithm will guarantee that it will have the best performance in term of time complexity as guaranteed by STL. Downgrading an algorithm by rewriting it such that it can be used with a lower category iterator could mean a much less efficient performance. For example, an algorithm that requires direct access iterator can be rewritten to use only Bidirectional iterator. Each time the algorithm will need to jump to another element, it will have to start from the beginning or current position and access all elements lying before the desired one. Such an algorithm could take very long time to complete.

2.2.4 A Blue Print for the Connectivity

Figure 2.14 shows how we can connect the categories of STL building blocks correctly. An arrow means that the two blocks are compatible.

We can divide the components into other categories using different criteria. For example, we can divide iterators according to their mutability; read-only iterators allow algorithms to take a copy of container elements without being able to change it (a const iterator). The container element itself can be constant for all algorithms, or can be mutable by other algorithms. Containers can also be divided into sequence and associative container, regardless of the above categories.

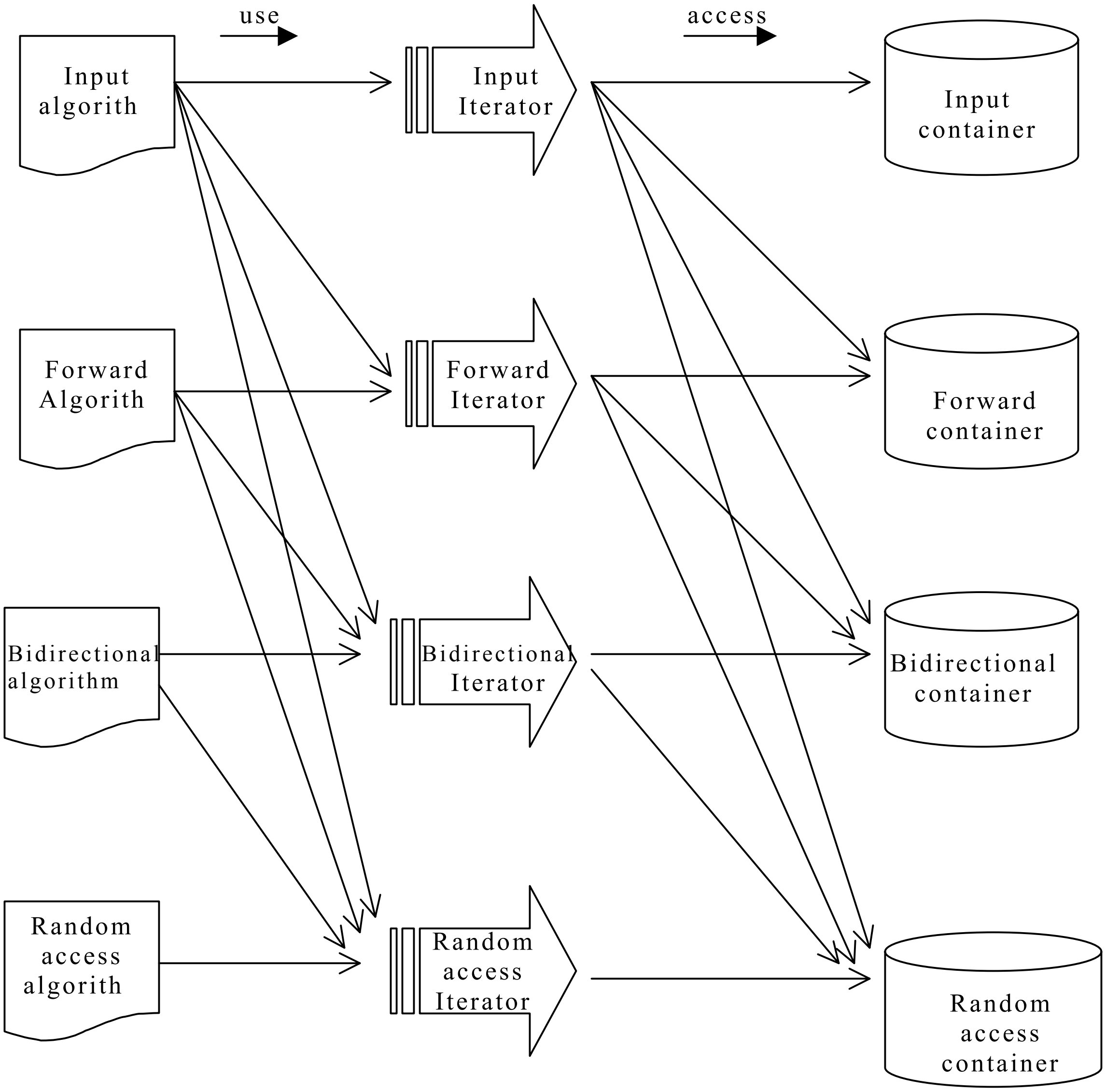


Figure 2.14: A Blue Print for Connectivity

Chapter 3 Database Indexes

3.1 Introduction

Indexes help access information. To make a large amount of data useful, it has to be organized and classified into logical parts, 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. A simple example is the table of contents of a book. We use the table of contents to find information in the book using the topic name as a search method through the index.

To build and use such an index we perform the basic steps:

1- We divided the book data into Partitions (in this case chapters, sections and paragraphs).

2- We found a key for each partition to give us some clue about the contents of the partition (in this case a header or a caption for each part relevant to its content).

3- We gave a physical address; a reference to each partition, marking its exact location (in this case unique page numbers).

4- We decided on a certain access method; a criterion of how to search for information (in this case since the number of entries is relatively small, the index would be limited in size to few pages, we will search the index sequentially by scanning the keys from the first one until we find a topic of interest. This would take O(n) time to search where n is the number of entries. Since n is small, an O(n) algorithm performs well enough. Sequential listing of contents will also help to give an overview of the book, but that is not one of our concerns for an index.

5- We built the index by collecting all the keys and references in one part – the table of contents.

Another example of an index is the keyword index that can be found at the end of the book. This index is built to locate data in the book using keyword as the search criterion. We follow the same four steps to design and build the new index:

1- Partitions are the keywords we want to highlight in the data and include in index.
2- The best key for the keyword is the whole keyword (or part of it).
3- Each keyword has a page number (or numbers) where it is mentioned in the book
4- Since this index will have many more entries than the previous one (n, the number of keywords, is now much larger, O(n) might be unacceptably high. Searching a table that contains a few thousand unsorted keywords can be annoying. We thus decide to sort the keys alphabetically and use binary search as an access method. We go to the middle of the alphabetical index and check the word and then decide if the keyword we are searching was before or after the middle, and so on. This will have a time complexity of O(log n) on the average, giving an improved performance
5- We then build the index and add it to the book.

Saving time is clearly an advantage in using an index. We could scan sequentially through all the book headers (chapters, sections and paragraphs) to find the topic we are looking for, but scanning through the index gives the same results in a much faster time. With keywords it is even clearer. To search for a keyword in a book without a keyword index, we need to search every word in the book sequentially which is a lot of work.

3.2 Index Tree Applications

Almost any kind of data can be stored in a database. Numbers, text data, images, maps, fingerprints and even audio and video files are typical examples. A lot of work has been made on the concept of index tree, resulting in many different types of trees suitable for this variety of applications. The idea is to be able to categorize data in such a way that we can retrieve it fast, and then build an index that allows us to locate data of interest in an efficient way. Similarly, different index concepts were developed to be able to deal with the variety of application data. In the next pages, we will briefly present some of these index trees, and show how to use them with database applications.

3.3 B+ tree Index Example

Figure 3.1 shows a simplified B+ tree index layout. It contains pages that store pairs of keys and pointers. Pages are laid out in a tree-like structure. The top level contains only one page, the root page. The lowest level contains the data (or references to the data). Each page in that level is called a leaf page. Data is included at the leaf level only. No data is allowed to exist in other levels. The tree is balanced; all leaves are at the same level. The number of pointers, R in a page determines the number of children of that page. This number, called the fan out of the tree, has a maximum allowed value, and is fixed for the tree. It shows the multiplication factor from one level to the next. Each page is allowed to be totally or partially filled with a factor called fill factor. The minimum fill factor is typically 50% and the maximum is 100%, except for the root page, which can have as little as two children or as much as any other page. The example shows the basic search operation for a value 202 in the leaf level, starting from the root. Other operations involve writing to the index, or modifying its contents, like insertion and deletion operations.

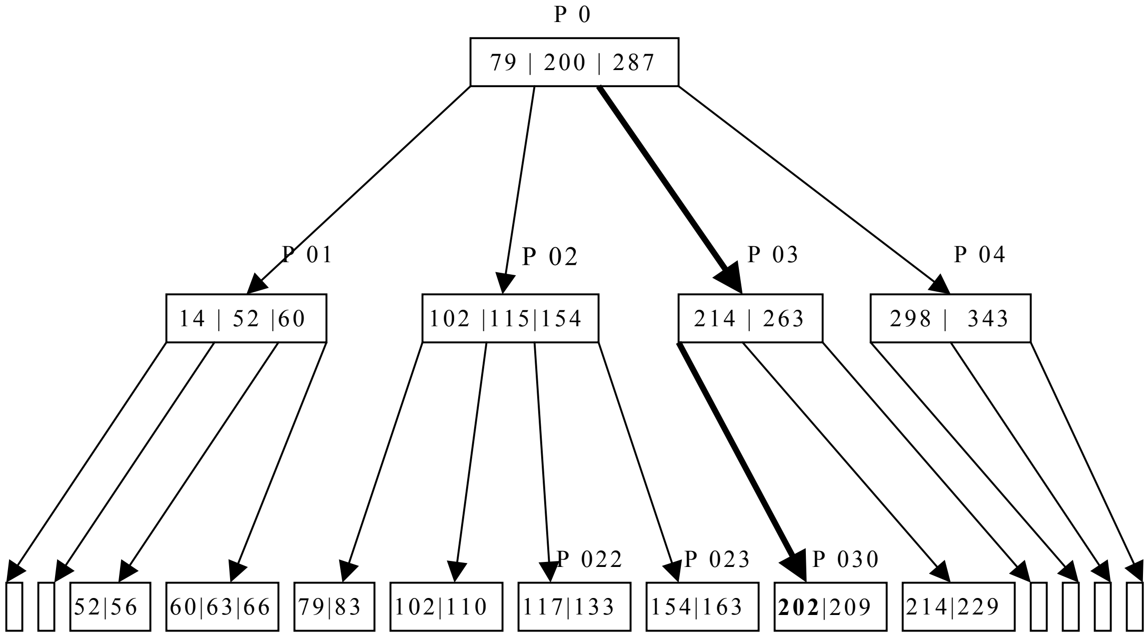


Figure 3.1: B+ Tree

These operations are far more complex in their algorithms and procedures than a simple search operation. A single insertion or a deleteion operation can result in one simple step, or it can trigger a chain reaction leading to a large number of related steps spread over multiple index pages with multiple sub operations of splitting pages, merging pages or moving some contents between pages before it comes to an end. The algorithm for these operations and their related sub operations must guarantee the integrity and correctness of both data and indexes after any number of operations with any combination of data.

3.4 General Domain Applications

The previous example showed an index using integers (or comparably strings of characters) as its data type, and equality as its search criteria (query). This works well for traditional relational databases. It can also be extended to serve other queries (range query for example). For some applications, however, that example does not quite serve their needs.

One domain of those applications is the spatial database, with a multi dimensional data type. The data in this domain can be represented in the Cartesian space as multi-dimensional objects. For examples geographical information about earth surface, cities layout in a country, or detailed city maps for buildings, streets, etc. Some index trees have emerged to deal with these database applications including K-D-B-tree [Rbn81], R-tree [Gut84], R+-tree [SRF87], and R*-tree [BKS90].

As an example, the R-tree is a height-balanced tree similar to the B+-tree, but it considers the data type to be an n-dimensional polygon in Cartesian space, composed of a group of lines related together. Figure 3.2 shows a collection of polygons representing some data. This could be any geometric figure, a city map, districts map, etc. Arcs can also be considered by approximating them to a number of lines.

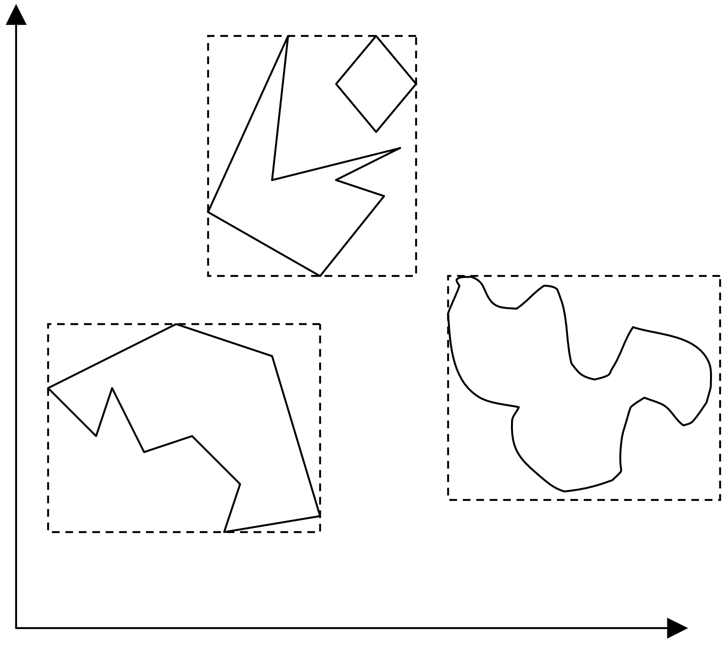


Figure 3.2: Finding the Key for Partitions of Polygons

To build an index, we need to partition data, by considering each polygon or a group of polygons as one partition. We then need to find a key for each polygon (or group of polygons). The key is the bounding rectangle to the spatial partition, covering all the partition (of one or more polygons) while reducing the data size from many points of (x, y) describing each polygon to just four points. To represent a rectangle we just need two points; the op left, and the bottom right corners. Now this key covers all the partition with much less data size; the desired nature of any key. Note that partitions might be allowed to overlap depending on the application. This will not affect data integrity, as for overlapping keys their associated data can still be easily distinguished.

Searching will typically have a point or an area (another polygon) with the search key as the smallest rectangle to cover this area. The search mechanism will try to find those polygons in the database that have something in common with it, by finding the database index keys (rectangles) that intersect with it. Figure 3.3 shows an example of searching a key and we can see the three data keys that match that key (by intersecting with it).

The figure shows that some rectangles might intersect with the key rectangle we search, giving positive search result while the two actual data parts inside the polygons do not intersect, making the data found by searching the database useless. This is a side effect of using keys for the polygons and is not harmful in being unable to locate correct data inside the database; It will just impose some acceptable performance burden by accessing some useless data once in a while.

Similarly for an index, we need to merge partitions by finding one key to cover all merged partitions (for merging two pages, or for building a higher level in the index where each key will cover many keys below it).

Other applications use sets as their data types where data entities are sets of related or unrelated elements. Keys for sets can be found using different techniques and algorithms to extract a summary for each set. Set theory with its notations like union, intersection, subset... is used here to combine keys, search for a key, etc.

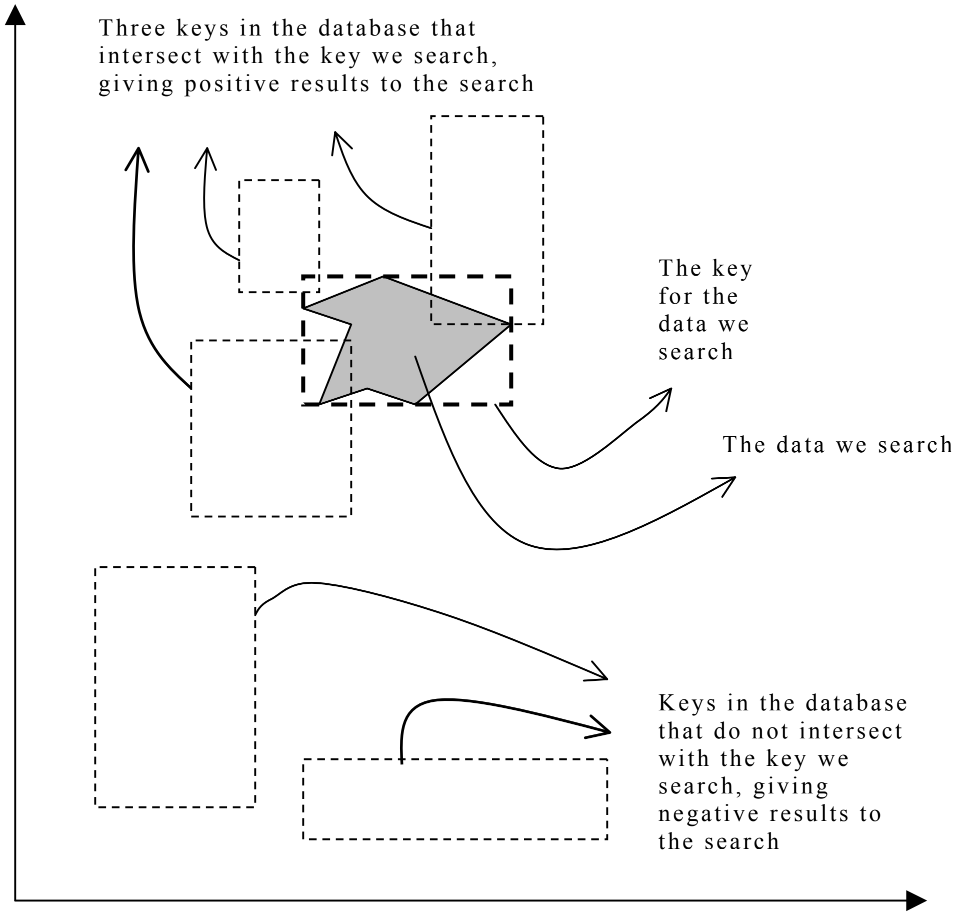

Three keys in the database that intersect with the key we search, giving positive results to the search
The key for the data we search
The data we search
Keys in the database that do not intersect with the key we search, giving negative results to the search

Figure 3.3: Searching Data Using a Key to Find Matching Keys

3.5 Similarity Search Applications

The general domain database and the linearly ordered domain have concrete results for a match, either true or false. We can see them as equality search applications. In similarity search, on the other hand, we do not look for an exact match, we are looking for data that is similar to what we have. This is useful in domains that need to build a database of images, like multimedia, journalism, art, astronomy, image and voice recognition, traditional medicine (e.g. 2D X-ray and 3D CATSCAN medical imaging) as well as new genetics and protein databases.

One main difference in these applications is that data is not directly represented by concrete numbers. Another difference is that even after we have translated the images to numbers (they have to, to be stored in digital medium), we cannot simply compare these numbers to decide on matching.

Considered a simple case of scanning and storing the same image multiple times as pixels with digital values, then comparing them with a similar image to retrieve back these stored occurrences of the image. Apart from the great inefficiency due to the large amount of data to be compared, a slight shift or rotation, or even a rounding error might give a different number for few pixels, giving a negative search result for the same image, which is unacceptable. If those images were compressed to increase storage- and transfer efficiency, chances are all occurrences of the image will be different.

Seidl and Kriegel [SK01] show how to add flexibility to large image databases including pixel-based shape similarity to tolerate and compensate for such errors and locate similar images.

This example shows that we are looking for the same contents with similar digital representation. Similarity search can have much broader meaning when we want to look for similar contents. This makes the subject more complicated than just numbers. Deciding whether objects are similar or not is very subjective, and could be different for different persons, or even for the same person under different circumstances. We will show this with an example.

Consider three objects: A goldfish, a white shark, and a dolphin and we want to select the two most similar ones out of them. For a biologist, the gold fish and the shark are similar, both are cold-blooded, fish, while a dolphin is a worm-blooded mammal with lungs to breathe. Now we change the person. For an ordinary person swimming in water, the gold fish and the dolphin are similar, both are peaceful water creatures, while a white shark is a terrible monster. Now we change the circumstance of the same person. If this person is looking for a house pet to put in a glass bowel, the dolphin and the white shark are similar and must be avoided, both are too large and have enormous appetite and not sold in pet stores, while a gold fish is perfect for the purpose.

Due to this subjectivity, content similarity research developed different criteria to manage and decide on the similarity decision. We can recognize two main parts: criteria for extracting a key, and criteria for comparing keys to decide on similarity between them.

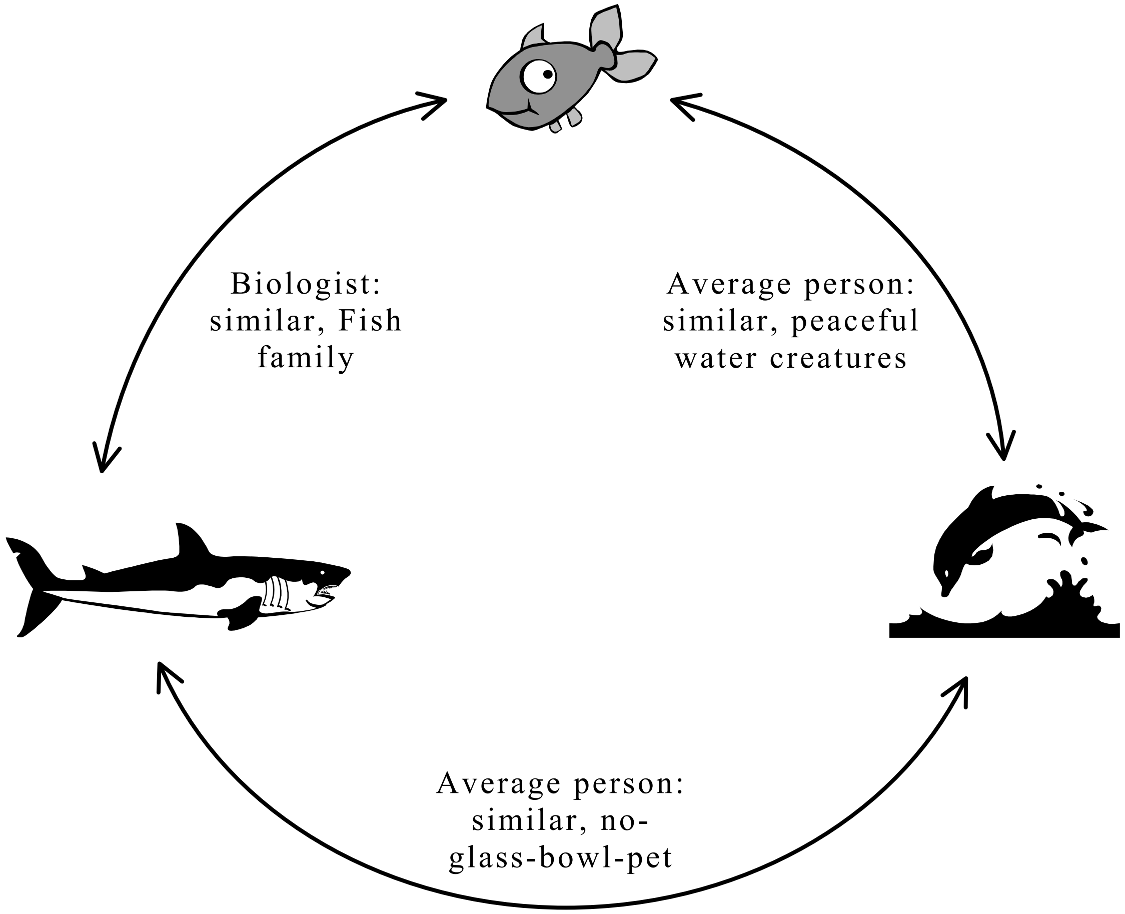


Figure 3.4: The Subjectivity in Similarity Decisions

3.5.1 Extracting keys: Feature Vectors

Images of 2D or 3D objects could be digitally represented and stored as a mathematical representation using a multidimensional matrix or CAD tables or other digital forms. However we want to extract a key for this large object to satisfy the main properties of a database key: smaller size while still having information about the contents. We will consider two examples for 2D and 3D images.

For 2D color images, for example, the SR-tree [KS97] develops a criterion to extract a color histogram to be used as an efficient key. It divides an image into 4 regions from upper left to lower right. For each region, munsell color space (hue, saturation, intensity) are measured and quantified into nine basic colors, giving a color histogram for the region. The four histograms are concatenated to give a 36-dimensional feature vector, which is then reduced to a 20-dimensional feature vector (since higher dimensions dramatically increase search time, making queries slower). Each image will have a 20-dimensional key (feature vector) representing its contents. The key is much smaller than the image itself.

For 3D images, Examples of 3D-protein database [AKKS99], [KKS98] divide 3D images of complex-shaped protein molecules into 3D cells (concentric shells of a sphere, sectors of a sphere, or a combination of both). Assuming that protein images are given as sets of points in 3D space [SK95], the shape histograms are determined by counting the number of points within each cell. This gives a feature vector for each protein image that represents its shape. This feature vector, again, is much smaller than the original protein object, and represents its contents, so we can use it a key when building a database for the 3D protein images.

Feature vectors can also be text or keywords about the images. For example in journalism, we can generate a feature vector for the thousands of pictures taken daily by giving them numbers to represent the date and place they were taken, the contents (people, objects, landscape) and the subject (politics, celebrity, fashion, sports) and so on. This could generate the feature model for pictures. We can also build a hierarchy of these keys (feature models) by defining criteria to combine groups of keys into a super key that represent them. Algorithms can then be written to generate these keys.

3.5.2 Comparing Keys: Distance Functions

Feature vectors give compact digital representations of images. Nevertheless comparing two feature vectors as keys is not a straightforward application. As we can see, we do not look for exact match because of the nature of digitizing images errors, and contents differences. Different criteria have been developed to decide how similar two n-dimensional feature vectors are, such as the Euclidean distance in SR-tree [KS97] or the quadratic distance function explained in [AKKS99].

3.5.3 Searching the Database: Query Processing

After developing a criterion to extract keys and a criterion to compare them, we can use them to query the database to find similar objects. Goals can differ from one application to another, or within the same application. For example in biomolecular databases (like 3D protein images), a basic task is classification of new molecules [AKKS99]. In molecular biology, there are already defined classes for molecules. So after a new molecule has been discovered, we need to determine its class. First we look up the database and find the closest neighbors to it, then domain experts can decide on the exact class by comparing the new molecule against the similar ones in the database with more complex classification criteria than just the 3D shape. In this case, efficient classification algorithms can be used as fast filters for further investigations by the experts.

In other 2D, 3D, or color image applications for example, similarity search can use other algorithms to lookup similar pictures for fingerprints, or for news archive image search.

3.5.4 Building an Index to Speed up Query Processing

Comparing feature vectors to determine the distance typically include a large amount of calculations, which can be a heavy load on the processor. Some search sessions can take overnight to finish, so research has been done in the area of reduction of dimensionality of feature vector.

Searching a database in parallel can some times speed up the search.

As the dimensionality of the feature vector increases, or the complexity of distance function increases, or both, the time for an evaluation of two vectors can significantly increase. We can avoid many unnecessary feature vector distance evaluations by using an index as a pre-filter. Multi-step keys can be used to eliminate large number of objects without the expensive full-evaluation of their feature vectors, thus building a multi-level hierarchical index to the database. The idea is to cluster a group of similar keys (feature vectors) together into one super key working as a pre key (pre filter) to all members of the group, and apply this concept recursively to build a hierarchical index. [BBBK00] explains the difference between hierarchical algorithms and partitioning algorithms. Hierarchical algorithms decompose the database into several levels of nested partitions (clusterings), whereas partitioning algorithms construct a flat (one-level) partitions, each of which contains similar database objects. In this paper, a high performance clustering concept based on the similarity join is introduced to help build the key of keys by efficiently clustering a group of similar keys.

Another approach to speed up the search is to use a parallel similarity search [BEK+98], [BBB+97]. In parallel search, data is declustered (by uniformly distributing it over different disks) and thus the time for accessing and processing can overlap. [BBB+97] explains the importance of efficiently distributing data among the available disks and shows how this can speed up the search and introduces some efficient declustering algorithms.

3.5.5 Traversing an Index

Access methods used to traverse a multi-dimensional database in similarity search domain have many more variations than the equality-based database domains. [GG98] gives an extensive overview of multidimensional access methods. As explained before, indexes are built by recursively clustering data together and creating keys for the clusters. The higher level keys cover larger clusters and are thus coarse-grained compared to the lower ones. Traversing the index will produce different similarity distance at different levels, which may result in moving up and down the index structure as these values change, in order to locate the nearest neighbors. P. M. Aoki [Aok98] gives an example obtained from the SS-tree [WJ96] to demonstrate this feature. The SS-tree organizes records in hierarchical clusters. Each cluster is represented by a centroid and a bounding sphere with minimum radius to cover the elements within a cluster as shown in figure 3.5 in centers c11, c12, c13, c21, c22. Each group of spheres can be further combined into a higher level cluster (sphere) of one centroid representing the weighted center of mass and a radius to cover the elements in the group. This is shown in figure 3.5 as c1 and c2. As we search a key using a query, we can represent the search process as searching the closest (nearest) neighbor data (shown as black spheres) to our element key (shown as x in the figure). Since this is a multilevel index hierarchy, data elements (black spheres) are not directly exposed until we reach data levels. At higher levels of the index, only hierarchical keys (the hollow spheres) are exposed. In the first step, we compare our query to the high level clusters centered at c1 and c2. c2 looks closer so we descend to the right subtree (link b) at second level in the index. We retrieve the c21 and c22 clusters. At this point, both c21 and c22 are further than c1, i.e. c1 is now the nearest to our query of all c1, c21, c22, so we abandon the subtree of c2 (link b) and jump back to the subtree of c1 using link a. Descending on the subtree of c1 yields c11, c12, c13. As we measure the distance, we find that c13 is the nearest neighbor. We can now access the three data components of c13 and find the nearest of them to our query x. From this example it is clear that we may need to go back and forth within the tree levels to get to the nearest neighbor. The example also depicts that as we get to a lower levels (like c21, and c22 level) the closeness (proximity) of clusters differ as they become more exact that the coarse-grained keys at higher levels (c2). That is why c21 and c22 appeared further than their super key c2 promised when comparing it against c1.

Similarity search algorithms use different distance evaluation techniques to determine the distance between two objects, which will indicate how similar they are. Less values for distance will imply higher percentage of similarity, with zero distance considered as 100% similar.

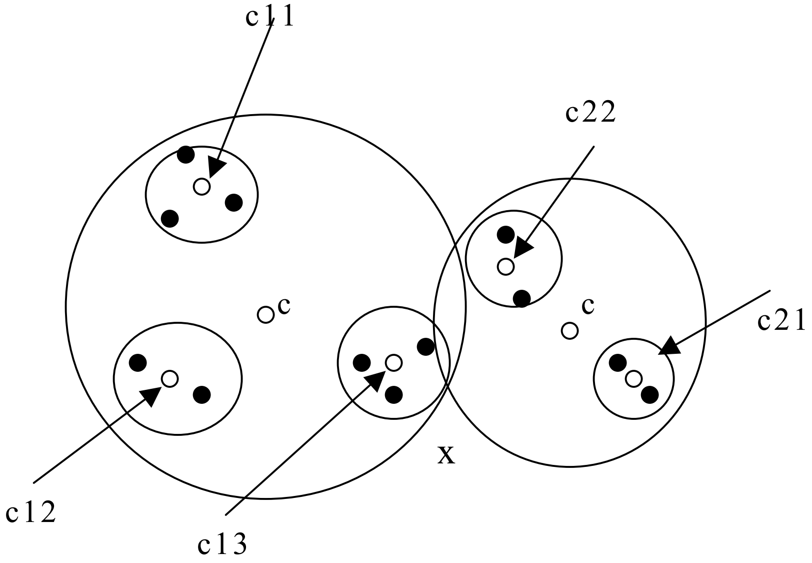


Figure 3.5: The SS-tree in Spatial Presentation

Chapter 4 Context Analysis

Database is manipulated in different ways depending on the user. In this chapter we concentrate on the different users and the context of using a database system.

4.1 Context Use Cases

Figure 4.1 shows the main actors of a DBMS.

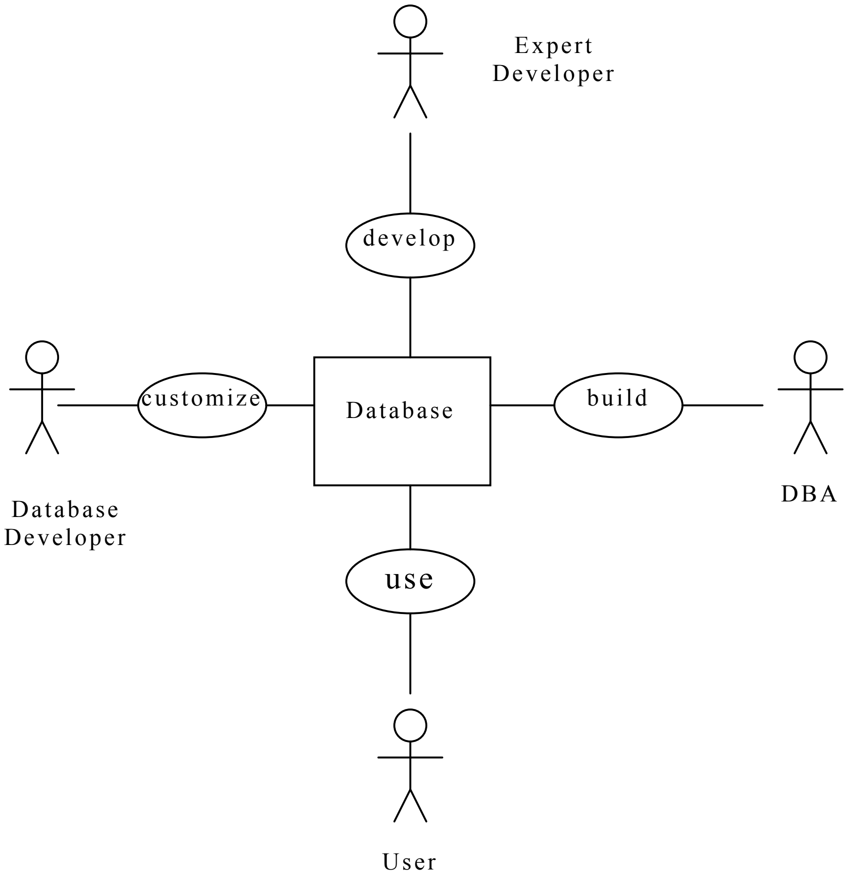


Figure 4.1: Context Use Case Diagram

4.1.1 Actors

Expert developer

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

Database developer

Database developer is a knowledge domain expert, responsible for customizing an existing database system by modifying some of its parts (components) or by replacing some parts with others to be able to support a new data or query type, or support a completely new index structure with new access method. Database developer is also responsible for setting the parameters to suit the system platform and the application variables.

Database administrator

Database administrator is the person responsible for building the Database system, which include building the scheme, the physical tables and the indexes used to access them.

User

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

4.1.2 Use Cases

Develop Database

The expert developer puts a complete design for a database system and implements it to run on a platform.

Customize Database

From the extension programmer's point of view, part of the database can be customized to support new data types and/or query types. New index structures supporting new access methods can also be produced by modifying few parts of an existing index that has a different structure.

Build Database

The DBA is responsible for building the database by using a DDL language to define the scheme, and by defining the physical data tables, storage formats, and data partitioning, and building the indexes needed to access them in appropriate index files.

Use Database

This is the main use case that concerns the application, while the previous use cases are invisible to it. It involves connecting to an existing database and either looking up some data or doing some transaction (insert/delete) on that data using a query language.

4.2 Detailed Use Cases

In this part we will consider the index as a part of the database with the following properties:

- It is built as a balanced tree structure
- It can be customized to suit different applications
- It can be used to lookup data in the database (read-only access)
- It can also be used to insert or delete data in the database (read-write access)

Figure 4.2 shows the diagram for detailed use cases.
For more details on the index, see section 2.4, "Database indexes".

Design database

The expert developer puts a complete design for a database system to satisfy the functional need of database domain experts as well as the non-functional needs like storage, and retrieval issues, platform mounting, etc.

Implement database

The expert developer implements the design including all details such as the physical access details and the platform-dependent details.

Provide new query

The database developer defines a new query and passes it to the existing database to be used in searching the existing data.

Provide new data type

The database developer defines a new data type by writing an index implementation to produce a new index capable of handling the data type used by a new application. This includes defining suitable keys to describe the data partitions, and defining ways to compare them, setting a suitable layout for the node pages and a page policy for adding/deleting this new data type keys inside them.

Define new access method

The database developer defines a new method 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 method

Set index parameters

The database developer sets the index parameters, like the tree order, the page minimum fill factor, and the page size, depending on the system environment, hardware profile, and application variables.

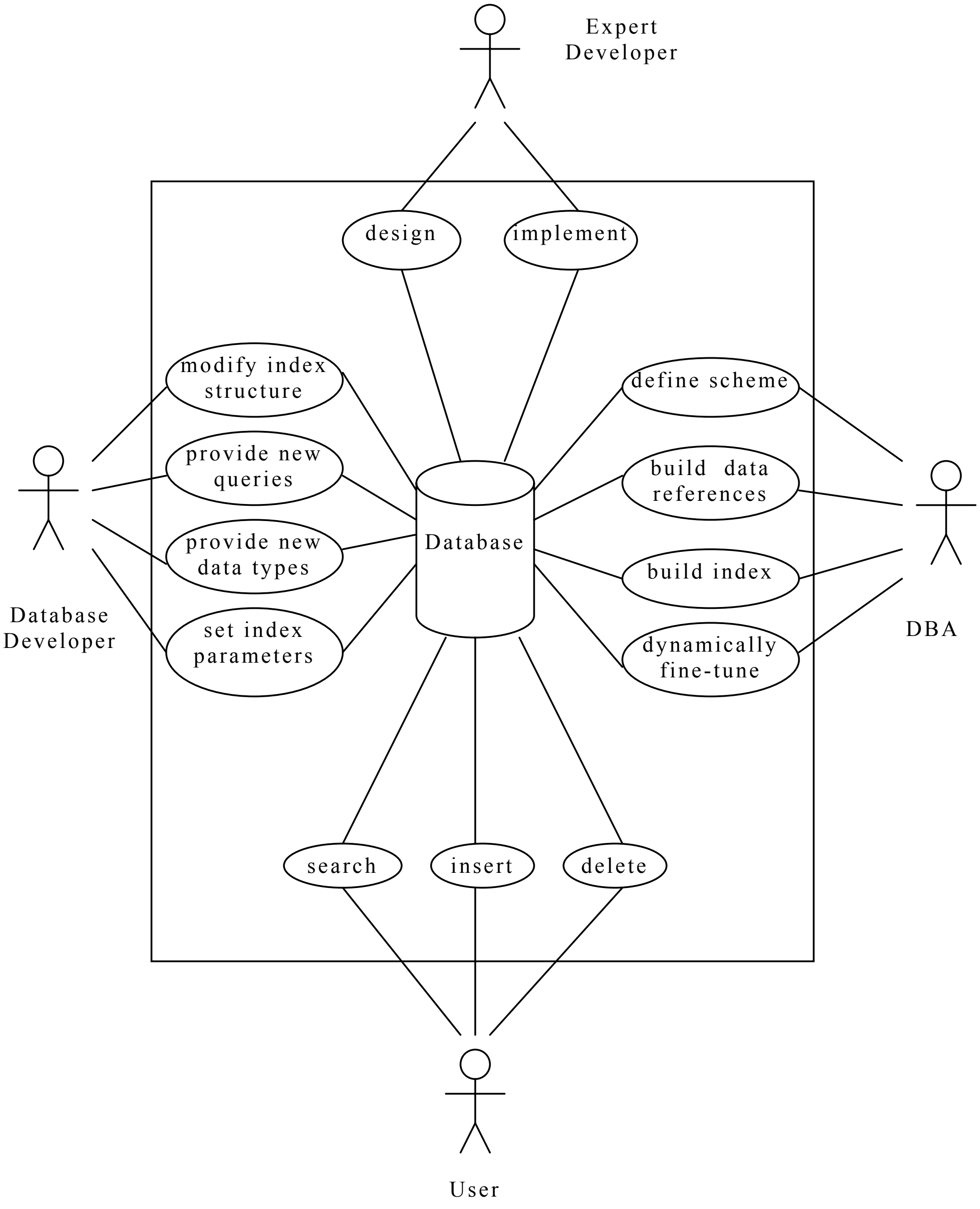


Figure 4.2: Detailed Use cases

Define scheme

The DBA will need to write the scheme describing the data tables.

Build data references

The data, once available, is prepared for the indexes by extracting suitable attributes to be used as keys in each index (primary / secondary) along with the corresponding physical location references. They are then sorted by their key value and in case of a secondary index further sorted by their physical location or by their primary key values, then put in the reference file as pairs of key, data reference.

Build Index

The DBA will build the index by successive insertions of the pairs of key, data reference into an empty index file (bulk loading the data reference file). A successful bulk loading operation will yield a complete index to the data.

Dynamically fine-tune

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

Search

The application will connect to the database and use some query or a key to search for some data. An index will be used to lookup data. The index will return the matching data in the form of references to their physical locations. The references will then be used to access the data.

Insert

For an insertion, the search use case runs first to find a suitable insertion position. The position is then used to insert the data, and the index structure is adjusted if necessary to reflect the changes to the data.

Delete

For a delete use case, the search use case runs first to locate the matching data to be deleted. Then the candidate data is deleted. The index structure is adjusted if necessary.

Appendix A lists the flow of events.

4.3 The Database Data Model

The index is a major data component. Typically we have one primary index and multiple secondary indices, all built on the same data set. They help access the data using different keys/ queries for the search.

Each index is built on the physical data indirectly, through a data reference file. This reference file is composed of a key, data reference pair, and constitutes the data level of each index (see figure 4.3). This separation between physical data and data references allows for building multiple indexes on the same data set (data is not included in the index) and for the ability to change physical data formats without affecting the existing indexes.

In dealing with an index, we provide a data reference as an input (for insert / delete) or obtain one as an output (search). The data access component will then take the responsibility to bind the references used by the indices to the data tuples on the physical storage.

Depending on some system characteristics (like access time constraints, and system volatility) different binding mechanisms are applied (see book). The mechanism used must guarantee that changes in the tuple physical locations will still allow for all associated indices to access them correctly.

4.3.1 The Index Data Model

The index does not provide us directly with data, but with information about where the data is. This mean that the data stored in the index is different from the typical database information (tables, records, etc). Figure 4.4 shows a graphical presentation of the relation between the index data and the database data.

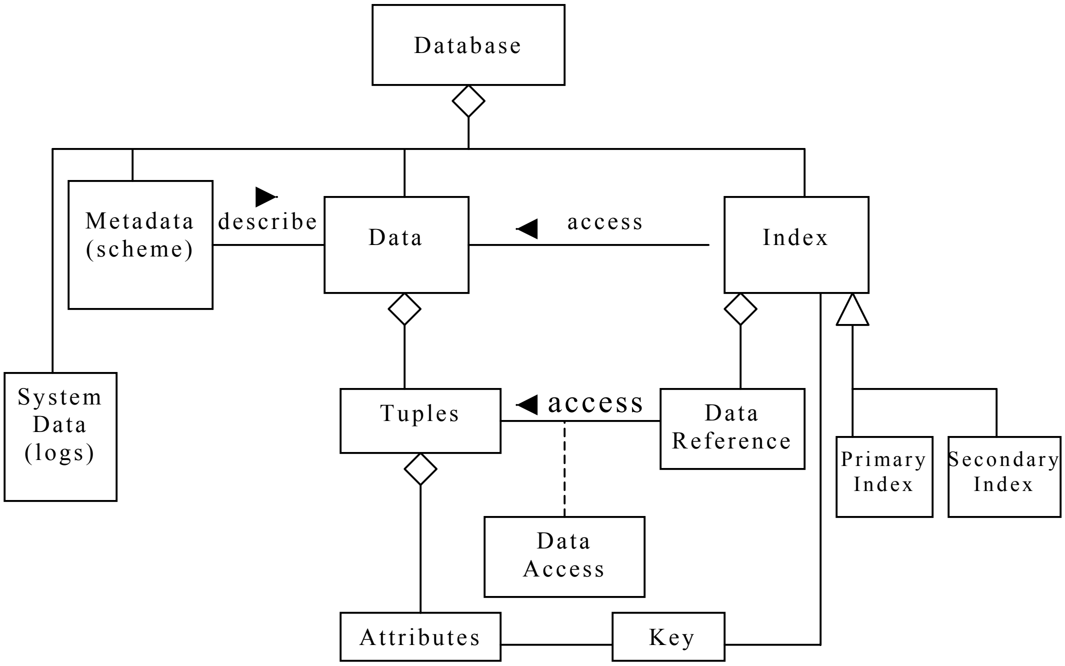


Figure 4.3: The Data Model

From this figure we can see that index data is references to physical data. We start querying the index using a certain key. With some comparison criteria, we try to find our way through the index down to the index leaf level, where the index data is stored. This will bring the index responsibility to an end. We should then refer to the physical database access mechanism to get the real data we are looking for.
Implementing an index goes the opposite way. We start with physical database data that we want to build an index for, build the reference file, then build the index using the reference file.

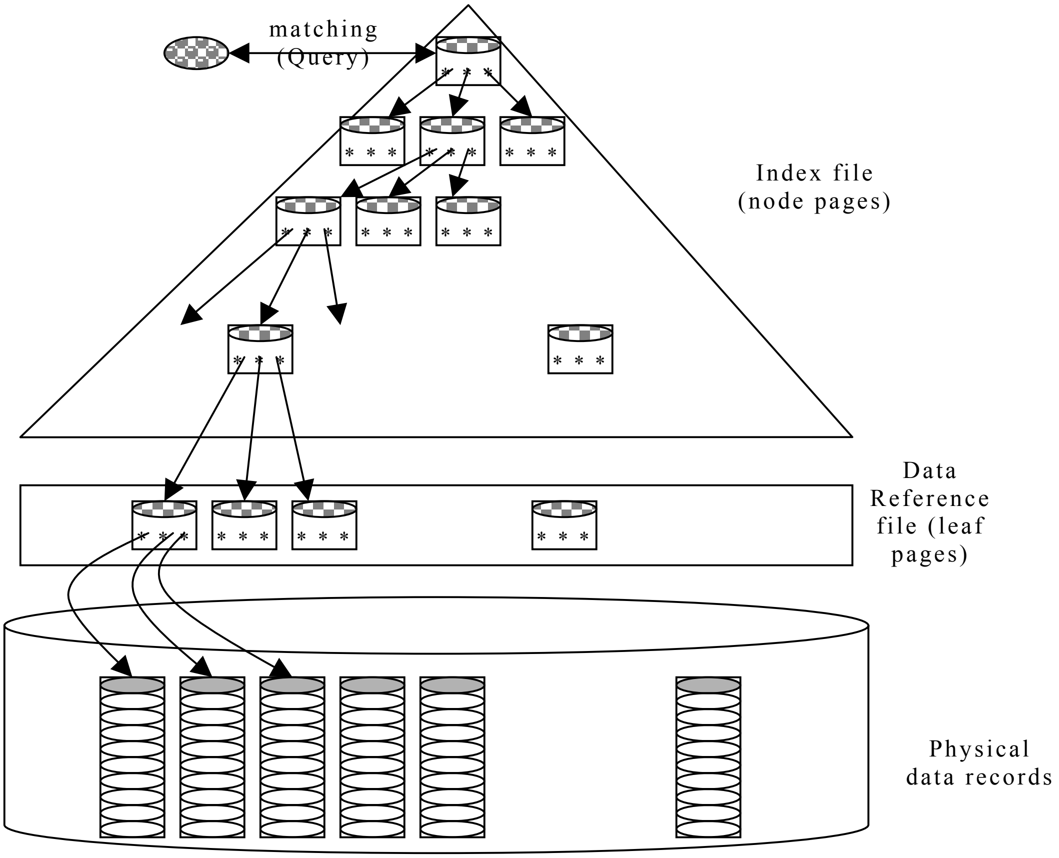


Figure 4.4: The Reference File Model

Chapter 5 The General Design

This chapter presents the design of a generic index system that can be adapted to different applications.

5.1 Design Rationale

System Modularization

The system is designed as an integrated set of modules. These modules provide all the necessary parts needed for implementing a complete system. Each of these modules, 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. This provides highly cohesive building blocks, a fundamental requirement for a good Object-Oriented design. Building blocks are also completely independent of each other. Each block can perform all its functions regardless of the types of the other blocks it is connected to, providing a very low coupling between these blocks, another fundamental requirement for a good Object-Oriented design.

Adopting the STL Concept

The STL, 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. We can build complete systems with the majority of its building blocks obtained from the STL library.

Code Reuse

A large number of building blocks already exist with a complete implementation on hand. This dramatically reduces the time needed for the implementation for many large systems where a great percentage of the code is simply imported from the STL.

Interface Reuse, and Combinatorial Composition

Each building block belongs to a group such as container group, iterator group, …etc and 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. The interface of these blocks is carefully designed to seamlessly connect to other blocks of the same or a different group, giving an endless number of possibilities of arrangements to achieve the needed system design. Some of these blocks do not perfectly fit together. This would imply that they cannot be connected or that their interface needs to be modified. This would, however violate the concept of generality and implementation-independence. Some extra building blocks are specifically designed to solve this problem in an elegant way. Without changing the standard interface of any block, some incompatible blocks can be made compatible using adaptors, a type of building blocks designed to smooth out some interface incompatibilities, thus increasing the number of possible combinations between the blocks.

Implementation Independence

The system design was made regarding only the interface of all the building blocks without any implementation details. This implementation-independent design frees the system from any unnecessary constraints, needed only for a later implementation, producing a simple abstract design.

Efficiency

All the STL building blocks used are written with the most efficient implementation possible, therefore allowing the system itself to be efficient. The efficiency, as a fundamental issue in any new system, is addressed as a design goal of STL.

System Flexibility

The framework has a flexible design by adopting a complete building block replacement policy. No building block is made mandatory to the design. In other words all the blocks that make up the system are replaceable. This gives the designer the freedom to isolate and replace any one or more of these blocks with no- or minimal impact on the system. The design can be adapted to the needs of any application by simply changing some of the building blocks. This was made easy since the building blocks are highly decoupled.

Wide Rage of Complexity

Even that the available generic blocks cover a wide range of applications; other specialized systems can still be implemented when the existing blocks do not cover all their functionality. The user can add any specialized building block to the system to replace an existing one, provided that the new block has the same interface. This allows for a minimal impact on the system since the user-defined block will seamlessly integrate with the other blocks eliminating the need to do a completely new system.

Default Values for Simplicity

In order to keep the system simple, only the necessary building blocks are exposed to the designer. Default building blocks are placed automatically in the system to do some of the necessary work without exposing them to the designer. Only when the designer wishes, are these default blocks replaced with other generic or even user-defined ones.

Favoring Templates over Inheritance

Inheritance is a relatively old concept in C++, compared to templates. Inheritance produces what can be seen as vertically-built or deep system, which can, depending on the design and implementation, compromise the system run-time performance, a fundamental measure in database system. Templates, on the other hand, produce a horizontally-built or shallow system, which would, in general, have a better run-time performance than a comparable system designed with inheritance. Our design was built using templates extensively with minimal inheritance. This does not mean that the use of further inheritance is excluded from the system. It is always possible to inherit from the existing building blocks to produce new blocks with added functionality.

User-friendly, Readable Code

The system was made up of standard blocks. Each of these blocks has a clear standard interface and a well-defined functionality. This makes all the blocks easy to understand. Also new blocks may be added with the same look as standard ones. The code will then add up these blocks with some extra lines of code working as glue between the blocks, without masking their interface. This will make the code more readable that a specialized code that does not use standard building blocks.

Ease of Modification

For programmers, modifying a code written by someone else (or even by themselves after some time had elapsed) can be a nightmare, especially if the original code was poorly designed and/or insufficiently commented.

We opt for ease of modification for any design using the framework by having a completely modular system design. This allows any programmer modifying the system to focus on the modules to be changed without the need to understand the details of the whole system.

Sustained Code Quality

A neatly designed and implemented code can quickly deteriorate in quality after few patches and modifications.

The system modularity allows for an endless number of modifications and patching by simply replacing some blocks in the system without affecting the system quality. The system quality will not deteriorate by adding scattered patches all over the code because modifications are mainly done by replacing modules, keeping code quality the same during its lifecycle.

5.2 System Structure

5.2.1 The Basic Components

The system will be built using STL components. These components will provide the necessary index structure, and manage the access and storage of the index.

The index is a container that provides an iterator to its contents. The only way to interact with the container is through its iterator, which provides controlled access to the elements of the container.

The index container will be an index of pages, so the elements that the index stores are of type page. They are passed to the index as template type in the implementation phase. This makes the index independent of the page design and can work with any page type.

The index iterator is therefore iterating through pages, one page at a time. This is exactly the database concept of indexes: *paged indexes* to facilitate access and improve performance.

The index uses an index allocator to manage the storage of its own elements, the pages. In practical application the index exists permanently on non-volatile storage, like a hard disk or other mass storage media since it is normally too large to fit entirely in memory. This means that the container will use a specialized allocator that takes the responsibility of retrieving the page from the storage into memory for access and controlling the different objects accessing the same page simultaneously. This will ensure data integrity by applying a suitable access and locking policy (pinning the page). After the pages have been modified, they also need to be updated in the physical storage (flushing the page) by the allocator. A default, in-memory allocator is provided for simple applications where the whole index can fit in memory at one time. It is up to the system designer to use it or override it by providing a storage-dependent specialized allocator.

.

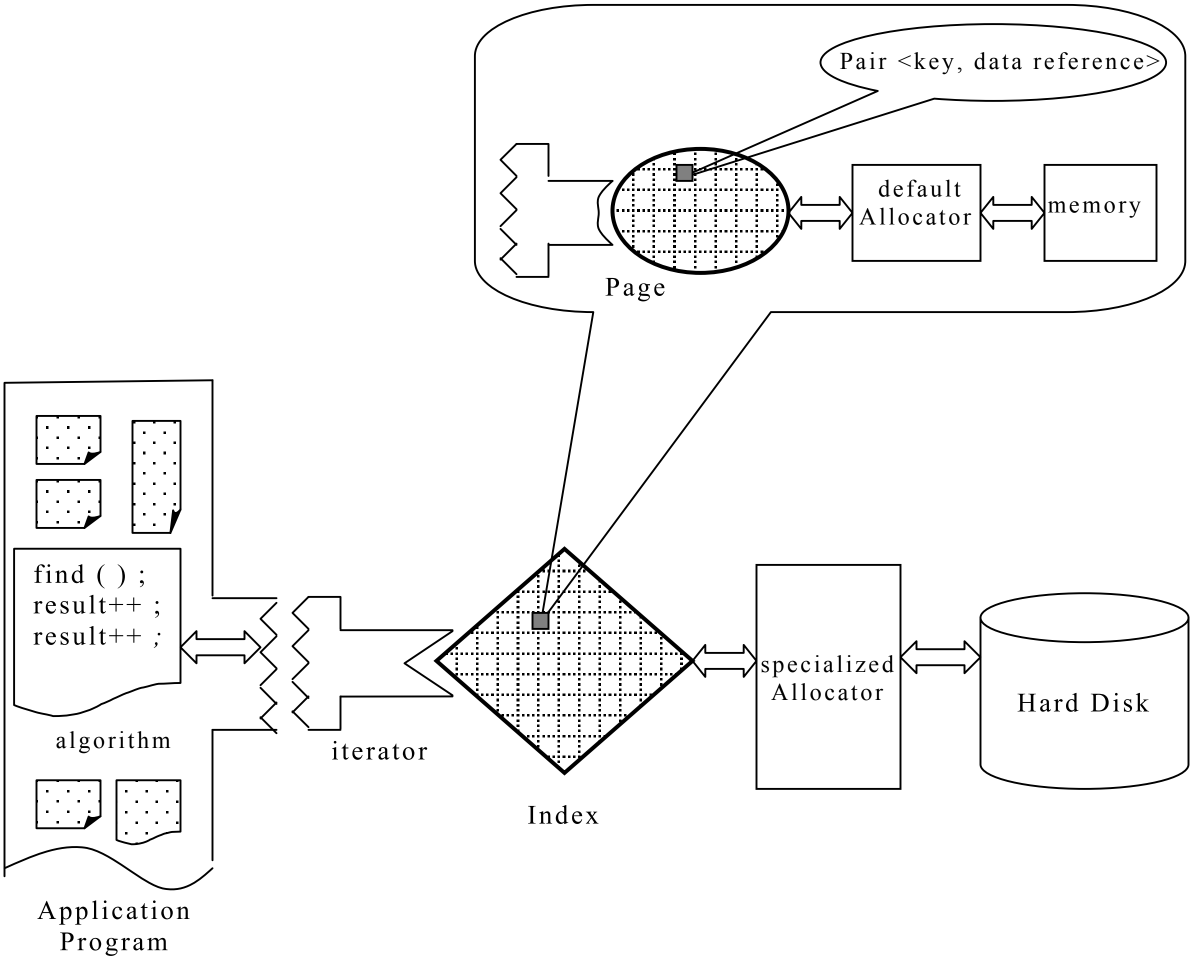


Figure 5.1: The Components Layout

This separation between the container and the allocator allows for the system to be completely independent of the physical storage of the index. The container uses the standard allocator interface to ask for storage services without any knowledge of the real storage dynamics

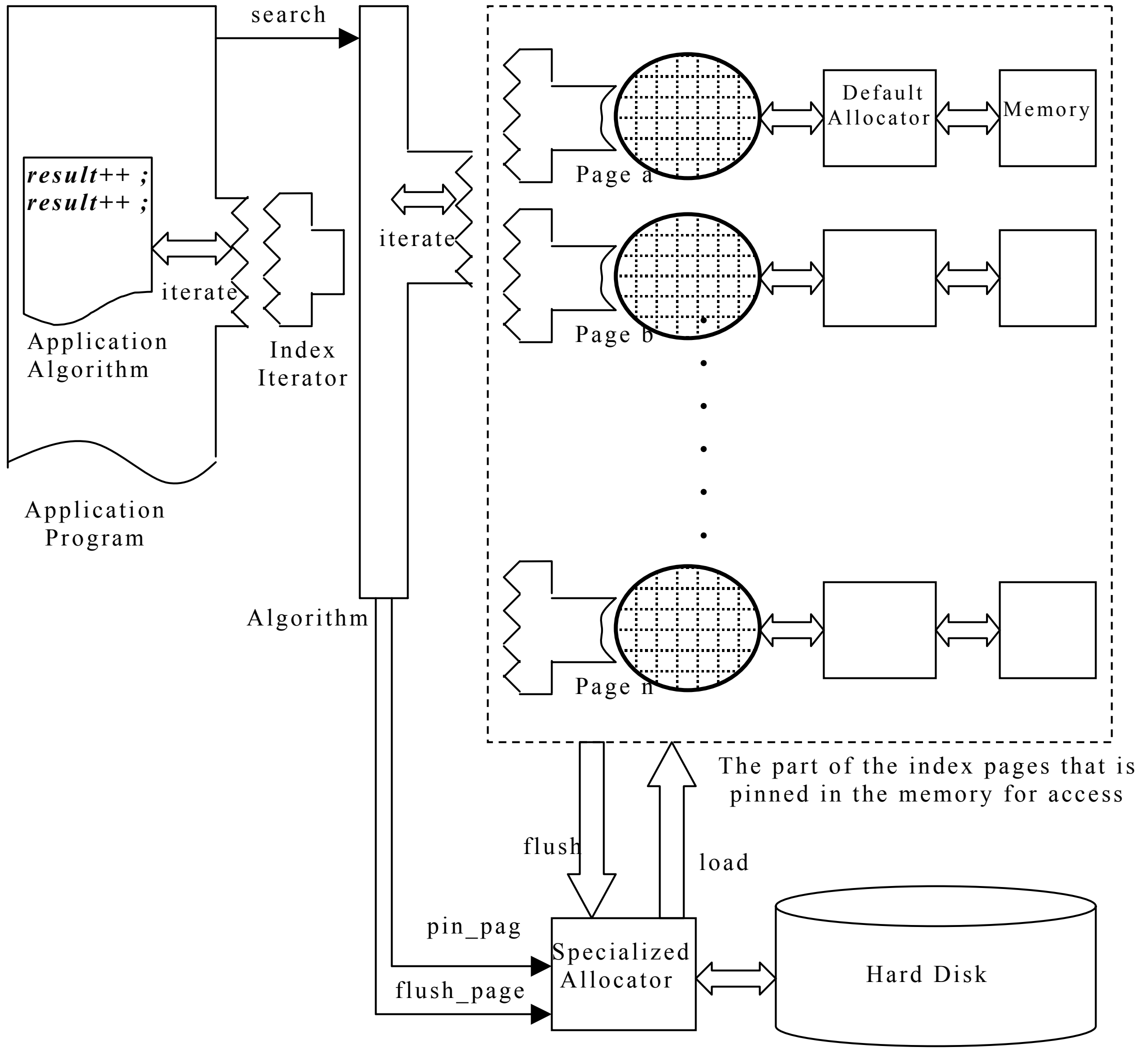


Figure 5.2: Detailed System Layout

The page is also a container on a smaller scale. While the index is a container of pages that typically resides on hard disk, the page itself is a container of pairs of key and data references. Again the page is made independent of the key and data reference type by having them as two templates passed to the page in implementation phase. Therefore the same page can work with any key type and data reference type that are passed to it. The page typically resides in memory for use. So whenever a page is needed, it is retrieved from the hard disk into memory. At this point the page can perform its tasks of searching for a key in its contents, accepting new entries, deleting some existing ones, etc. This design produces a simple system structure without affecting its complexity level or extensibility. The basic modules are shown in fig. 5.2.

5.2.2 The Class Diagram

Figure 5.3 shows the class diagram of the index system

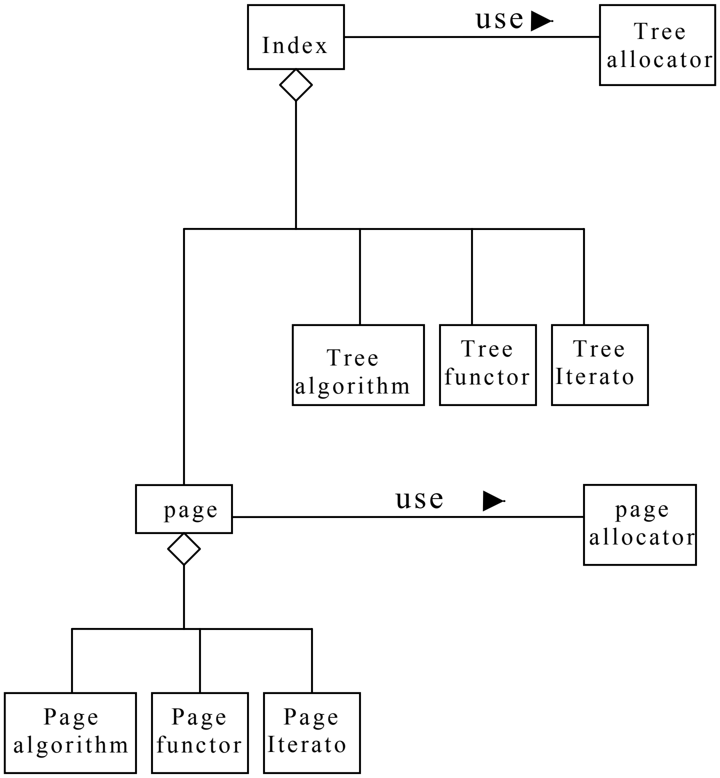


Figure 5.3: The System Main Classes

5.3 The Design Dynamism: Modules Replacement Scheme

The framework provides a modular design that can be adapted to different applications requirements. The resultant design is an index that helps an application access storage to get some useful information. Depending on the application domain and the type of storage, the system can vary in design. A simple layout will generate a system that uses its internal components (the built-in set of functions) to operate. The application uses the internal algorithms (like find ( ) and insert ( )), which accesses the container elements via its internal iterator and apply the internal functor to them to decide on the candidate elements to a key provided by the application. The container uses a default allocator to manage data storage and retrieval.

Figure 5.4 shows this layout. As we can see all the components of the container class can be replaced by other external components.

In order to achieve this flexibility of replacement of objects without affecting the run time performance, an interface is provided for each of the system classes as in figure 5.5. The built-in classes are then an implementation of these standard interfaces. Any new class that is meant to replace an existing one must satisfy its exact interface, so it should inherit the necessary functionality of the original object by simply inheriting its interface. This is just an abstract inheritance that has no effect on the run time performance.

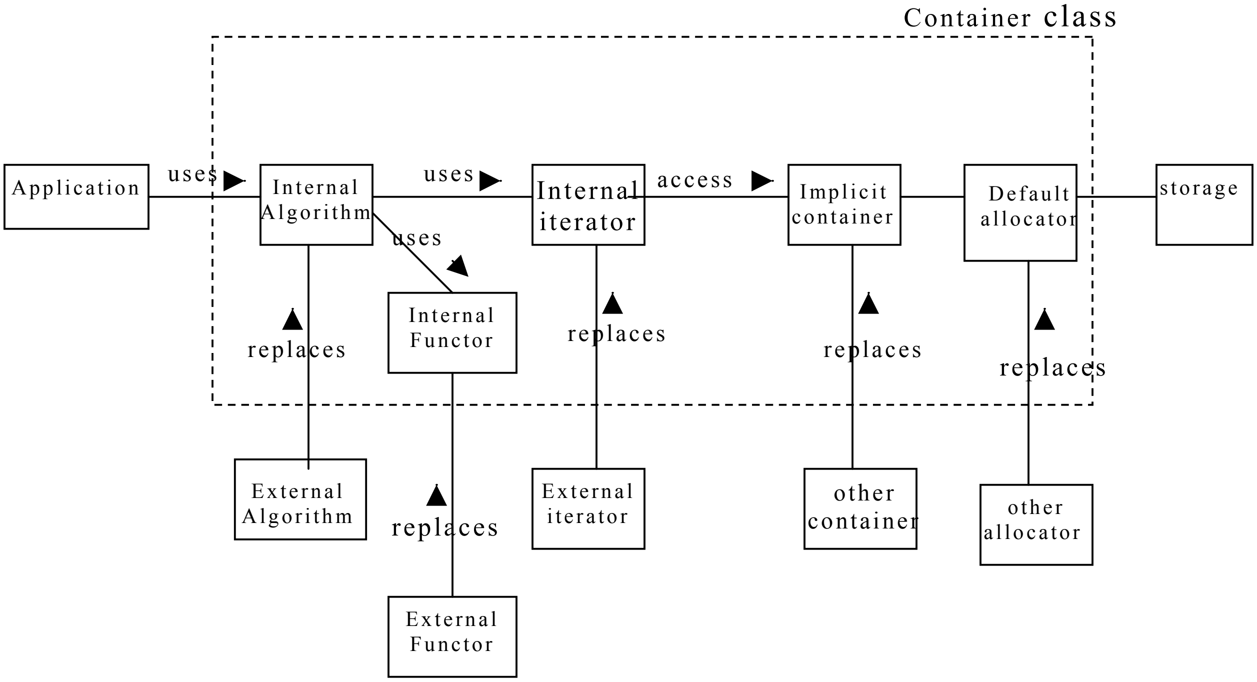


Figure 5.4: Components Replaceability

The basic interfaces of the container, iterator, and algorithm modules are shown in figure 5.5.

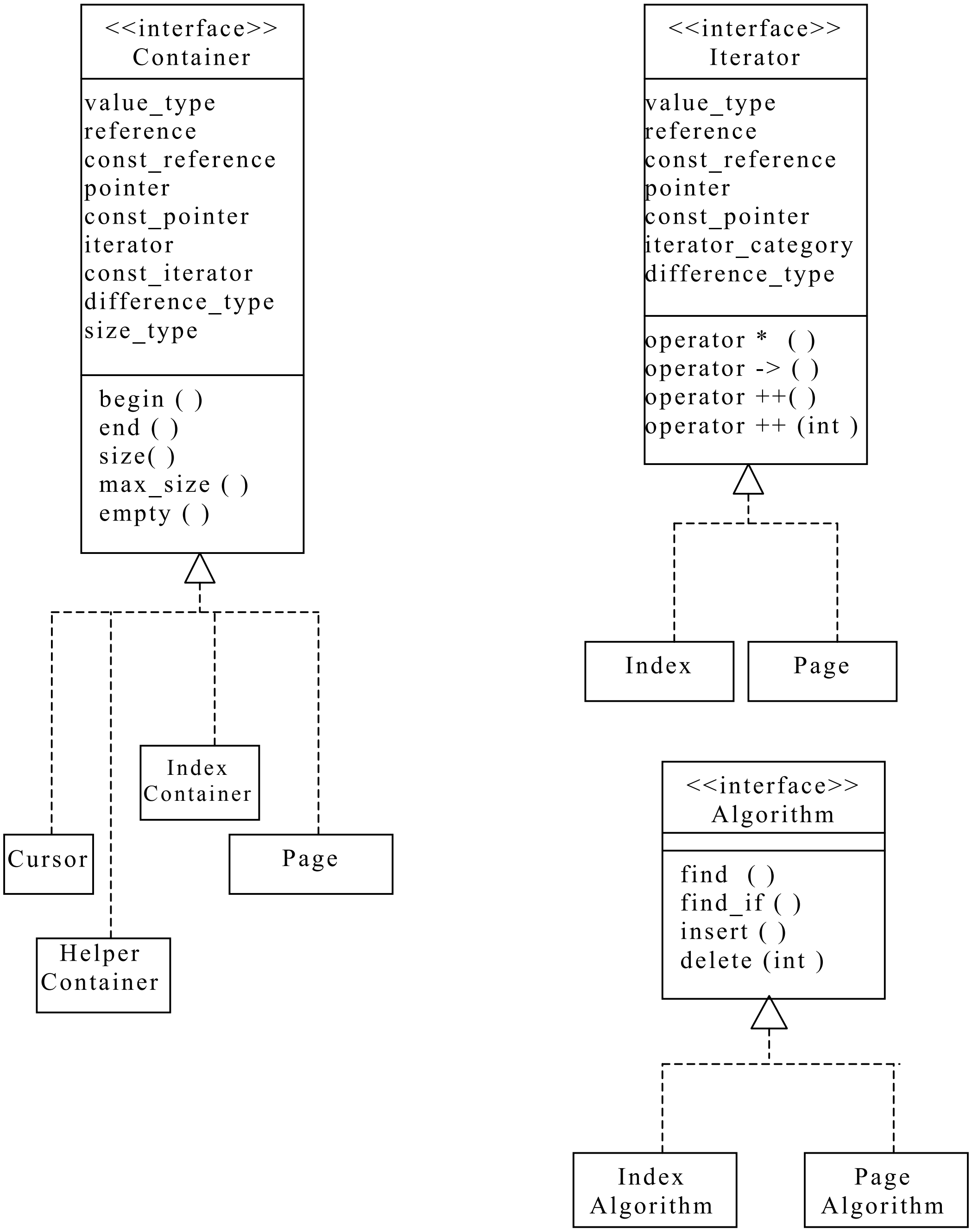


Figure 5.5: Detailed View of the Main interfaces

5.4 Exception Handling

The system manages part of its functionality through exceptions. An ordinary insertion operation would result in adding the new entry to the page and probably adjusting its key with no further impact on the system. Considering the fact that the index starts with its pages partially empty, the scenario of adding new entries will continue to the point where the page becomes full and thus it becomes necessary to split it. Assuming a comparable amount of deletion along with the insertions, the page will take much longer time before becoming full. Once the page becomes full during an insertion operation, the operation will throw an exception page_full as a signal. The index class will catch this signal and start the procedures of splitting that page.

A comparable situation can happen during a deletion operation. Typically a page starts only partially full, and can support some extra deletion operations with no impact other than deleting the entry and probably readjusting the page key. Adding to that the counteraction of adding some entries to the same page, the page size will fluctuate within the allowed range without impact outside the page. Once, however, the net amount of deletion to a page becomes dominant, the page size can shrink to below the minimum allowed fill factor (typically 50 %). At this point, during the critical deletion operation, the page will throw an exception page_sparse. The index will catch that signal and start the procedures to fix the sparse page.

One of the first steps to fix a sparse page is to try to borrow some entries from one of its siblings. Redistributing the entries of two pages equally between them will solve the problem with a minimal impact on the index. Only the two pages involved will be modified and possibly their keys at the parent nodes as well. If the two siblings of the sparse node were not suitable for borrowing, the operation check_to_borrow ( ) will fail and throw an exception none_to_borrow. The index will catch this signal and start the procedures of merging the sparse node with one of its siblings.

Figure 5.6 shows the signals received by the index class and the hierarchy of signals sent by the page class.

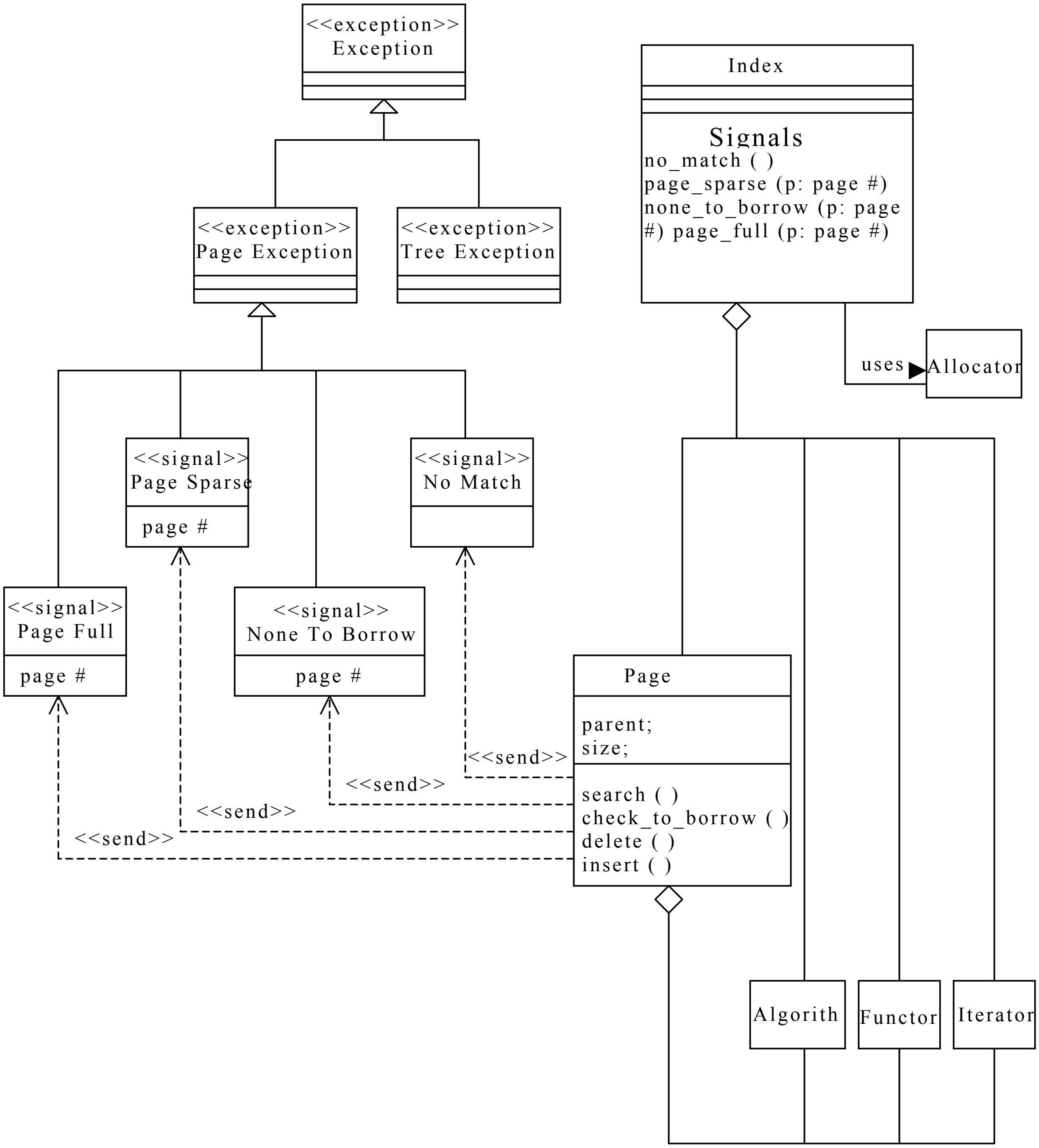


Figure 5.6: The Exception Class Diagram

5.5 The General System Interface

Page is an associative container that manages its data (page entries), which are pairs of key and data reference and allow for searching these pairs for a given key. Page is an STL-like container that must conform to all STL interface characteristics. The page container can be implemented using one of two implementation criteria: Interface realization or Implicit container inclusion.

Interface realization

Interface realization is implemented by inheriting the public container interface from STL container class and implementing the class methods completely. This will provide a more specialized code that is still conformant with the design. This is particularly useful when special time or space restrictions might apply.

Implicit container inclusion

For efficient implementation, the page container can be built on top of an existing STL container. The page will work as a wrapper class that has an STL container class as one of its data members (an implicit container). This will allow for masking the unneeded member functions in the STL class, and will save a lot of implementation time. The page flexibility will depend on the container used. This will give a more generic code.

In both cases we then need to add the needed functionality of the page. Many STL containers can be used as an implicit container, but we consider map to be the most suitable one for the following reasons:

i- It supports any user-defined keys (sequential containers can only support integers as keys)

ii- It supports unique keys (multimaps support multiple keys, if needed)

iii-It stores data entries as key and data (sets store keys only, if needed) iv-It has efficient storage and search time

All containers provide their own public functions (built-in algorithms like find ( )). They also provide public iterators and type definitions to allow for interaction with external STL algorithms like find_if ( ) or any new user_defined algorithm. Depending on the search algorithm of the index, we can adopt any of these searching policies.

-Each page keeps a reference to its parent page if depth first with no stack were used as traversal method or for tree maintenance purpose.

-If tree supported sequential access (like B+ tree), leaf pages will have references to both its left and right siblings as well.

-Each page has a method to extract its key (the page_key( )) and – possibly- to save it as a data member for efficient retrieval of it with some storage overhead. As with other computer science issues, it is a compromise between efficient dynamic behavior and efficient storage (time and space).

-The page will receive pairs of <Key, page &> as entries to store within the page.

-The page is capable of adjusting its page_key after an entry insertion.

5.5.1 The Page Container

The page container is invisible to the application. It is created and managed by the index container. The page stores a pair of (Key, T). Class T can be any user-defined type, typically a reference to another page. Class T will be referred to as the mapped object of type mapped type. The pair (Key, T) will be referred to as the value object of type value type.

Key is used to distinguish the different pairs. It can be any user-defined type. Key does not have to be part of the mapped object T, extracted from its contents, it can be added to it. Keys are unique (other containers allow for duplicate keys).

Compare is the functor used when searching objects to decide on the matching keys. The default values allow for simply using the built-in comparison methods as built-in queries, or adding external functors as new user-defined queries new matching criteria. The page provides its built-in set of functions (as standard algorithms), type definitions and iterators to its contents to interact with external algorithms and external iterators. Insertion / retrieval takes logarithmic time when using the STL map as an implicit container in the implementation because objects will be stored in the page in a tree structure. This has, however, no big impact on performance since the whole page container is meant to fit in memory as one page and is typically small (on block size) so the retrieval time is insignificant compared to disk access time. Figure 5.7 shows the page class. Appendix B shows some public and private members of class page.

```
template <class Key,
            class T,
            class Compare = less <Key> ,
            class Allocator = allocator <pair <const Key, T> >>
class page;
```

Figure 5.7: Page Class Signature

5.5.2 The Leaf Page Interface

Figure 5.8 shows the leaf page class. Leaf pages are different in the data type stored inside them (class T). They also have a true value for the private flag is_leaf, which is accessible through the constant public function is_leaf ( ).

```
template <class Key, class T,
            class Compare = less <Key>,
            class Allocator = allocator <pair <const Key, T> > >
    class leaf_page: public page<Key, T, Compare, Allocator>;
```

Figure 5.8: The Leaf Page

5.5.3 The Index Tree Container

The index is an abstract data type that can use different implicit data types in its implementation. As with the page design, the index container functionality can be either inherited from an STL container interface and undergo a complete implementation, or can be implemented by using an STL container as an implicit container in a new index container class. We follow the second approach. The index container provides methods that use some of the implicit container methods, but this is transparent to the user. The page container is meant to be small enough to easily fit into memory (a fundamental concept in the index page). The index container, on the other hand, does not have to fit completely in the memory. In typical cases it will not. The characteristics of the index container will decide on the implicit container used. Since index exists mainly on hard disk, inserting new pages can only be made at the end of the index file, or by using a recycled page (for efficiency reasons). No support for insertion anywhere else is needed from the implicit container. For the same reason, the index uses a special allocator to manage the storage on the hard disk (pinning the page). The access of pages is required to be a direct access. Given a page number (or offset) we expect to be able to read it directly without the need to access all pages before it. Therefore the implicit container must support random access using operator [ ]. Sequential access to the index pages would result in unacceptable performance. This means that we have two choices. We can use a sequential container that supports random access to its elements (pages) using some index number to identify each page (by translating it to the physical page address). Alternatively, we an use an associative container that supports random access to its elements (pages) using a unique key for each page, (again by associating it to a unique physical page address).

The Interface

The incrementing operator implementation in class cursor might have different implementations depending on the index type

1- It can be as easy as reading the next entry from the private vector component if all results were deposited into the cursor in one batch as in general domain batch search (breadth first policy).
2- It might include sending messages to an index page to iterate to its next entry as in linear domain sequential access.
3- It might include sending a message to the index to continue searching and locating the next entry as in general domain depth first policy.

For class index, it is more likely to use the default container than the default allocator, so allocator comes first in the template argument.

Figure 5.9 shows the signature of the query, cursor, and index classes, all included in the index class.

Public type members and data members:

Each user-defined container must provide a set of standard STL types that are internally translated to the equivalent container-specific types with typedef statements. This allow for the container to be easily integrated with other STL components.

This can be accomplished directly as public type members or by providing a struct traits to define these types. We follow the first approach.

```
template <class T>
class query : public Unary_function <T >
{
typedef  page <Key, T, Compare, Allocator> :: key_type   key_type;
//the query includes a page key in it.

T  operator ( ) (T a, key_type k ) const;
// return value must be convertible to boolean

}

template <class T>
        class cursor
        {
        public:
                T  operator = ( );          // for index to deposit results
                T  operator * ( );         // for application to dereference
                                           // results
                T  operator ++ ( );       // for application to pre increment
                T  operator ++ (int ); // post increment
                begin ( );
                end ( );


        private

                vector <T> V

        } //end class cursor

template <class T,
            class Allocator = allocator <T>,
            class Container = vector <T>  >
class index;
```

Figure 5.9: The query, cursor, and index Classes

Chapter 6 Design for Linearly Ordered Domain

Before describing details of system design, we will explore some properties of linearly ordered domain. This will clarify the design decisions made for a linearly ordered domain index.

6.1 The Concept of Linearly Ordered Domain

Data in this domain is stored in a sequential order. The data is at least Less-Than Comparable, meaning that it must be possible to compare two objects of that type using the operator < and the operator < must define a consistent order [Aus99], so it can be at least partially ordered. For example, for two elements A and B, we can decide which one is less than the other, and store them in their order. If we cannot decide, they are said to be equivalent and we store them in the same order as one group. So ordering here can be done only partially as for two equivalent elements, there is no specific order even that they are two different elements. But at least between groups, there must be a complete order. No two groups can be equivalent unless they are the same group.

An example is when storing information about a number of cars. We can order them according to their model year. For every two cars, we can say if one is older than the other and put them in order. If two cars have the same year model, we cannot order them according to this criterion. They are equivalent, even that they are two different cars, and we put them in one group.

Data can also satisfy the more stringent Strict Weakly Comparable concept, generating a total ordering for the data. This means that for two elements A, and B, we must be able to decide their order. They cannot be equivalent anymore as long as they are two different elements. Each element is equivalent to itself only, so we reduce the number of elements per group to one and the equivalent concept becomes equal. An element is equal to only itself.

In the previous car example, ordering the cars by their license plate will give a total ordering. No two different cars can have the same license plate, and for any two cars, we must be able to decide which one is less than the other. For every group of license plate, we have only one car.

This implies that the search for a specific key will find a specific element (or a specific group of elements) that satisfies the search comparison criteria. This element (or the first element in case of a group result) will be returned as the result for the search.

In case of a group result, the beginning of the group is sufficient to give the necessary data. Since data is ordered, the rest of the group can be easily located. This mandates a small addition to the generic page concept: Each leaf page should be physically connected to the next leaf page. Even that pages are not necessarily physically ordered, it is easy to find the logical order of leaf pages regarding their contents. Then it would be easy to link each page with the page that has the immediately next sequence of data. In the design, each leaf page will therefore have a reference to the page that follows it in the sequence. Acting like a linked list data structure, inserting a new page between two pages requires some modifications to keep the logical sequence in order.

Linking each page to the previous page as well can make a further addition. It will then act like a doubly- linked list.

We not that leaf pages act like linked lists when it comes to sequentially accessing the data, But, unlike linked lists, they also provide the functionality of direct access when performing a search.

It is clear that the search technique can be composed of one single dive downward through the index, with no stack needed for the lookup process.

6.2 Class Diagram

For a linearly ordered domain, the index will have an index algorithm class to provide the index functionality (search, insert, delete, etc). This algorithm will be used by the application to demand services from the index. The index will have two types of iterations, implicit internal iteration, done without a real internal iterator object, and explicit external iterator; the cursor.

Implicit internal iteration

This iteration will provide a special kind of internal tree traversal. It will generate a page reference and simply access the page. No iterator range (begin, end) or functionality is needed. Generating a page reference will be based on one of the following tree iteration concepts:

Concept 1: Iterating using a key:

> The iteration will start from the root page as the *begin ( )* position. The incrementing process *operator* ++ must be based on a search key; no arbitrary incrimination is supported since there is no specific next element without a search key. The next page in the index will be decided by interrogating the index current page using the key. This interrogation will provide the next element (a page reference), where the iteration can proceed. This will make it the new current element.

Concept 2: Iterating using a page reference:

> The page reference can be obtained by different ways.
>
> One way is by interrogating another page using a certain key. This case will be invoked internally by a search operation that uses concept #1 iteration without interaction with the application.
>
> The page reference can also be obtained by interrogating a leaf page for the next page reference (in case of sequential access). This case will be called externally by a cursor iterator that is accessing the result data sequentially.

> The page reference can also be obtained by interrogating any page for a parent page reference. This case will be called internally by an insert or delete operation that needs to fix a parent to a modified page.

All these iterations are invisible to the application. The only iteration done by the application is iterating through the result data using a cursor. The index provides the results to an external iterator (the cursor), which will then be able to iterate through the index leaf pages contents as well as iterating from one leaf page to the next to access successive results.

Explicit external iteration; the cursor:

The cursor is a special kind of external iterator. It is created by the application when searching for some data. As the index searches and locates the beginning of that data, the cursor can iterate through the successive entries (on request from the application). Here, unlike internal iterator, the next element is sequentially defined; it is the next page entry (a pair of key, reference). As the leaf page is depleted, the cursor is capable of jumping to the next leaf page to continue iteration. Here the next element is also clearly defined; it is the first page entry in the following page. The cursor will use the page iterator to go within a leaf page, and will use the index iterator to go from one leaf page to the next. This is invisible to the application. The application will be able to iterate through resultant data continuously as if they were inside a sequential container. The index will have pages as its stored elements. The index will use an index allocator to store the pages on an external storage medium that holds the index pages (typically the index size is too large to fit in memory in one shot). An in-memory allocator can be used if the index can fit in memory. The index will use an index functor (query) that will be passed to the pages to check for matching entries. This query is external to the page, but built in within the index and passed to the page to use it. This query will carry out the equality search. So the linear domain index comes with a built in query. Any other query can be passed to the index by the application and be used by the page. The page will have page algorithms to decide on the policy of searching, inserting, and deleting within the page. An in-memory page allocator will carry out the memory allocation and deallocation operations. The page will have a page iterator to iterate through the contents of the page depending on the contents layout. The application that uses the index will use the algorithms provided by the index to build, search, insert, or delete entries. The application will have a cursor, which the index will be made aware of. The index will pass the results to the cursor, and the application will then access them from the cursor.

Note that some components are missing, like the page functor class for example, because they are not needed in this design. This does not, however, exclude the possibility of adding it to the design in a later modification.

6.3 System Behavior

In order to easily track the behavior of the system using sequence diagrams, we will use a specific concept for numbering the pages of the index. For simplicity, we will assume each page to have a maximum of 10 children (the same concept can apply to any number of children). The root page, at first level, has the number 0. The children of root page have numbers 00, 01, 02, 03, … up to 09, where the first digit (the 0) is the parent number, and the second digit is the child number (0 through 9). Each of these pages also has children numbered by post fixing the child number to the parent number. For example, page 03 has children numbered 030, 031, 032, … up to 039. This also means that the number of digits in a page number gives the level at which the page is. Of course, it is possible for a page to have less than 10 children, so for example if page 044 has 8 children (0440 to 0447), then 0448 and 0449 will simply not exist. We will not use these numbers to implement a real index, so we will not consider unneeded details like a missing child in the middle between two children (for example as if 0355 was missing, the next page to 0354 would be 0356). We will simply assume that no missing pages exist in the middle. Each page will have a reference to it carrying its number and a key with the same number as well. For example page 0338 has reference P0388 to it and its key is referred to as key0388. From the page number we can get all the information we need; its immediate parent, its ancestors all the way back to the root, its siblings (if existed), and the possible children. For example page 05380 is the first child of page 0538, which is in return the 9th child of page 053. Page 04256 has the two siblings 04255 (previous), and 04257 (next, if existing)

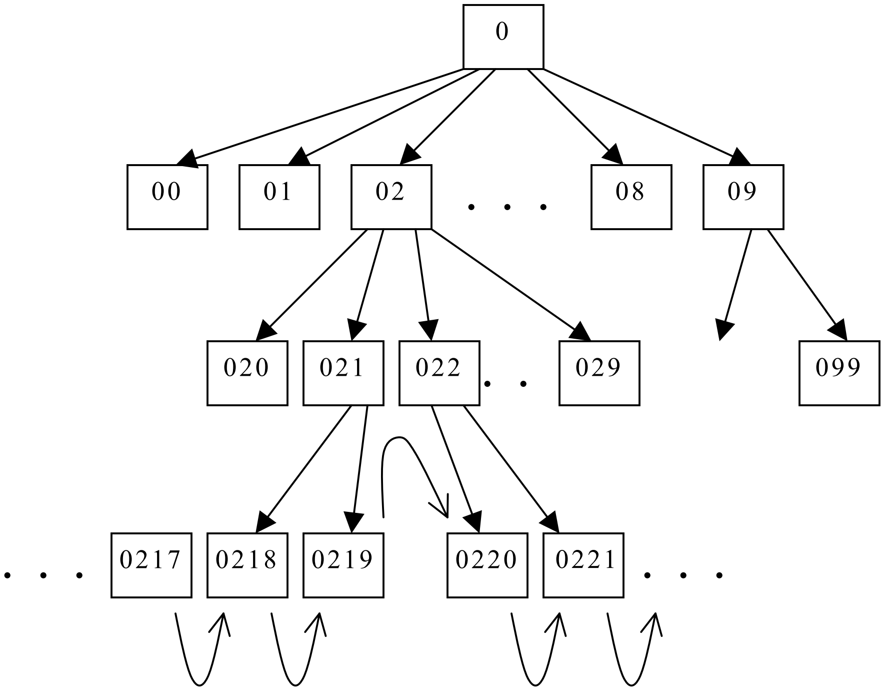


Figure 6.1: The Numbering Convention

Page 0219 has the left sibling 0218, while it has no immediate right sibling under the same parent (the parent would be page 021 with 10 children from 0210 to 0219). However, we can access the next sibling by finding the next sibling of the parent (page 022) and access the first child of it (0220). We followed the tree paths, and did not just add (0219+1=0220). This concept, albeit pure theoretical, can simplify our tracking of the system behavior across the index pages. Note that all page numbers will start with 0 as the index tree has one root only, page 0.

6.3.1 Scenario for “search” Using an Internal Query

Search is the fundamental operation in 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 insert and delete 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 search function comes in two fundamental forms: using a key or using a query. Search by a key is the basic form. It uses the internal comparison operator (the built-in functor) inside the index to find the first data entry whose key matches the searched key. As shown in figure 6.2, the application starts by (1) creating a cursor object to receive the search result in it. Then it (2) sends a search request to the index algorithm providing information about the key to search for, and the cursor to put the results in. The Index starts the search operation by creating a page algorithm object; this object has the semantics of searching a particular page for a key. From the STL concept, this object needs to be passed three things: a key, a page iterator, and a functor object. The tree creates a functor from the built-in functor class, and an iterator to the root page and passes them along with the key to the page algorithm object. This latter object will use the iterator to go through the root page entries, one at a time. With each entry retrieved, the page algorithm will extract the key, and check it against the searched key. If the check returned false, the page algorithm discard this entry and repeat the same process on the next entry. Once an entry's key evaluates to true, the page algorithm will extract the data part of the entry (a page reference) and send it back to the index algorithm. The index algorithm will remove the root page iterator if it is not needed any more, create another page iterator to the page returned by the first page search. The Page algorithm object will then be passed the three parameters: the searched key, the iterator to the new page, and the built-in functor to check the retrieved keys for a match. The same process is repeated for all page entries until an entry that satisfies the key is found and its reference part is returned to the index. The index will keep retrieving the pages and interrogating them for further pages down the tree until a leaf page is reached.

As the leaf is interrogated for the key, and the data part for a matching key is found, the data part is (2.1) returned to the cursor object. The search carried out by the index comes to an end.

Now the result for the search is in the cursor (in the form of a page iterator pointing to the page entry that matches the key) and the application can (3) retrieve this result by (3.1) instantiating the iterator object using (*result) and incrementing the iterator to the next result using (iterator ++). As the cursor is incremented, it will (3.2) iterate to the next entry in the index. The next entry is (3.3) sent to the cursor and then (3.4) to the application. As the application iterates to the next result, another iteration cycle (3.5), (3.6), (3.7) takes place.

Figure 6.3 shows a high-level sequence diagram of this interaction. Figure 6.4 shows a detailed sequence diagram.

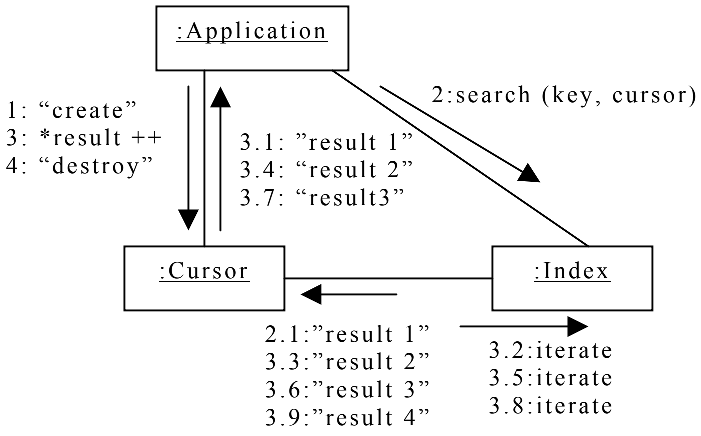


Figure 6.2: Hi-level Collaboration Diagram

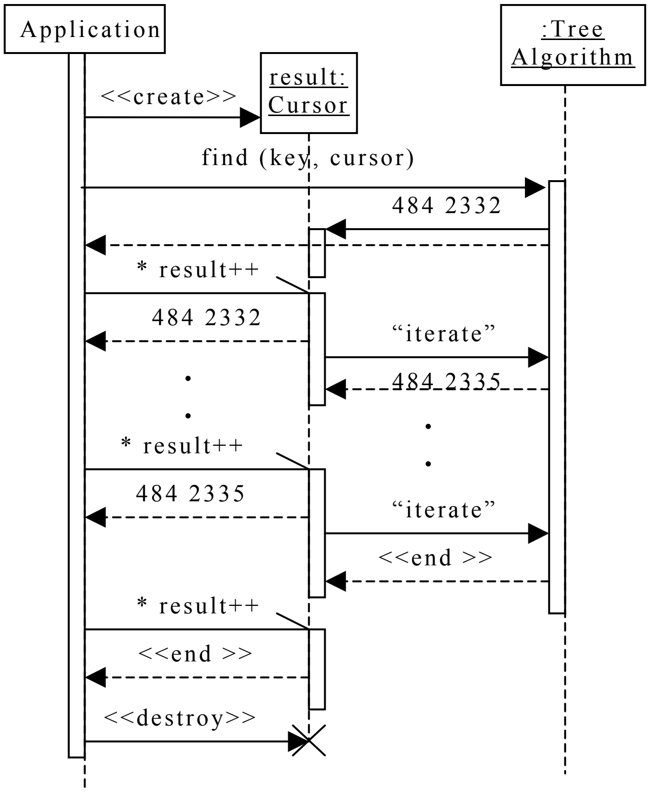


Figure 6.3: Hi-level Sequence Diagram

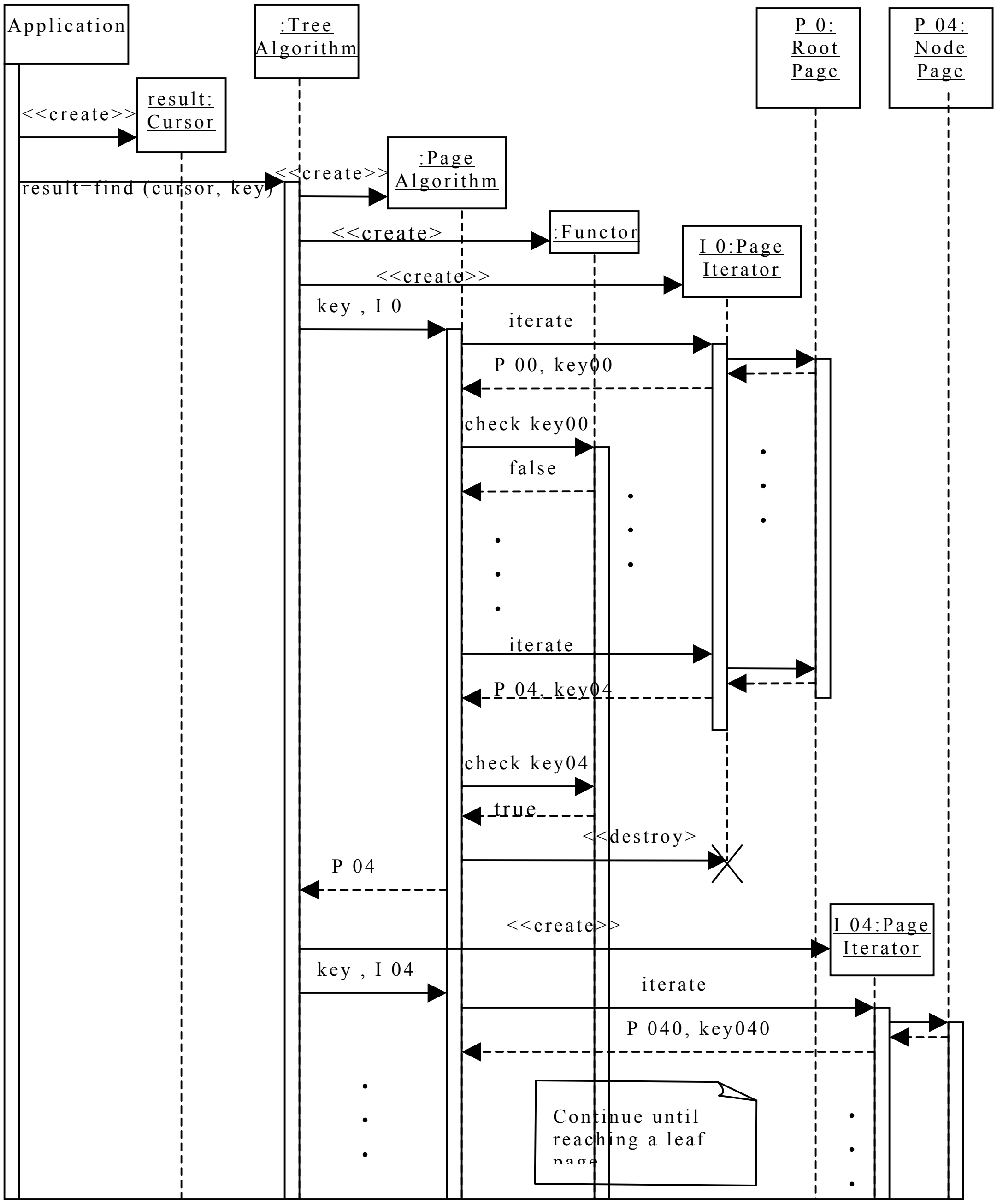


Figure 6.4: Using Internal Query

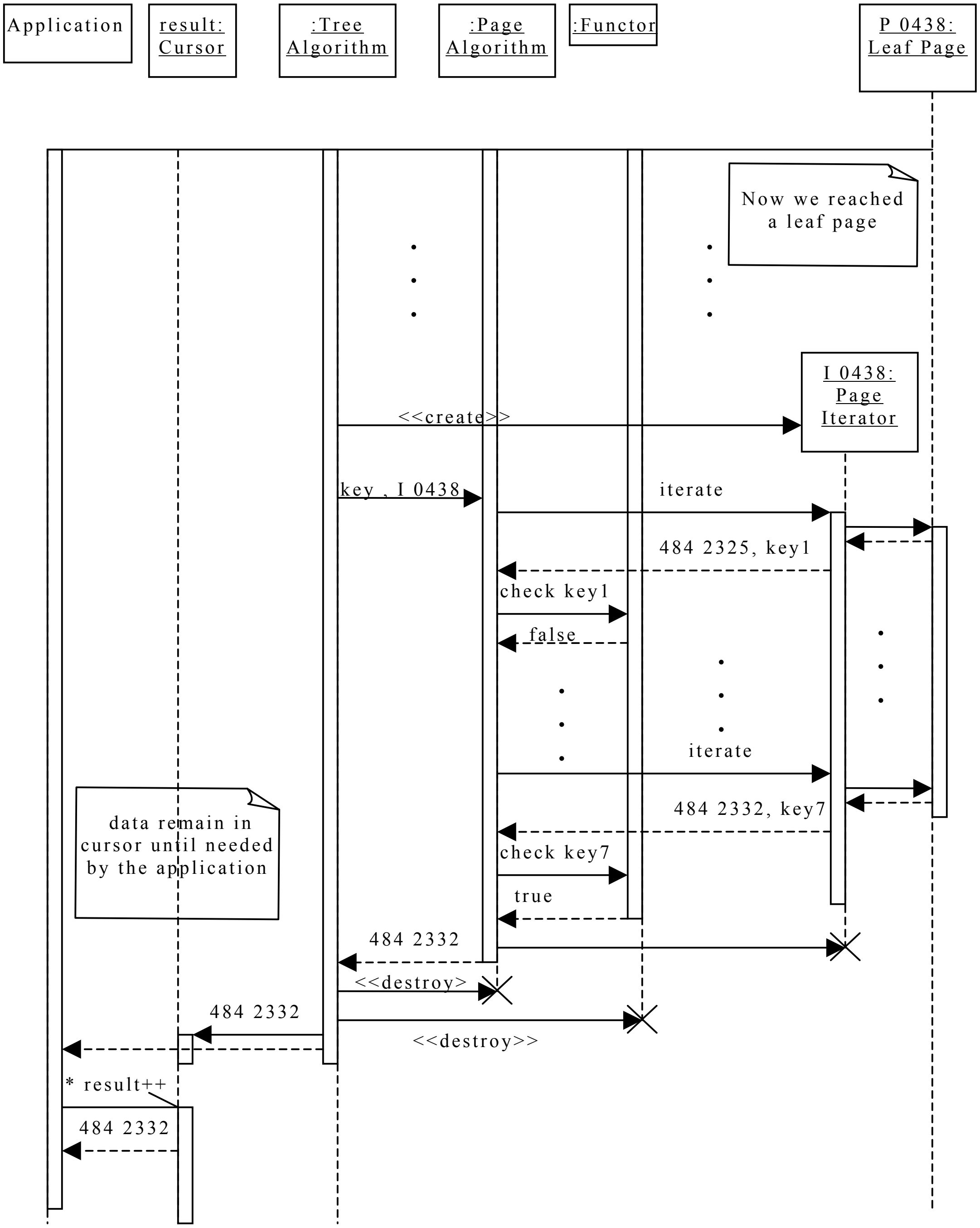


Fig 6.4 (cont.): Using Internal Query

6.3.2 Scenario for “search” Using an External Query

We can use the index to search for a key using an external query instead of the internal one. No major change is needed. In database concept, search using a query is a search for certain elements that satisfy the query. All we need to do to the system is simple: pass to the page algorithm object an external functor that resembles the query, so that when the page algorithm retrieves page entries, it will check them against this external functor instead of the internal one. From the page algorithm’s perspective, nothing much has changed. Only the functor used for the check is different. The scenario with these change reflected on it is shown in figure 6.5. Now the application creates the query object of its choice (it must, of course, conform to the standard functor interface by inheriting the abstract interface of a functor). The query object will contain the searched key inside it as well as the comparison criteria. It can check any key passed to it. It will be able to check the keys sent by the page algorithm and return true or false to the page algorithm. The index algorithm will make the necessary adaptation. As it receives a search request with a query instead of a key, it will create the suitable page algorithm object and passes to it the two objects: the root page iterator and the query object. The page algorithm will retrieve page entries as before, and check their keys against the external query, and select the matching ones. We need to notice one subtle difference between the algorithm of internal and external queries. For internal query case, the algorithm receives two parameters: the searched key along with the page iterator. It will then use the internal functor (the default one), extract the keys from page entries and send the two objects extracted key and searched key to be checked by the internal functor. For external query case, the page algorithm receives two parameters: the page iterator and the external query object. No key is sent as the information about it is now inside the query object.

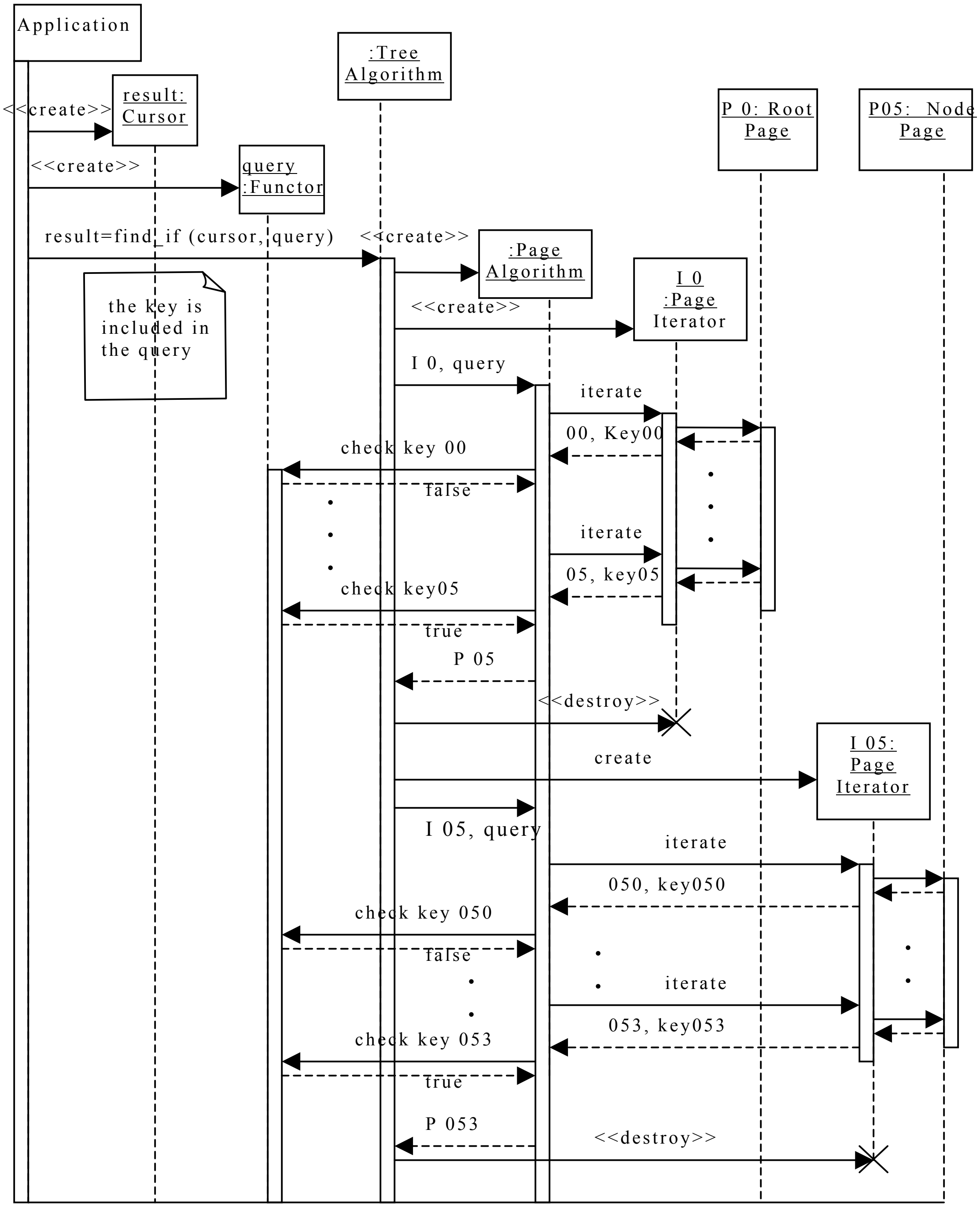


Figure 6.5: Using External Query

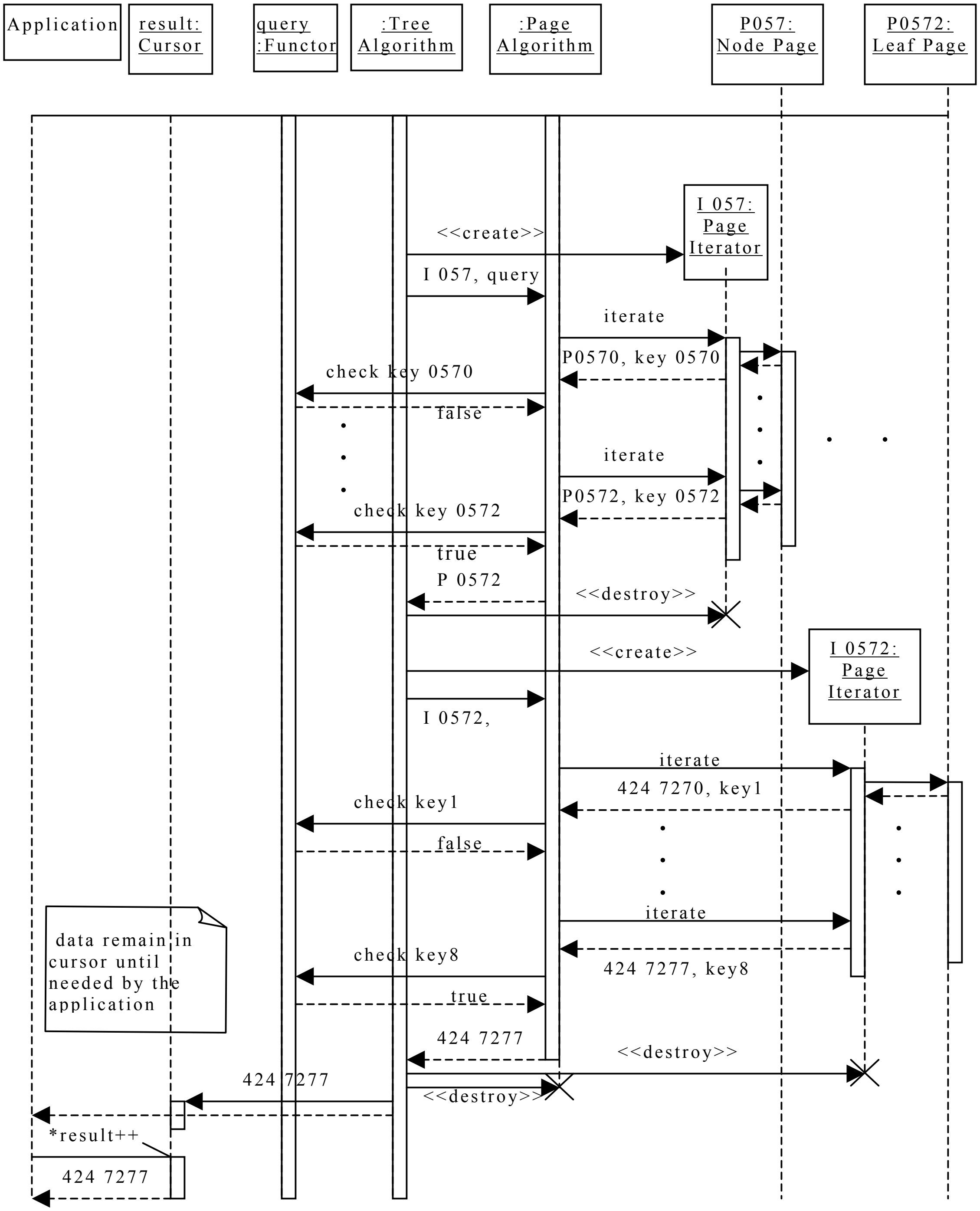


Figure 6.5 (cont.): Using External Query

6.3.3 The Cursor Behavior: Iterating Through the Search Result

The cursor object is an iterator object that has the capability of iterating through an index page and being dereferenced to give access to the object it references. As the index returns the candidate leaf page having the needed information, it will in fact return an iterator to the location of interest inside it. The cursor will therefore be referring to a location in a particular page where the information of interest starts. The application can then dereference the iterator and get the information.

Having the same capabilities as a page iterator, the cursor allows the application to iterate through the page sequentially using the operator ++. This is no done directly from the application to the data without asking the index. At a certain point, the cursor might come to the situation where the page contents are depleted and the application is still asking for the next element. Here we can see the difference between the page iterator and the cursor. A page iterator will return null value as the last entry of the page is accessed. The cursor might want to continue to the next page if data was sequentially ordered. For the cursor, data is a linear sequence and the end of a leaf page is followed by the beginning of the next leaf page. This mandates a slight change to the page iterator: as the page contents are depleted, the page will not return a null value but rather a reference to the next leaf page. As the cursor receives this result while trying to access the next page entry, it will be understood and the cursor will go to the next leaf page using that information and start accessing its contents by iterating sequentially through it upon further requests from the application.

This scenario continues until the application ceases asking for further iterations or the last entry in the last leaf page was accessed.

We should notice that this scenario applies for linearly ordered domains only, where data is sorted and stored in a sequential form.

For non linear data, other scenarios apply, as will be shown later.

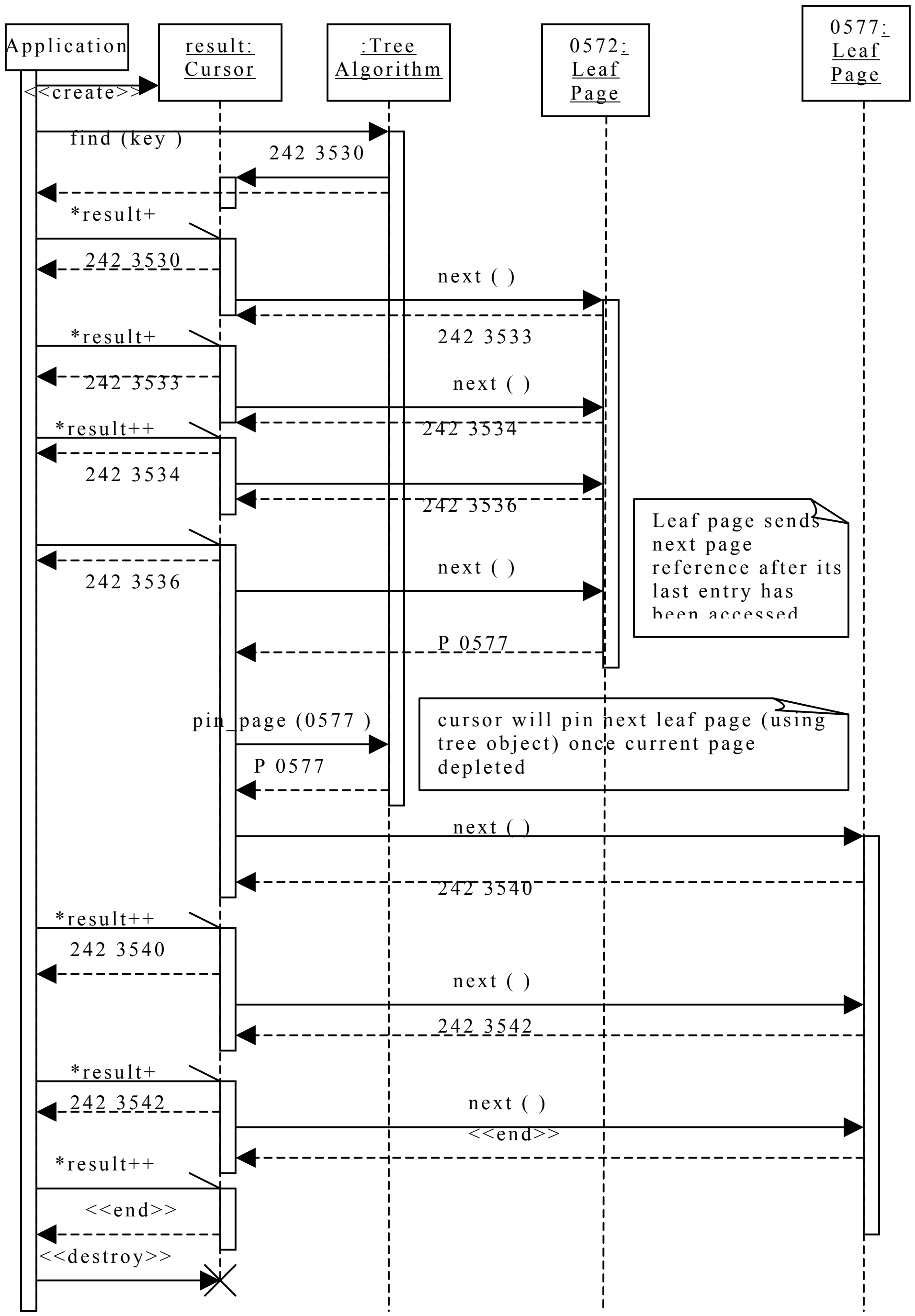


Figure 6.6: Cursor Iteration through Result

6.3.4 The Activity Diagram for Insertion

After an insertion, the page could become full. In this case the index will need to split it in two. Figure 6.7 shows the activity diagram for insertion.

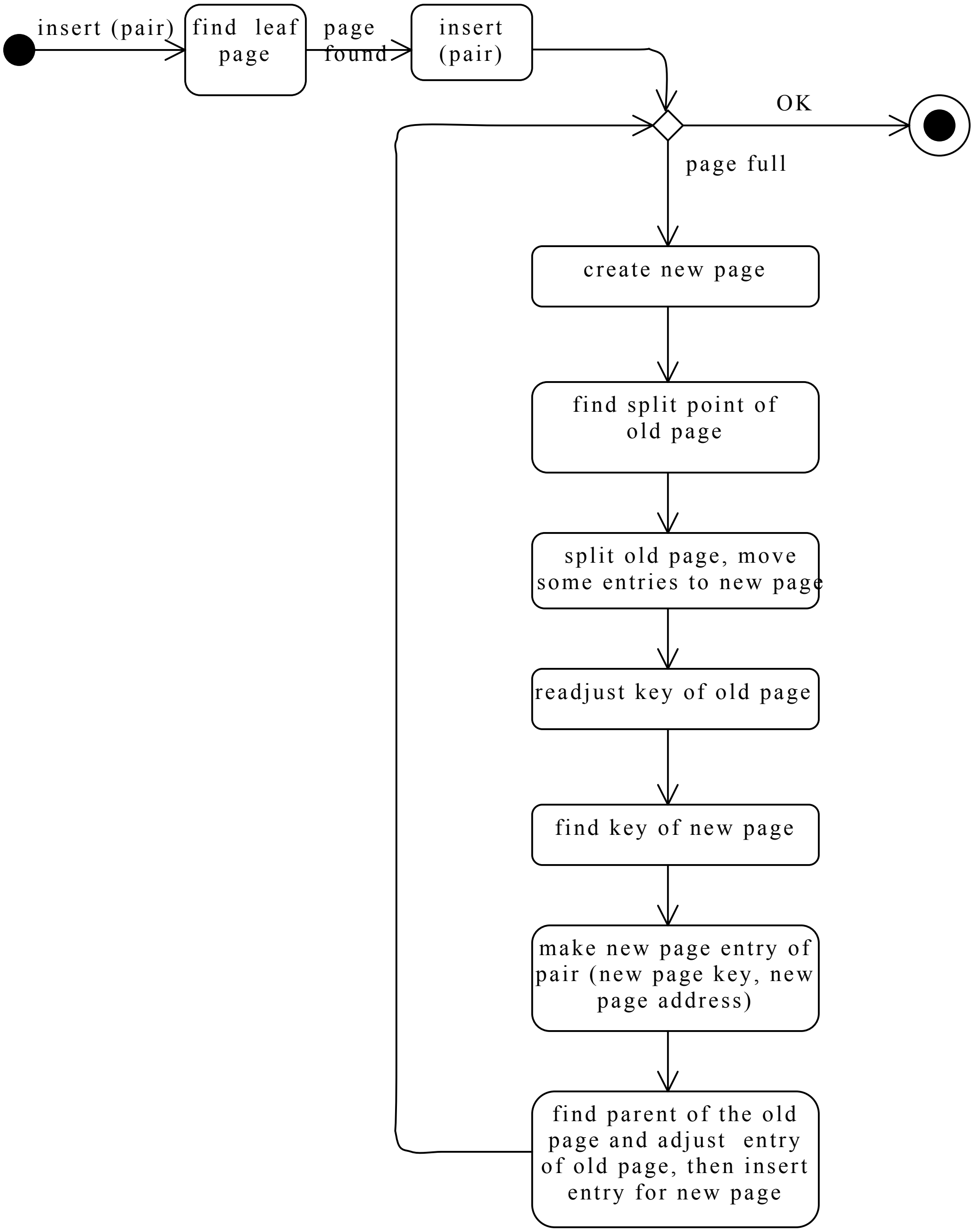
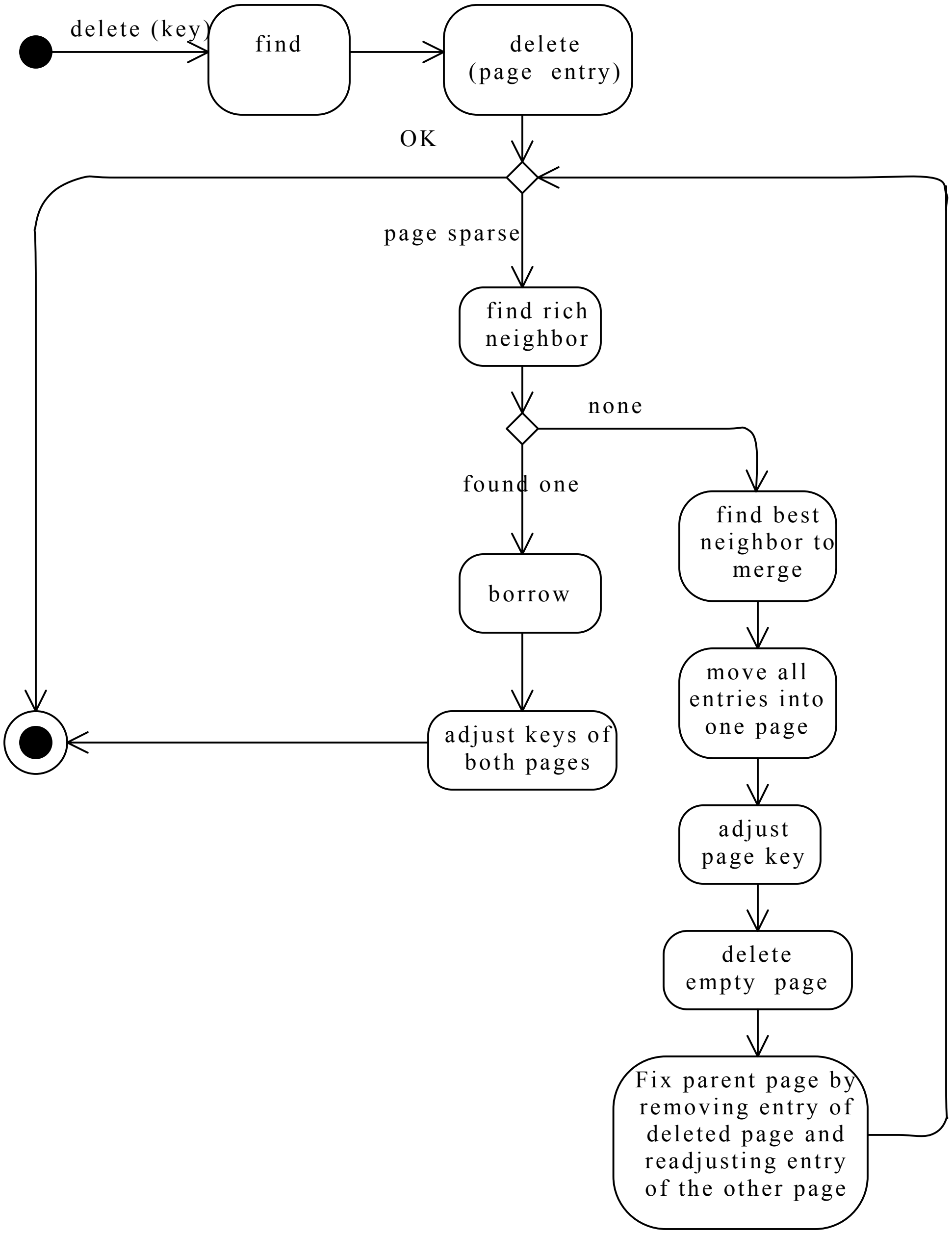

insert (pair)
find leaf page
page found
insert (pair)
OK
page full
create new page
find split point of old page
split old page, move some entries to new page
readjust key of old page
find key of new page
make new page entry of pair (new page key, new page address)
find parent of the old page and adjust entry of old page, then insert entry for new page

Figure 6.7: Insertion Activity Diagram

6.3.5 The Activity Diagram for Deletion

After a deletion, the page could become sparse. In this case the index will need to borrow from or merge with a neighbor page. Figure 6.8 shows the activity diagram for deletion.



delete (key)
find
delete (page entry)
OK
page sparse
find rich neighbor
none
found one
borrow
find best neighbor to merge
adjust keys of both pages
move all entries into one page
adjust page key
delete empty page
Fix parent page by removing entry of deleted page and readjusting entry of the other page

Figure 6.8: Deletion Activity Diagram

Chapter 7 Design for General Domain

The linearly ordered domain is a common application in database, but it does not work for all applications as it poses some constraints and assumptions on the properties of data to be used as a database. We need to isolate the characteristics that are specific to the linearly ordered domain, but not necessarily applicable to the general domain, then we generalize them to define the general characteristics that apply to the general domain. This would mean easing up some of the constraints required on data, to make the system workable with general data formats.

7.1 The Design

The main difference we need to address in general domain index is that each page searched may have more that one result. For example, in the R-tree index, as shown earlier, searching a key might give more than one matching entry. Each of these entries will lead the index algorithm to descend to a different child. So one page leads to descending to multiple children, each of which may lead to descending to yet more children. In the end the index search algorithm will arrive at more than one leaf page, probably tens or hundreds of them as shown in figure 7.1. All of these leaves may have some matching data for the application. The problem is that they are not stored in a sequential order anymore, so they cannot be accessed by just diving to the first matching data entry and then sequentially reading the rest of the entries. We cannot even define first match or sequential in an unambiguous way anymore. The design will include two main modifications to be able to accommodate a general domain database:

1-When searching a page, the searching algorithm should not terminate as soon as the first match was found. Since there might well be other matches in the page, the algorithm should check all the entries in a page and return all the matching ones.

2-When traversing the index tree, the traversal process should have the ability to go through different paths downward. This cannot be done simultaneously, so the index will use some temporary container to store all the candidate paths and then access them one at a time. A stack will be used to push all the children references returned by a page, and they will be popped one at a time for access.

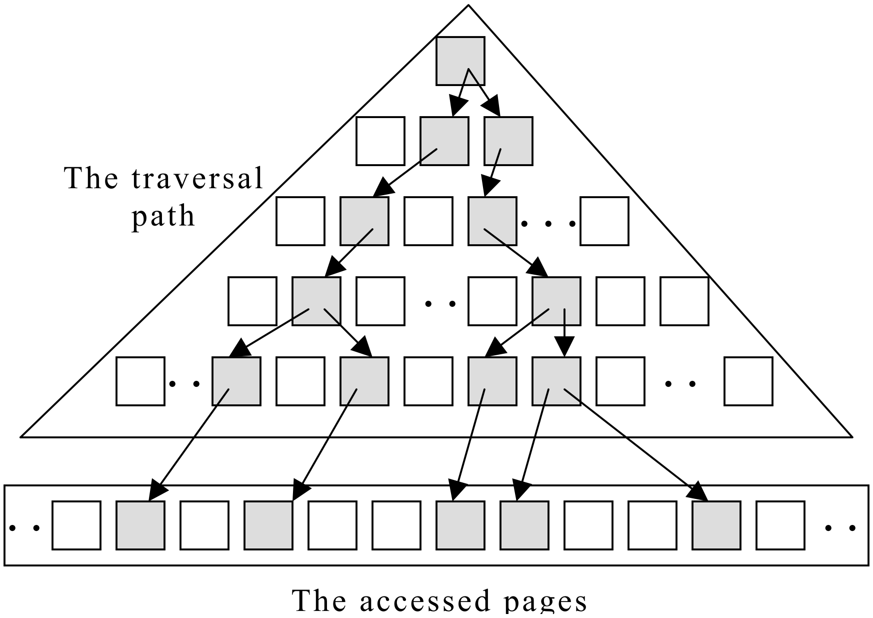


Figure 7.1: The Index Traversal for General Domain

7.2 The System Layout

Figure 7.2 shows a general view of the system components connected together. The index container will include an internal helper container, typically a stack. Note that the stack is not exactly a container, but just an adaptor that connects to an implicit container, masking its unneeded operations and offering only stack operations like push ( ) and pop ( ). The index, in this layout, cannot iterate through the stack by directly using its implicit container iterator. It can only pop the contents sequentially by sending a pop ( ) request to the stack adaptor.

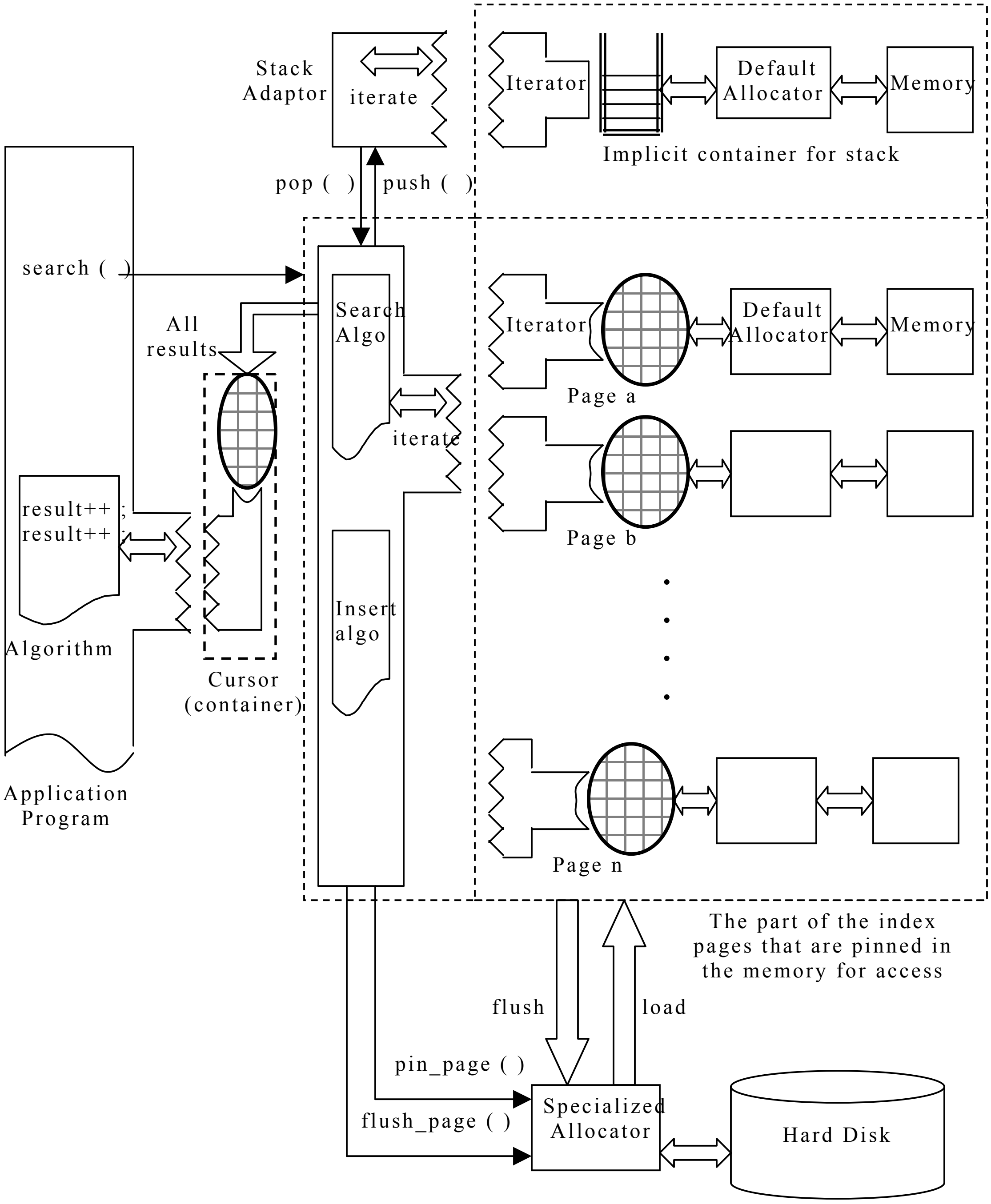


Figure 7.2: System Layout for General Domain Index

7.3 System Flexibility; Using Stack or Queue

Note that the leaf pages are retrieved in a reversed order due to the stack returning the last element first (Last In First Out, LIFO). This makes no impact on this batch search because in the end all matching entries will be in the cursor. If an application requires accessing the leaf pages in a forward order, the system allows for an easy adjustment by replacing the helper container stack with another helper container, Queue (STL container deque) without affecting any of the other system components. Since queue has a first-in-first-out policy (FIFO), this will give the needed result of accessing the leaf pages in a forward order. All the analysis, and requirements for using the stack use are applicable when using the queue.

7.3.1 The Effect of Stack and Queue on Index Behavior

Comparing the behavior of stack and queue and following the pattern of visiting the index pages shows some index behavior differences. The queue allows the index to access the leaf pages in a forward order, while the stack allows accessing them in a reversed order.

Besides, after completing a search in any page, the stack favors the depth direction while the queue favors the breadth direction (if possible). This makes the stack-based search capable of reaching the first leaf page faster (before exhaustively searching each level), while the queue-based search visits all pages in each level before moving to the next level, so after 50% of the search time has elapsed, a stack-based search would have accessed more leaf pages than a queue-based search. In the end, however, they will both have accessed the same pages.

Both these search policies have one thing in common; they both completed the search in every page before going to the next page. They then acted differently when deciding on the next page, going depth or breadth. A third search policy, depth first will follow a different behavior. The search in each page is not even completed, but rather interrupted after finding the first matching child. This behavior allows reaching the first leaf page even faster than the other two. Table 7.1 shows a brief comparison between the three policies. The Depth first policy is discussed in more details at the end of this chapter.

	Page search	Next page level	Index Traversal
Queue-based	Complete searching every page, visit each page once.	Complete searching all pages in same level.	Pure breadth-first policy.
Stack-based	Complete search in every page, visit each page once.	Quit other pages in same level, go to next level, and come back later to other pages of same level.	Mix of breadth- and depth-first policies.
Stack-based depth-first	Quit page after finding first child, go to child, and come back later to other children of same page.	Quit other pages in same level, go to next level, and come back later to same page or other pages of same level.	Pure depth-first policy.

Table 7.1: Comparison between Index Traversal Policies

7.4 The Class Diagrams

The nature of our framework allows for strong cohesion of classes and weak coupling. This is advantageous here as it reduces the modifications necessary to change from one design to another to as few classes as possible. The first modification, showing the advantage of strong cohesion, will be restricted to the page algorithm or the tree algorithm without affecting other system components. The second modification, showing the advantage of weak coupling, will be done by adding a new component to the design, a stack, and slightly modifying the index tree algorithm. We will study both modifications in more details. An object diagram of the new design is shown in the figure 7.3. It has a stack as a new component owned and used by the index algorithm to remember the multiple paths in traversing the tree.

The application always owns the result cursor. It uses the index algorithm to ask for the available services like search, insert, delete. The index algorithm will deposit all the results (in case of search for example) into the cursor. The cursor here does not have to access the page iterator or the index iterator to get more results, as they are all deposited into it during the search.

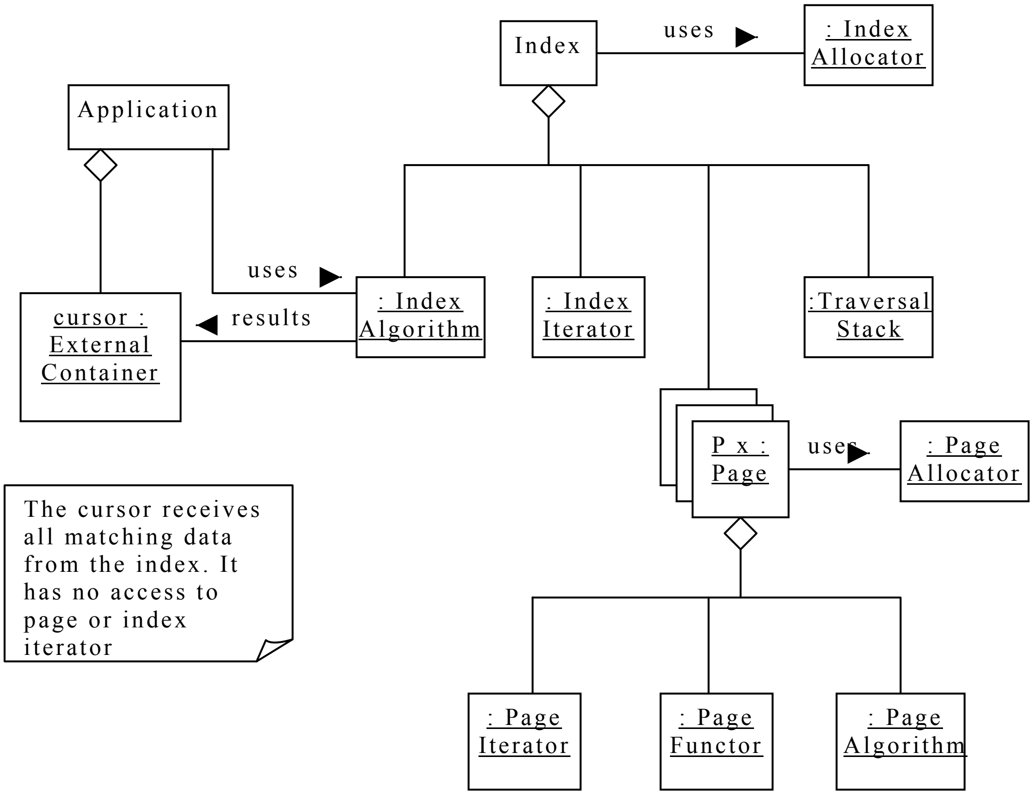


Figure 7.3: Object Diagram for Using Internal Query

7.5 The Behavior of the System

Figure 7.4 shows the sequence of interacting with the traversal stack using an internal query. The query is built-in inside the page container as an internal functor. The index will be the main object controlling the interaction here. It will create a page algorithm and connect it to the root page through an iterator. It will also create a traversal stack object.

Next it will start iterating through the page by sending a search command. The page algorithm will search for the first match, send it to the index and stop at the next position after the first match. Assuming that the page accessed is an internal node page having other page references as its data entries, the index, receiving the matching data in the form of page references, will send it to the stack for later use, (see also section 3). Next the index will send another search command to the page iterator, which will start the search from its current position, right after the first matching data, and search for the next match, send it to the index, and so on. As the page algorithm reaches the end of page, it will send an end-of-page, EOP, signal to the index. The index will proceed by popping the next page reference from the stack and access that page and iterate through it.

Modifying the system to use an external query requires only slight changes. The modification will be limited to making the page algorithm ask the external query if entries were matching, instead of asking the internal functor. The index will not always send the received page entries matching the searched key to the stack. It will need to decide on whether to send them to the cursor or to the stack depending on their type. Index page references will go to the stack, while data page references will go to the cursor. The source of entries will decide their type. Entries from a leaf page will certainly be data references and must be sent to the cursor to be retrieved by the application, so index will test if the statement *page is leaf* was true then send entries to the cursor.

Entries from a node page will not, however, always be page references to be sent to the traversal stack. For example, a single-node index will have only one node page (the root), which is also a leaf page. This page will therefore have data references as its entries (there is no other pages to send to the traversal stack). Testing if this page was a node page will give true even that, being a leaf as well, it has entries that are actually data references and should be sent to the cursor, not to the stack. Therefore, the index must check if *page is a leaf* was false and not *page was a node* was true to avoid such a problem. This should be reflected in the flag of the page itself, by having the flag to test leaf_page or not. There will be no need for another flag like node-page.

The iteration can be left to the page to do without index interaction. The page algorithm will send data directly to its proper destination.

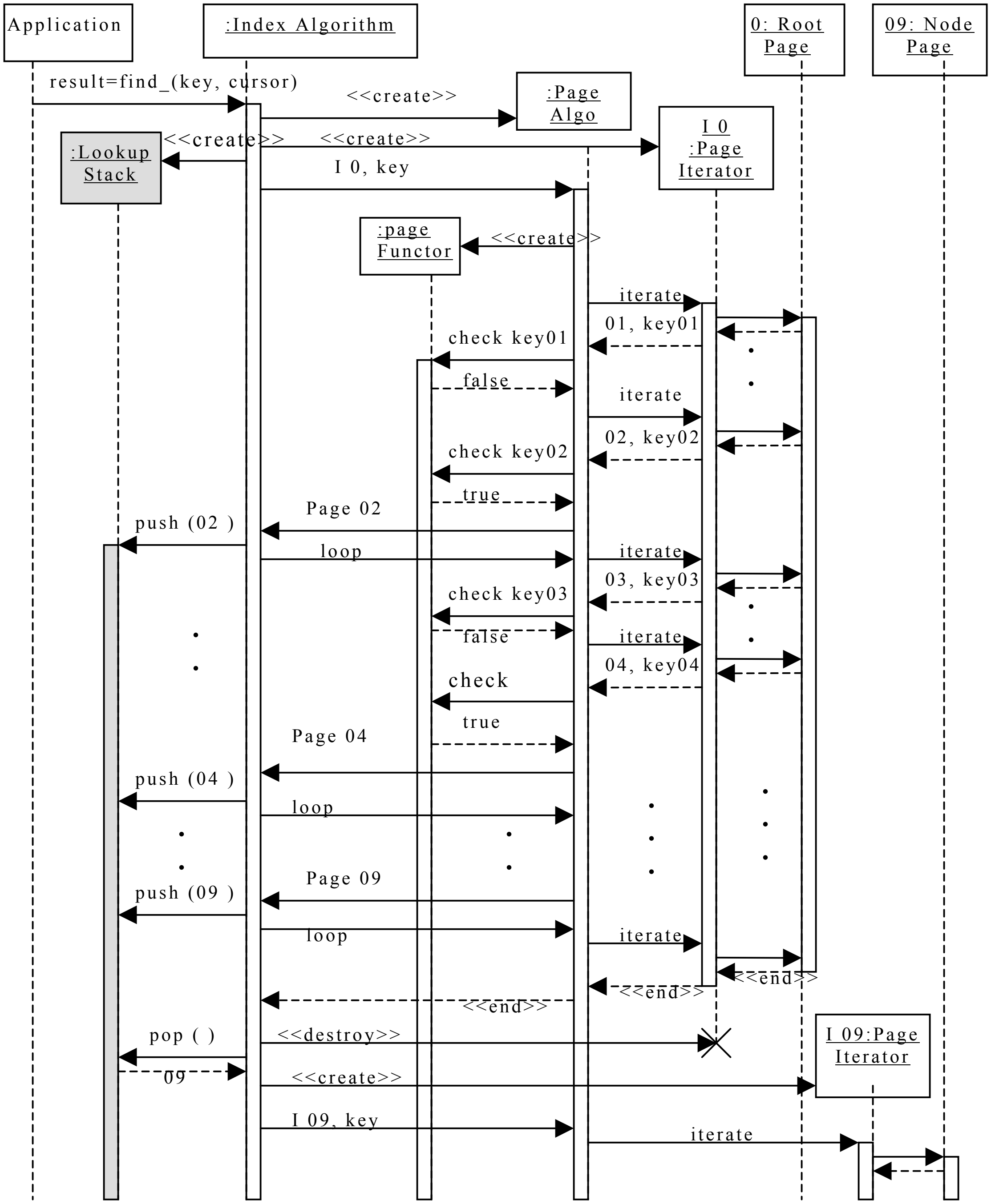


Figure 7.4: Sequence Diagram when Using Internal Query

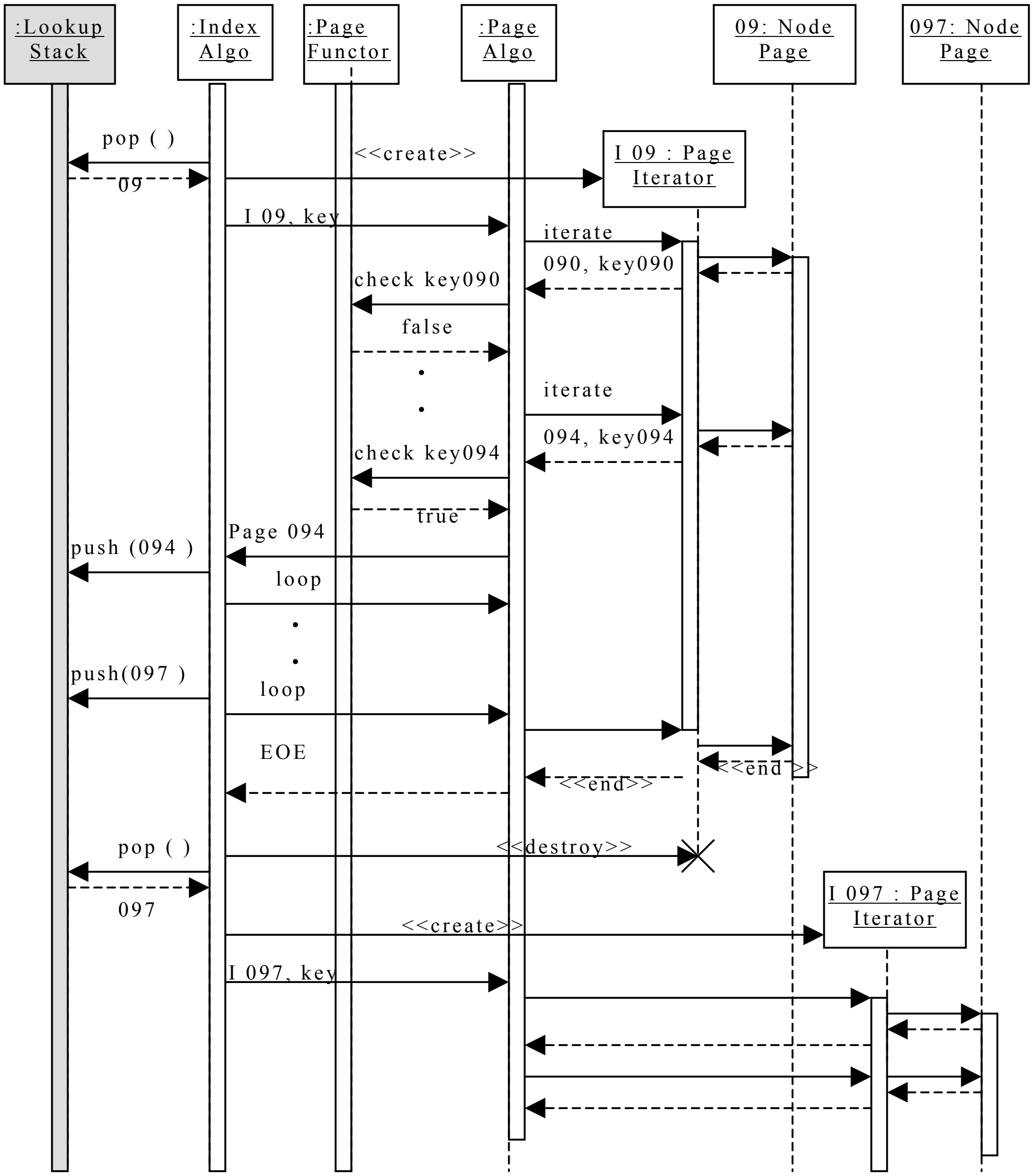


Fig7.4 (cont.): Sequence Diagram when Using Internal Query

Chapter 8 Similarity Search Applications

So far we have seen the analysis and design for two different domains: the linearly ordered domain, and the general domain (with both breadth first and depth first access methods). These domains have one common search concept; the equality search. As we searched for a key in different index pages, we compared the keys in a page with the key we are searching. Each comparison was done using an internal- or an external functor (query), and always gave one of two outcomes, matching or not matching. We then did our next move either downward (in depth-first general domain, and in linearly ordered domain) or laterally (in breadth-first general domain).

All these methods depend on choosing one of the two outcomes of the comparison object. No other outcomes of the comparison were allowed.

In some database applications, this is not always the case. Some times the comparison of two objects does not just give a yes/no result, but rather how close the two objects are; or in exact terminology how similar the two objects are. Even if the two objects do not exactly match, we are interested in them if they were similar, and not interested if they were not similar.

8.1 The Analysis

The similarity comparison would search for common features between two different objects, depending on the application. As explained in 3.6, the search result could produce a number to represent the distance between two objects (with 0 being exactly the same) or a percentage value showing how similar the two objects are (with 100% being exactly the same). A higher distance value or a lower percentage value would mean less similarities between the two objects.

Translating the percentage value into more similar or less similar is also application dependent. A 90 % similarity could be considered very similar in one application, while considered not similar enough in another. This introduces the concept of threshold value for similarity, wherein we may define a threshold value for similarity outcomes to be considered in the result set of similar objects or not.

Changing this rather vague concept of the percentage values outcome into a more solid one like similar or not will make data easier to handle and study. The application may set the threshold for the percentage outcome depending on the domain. All values below this threshold value will be considered as not matching and ignored, while values above it will be considered as matching and taken into account (another approach is to consider the n-most similar objects without using a threshold value. This, again, depends on the application, but does not affect our analysis here).

This helps in eliminating a certain amount of data that is not similar enough (to the data we are searching) to be considered.

As we search we will have on hand more than one object to consider as similar to the one we are looking for. We may want to visit some or all of them, so it would make sense to visit the more similar ones first; the ones with higher percentage similarity. In other words out interest in visiting these objects will be proportional to their value of similarity. The best way to do that is by ordering them in a queue according to their similarity value, and start visiting them from higher to lower.

With the index pages arranged into a hierarchy over different levels, it will be slightly more complex. Starting from the top, we go downward visiting all similar pages (with similarity percentage value above threshold value) from higher to lower percentage. As we visit a page, we exhaustively search all its entries and the search will return certain page entries (of key and page reference) accompanied with how similar it is to our key. All these entries are then added to a priority queue with their percentage similarity value as its priority. The priority queue will take the responsibility of inserting entries partially or totally ordered such that we can always access the entry with the highest priority next.

As we finish searching a page, we look for the next page to visit by reading the priority queue. We will get the page with the highest similarity value, and search it, inserting further entries with their similarity values into the queue, read the next page, search it and so on.

This search is neither a depth first nor a breadth first. As explained in chapter 3, the direction of our next move is not simply one step lateral or one-step vertical. It solely depends on the next value popped from the queue. This could result in any kind of movements. The next page could be anywhere in the index, two steps up, five pages to the right, … etc. On the index it will look like jumping in all directions, possibly multiple levels at a time, just favoring the highest similarity value as the sole motive to our next move (a demonstration with a numerical example of this movement is given in section 8.2). It is clear that, unlike depth-first general domain, and linearly ordered domain, and similar to the breadth-first general domain, we are doing an exhaustive search on each page. The reason is that we are not in a hurry to descend to the first match found, but rather want to find the best match in each page before leaving it. This implies that we delete the page from the priority queue after we visited it.

On the index level, we pick up the best page to visit, and search for the best matching entries in it (more pages). We are doing what can be called the best of the best search. To guarantee this criterion, we do not just take the best match in a page and visit it directly; we rather put all matching pages from every search, regardless of their level, in the queue with the matching pages from previous search. We then select the best global match over the whole index again and again, and search for the best matches in it, add them to the queue, and so on.

As we hit a data item, we send it to the cursor. This item should be the most similar item to the one the application is looking for. We then wait for the application to access it.

8.2 A Numerical Example

Figure 8.1 shows with a numerical example how the similarity search proceeds to find the matching data according to how close they are to data we are searching. For simplicity, we will assume that each leaf page contains directly the data we are looking for (an Image, for example). If the leaf pages contained keys with similarity percentage values, we can just add one more level to the example, and use the same technique. It would only be unwieldy large. This example here will show the necessary details only. The pages of interest are shown with their numbers, using the same numbering convention as before. The pages, which are of no interest to us, are shown only as smaller rectangles to work as a placeholder for those pages. They are considered anonymous pages and their numbers are not shown, for obvious space reasons

We note that as we descend on the branches of a subtree in the figure, we might get lower priority values than the higher level nodes of the same subtree. For example, node 07 had priority 89%, but as we descend on it, its children 075, and 077 have lower priorities of 75 % and 85%, then further child 0774 has only 74%. This is not an unusual feature of similarity search. The keys at higher levels of the index, covering larger subtrees, will be required to summarize the features of all the pages in their subtrees. This might lead to ignoring some details, or generalizing them, making them coarse-grained. Testing one of those coarse-grained keys might give a less accurate high similarity value. As we take on this key and descend in its subtree, we get more fine-grained keys as more details are revealed about the now smaller subtree. Those new details might result in a lower, but more accurate similarity values than first appeared on the parent page key. This could lead to the unwanted side effect of accessing less similar node pages before accessing more similar ones, being misled by this feature. Using the priority queue helps recover and avoid this side effects in the leaf level as we put all pages in one global priority queue, with their similarity to the searched key as the priority metric. As the pages in a branch get less similar than their parent page once did, they are automatically pushed behind older pages already in the queue with now more similarity to our searched key, resulting in abandoning the less similar branch, and moving to the more similar one. In figure 8.1, instead of going to 0774, we abandoned the whole subtree of node 07, and moved to 03. This will always guarantee that we will visit the page with highest priority, and reduce the possibility of going on a less lucrative path. In this example, the arrows show that we have information about that page with the percentage of their similarity to the key we search. The dotted ones mean that the page is not yet visited. The solid arrows show that we used this path, and already visited the page at its end. The visited pages are shown in a light gray tone.

8.3 The Order of Reading Leaf Pages

In both linearly ordered and general domain, we access the leaf pages in some forward or reverse physical order depending on the access method and the type of helper container used. The breadth first policy using a stack (we called it a mix of breadth- and depth first policies) always read the matching leaves from the last to the first. The breadth first policy using a queue (we called it pure breadth-first policy) always read the matching leaves from the first to the last. The depth first policy starts by reading the first and precedes to the last match. All these policies are position-oriented in that the order of accessing the pages depends on their position, regardless of their contents.

In similarity search, there is no predetermined order in accessing the matching pages. This policy is contents-oriented in determining the sequence of pages to access, regardless of their position. We access the most similar page, then the second most similar and so on. Changing the similarity value of the pages to the one we search will therefore be the factor leading to changing the order we access them.

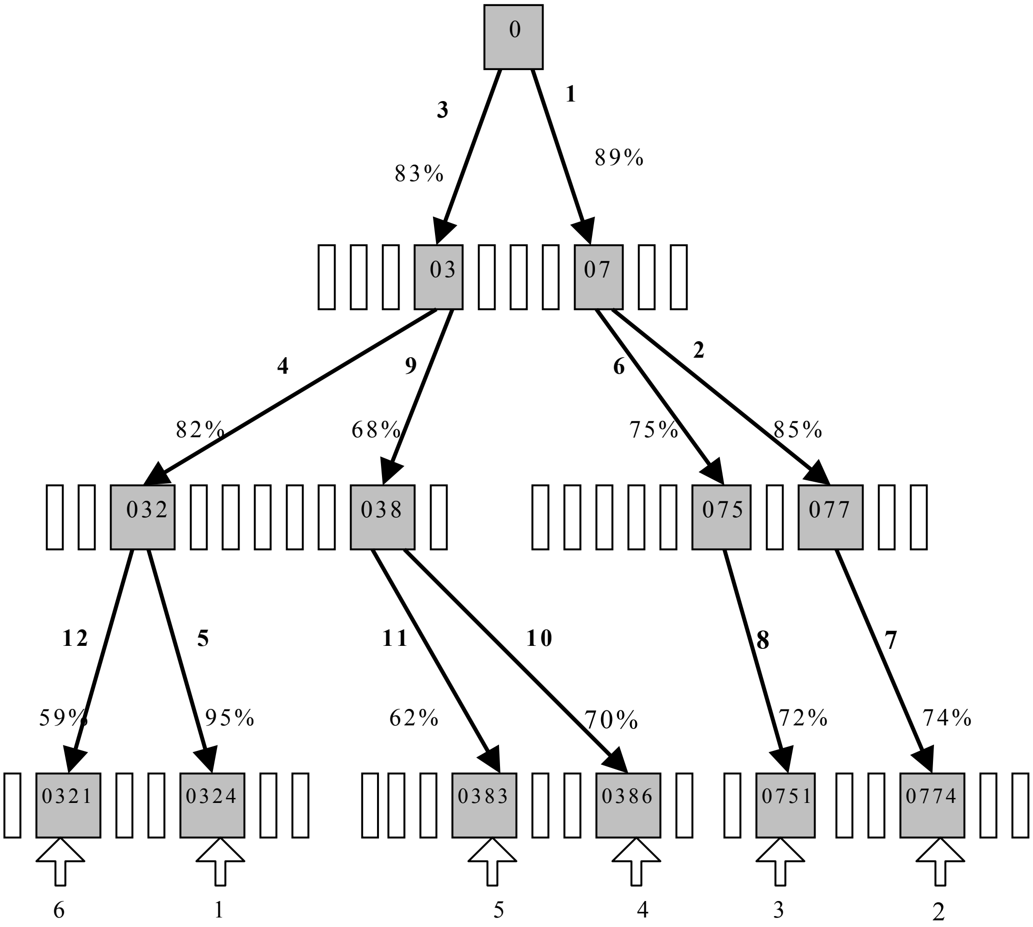


Figure 8.1: Search Sequence in Similarity Search

Chapter 9 Conclusion

In this thesis we describe how to build a generalized tree index from independent building blocks that are coherent and decoupled.

A database index is a data structure with a set of functionality to search and update a database. The index structure follows certain algorithms for searching, insertion, and deletion of data as well as internal index maintenance algorithms that guarantee data integrity.

The C++ STL library, a part of the standard library, introduces a concept for breaking a container data structure into a set of algorithm – iterator – container modules. We expand this concept to build a complete modular index system. In the design, we break the index data structure with functionality into container of pages with each page built as container of entries where each entry is a pair of <key, reference> .

The index module, being a container of pages, is further decomposed into algorithm – iterator – container modules. Similarly the page module, being a container of entries is further decomposed into algorithm – iterator – container modules. The entries are then stored in each page as pairs of key, reference. Other modules; allocators, further separate the process of primary and secondary storage management from the other modules. We also use some helper modules; like result cursor container, search container (stack / queue), and functors to complement the necessary system parts. We use adaptor modules like functor adaptors, container adaptors, and iterator adaptors to adjust the slight irregularities between some incompatible modules. In the end we add the data and the data reference modules to complete the system. This allows us to construct a complete index system from modules.

In order to adapt the system to different keys / data types, different queries, different access methods, and different storage media, we need to locate and modify (or simply replace) some modules in the system. We provide some examples to show how to adopt the system to linearly ordered domain applications by providing an analysis of this domain. This analysis identifies the modules in the system that needed modifications. We reflect these modifications on the system design and eventually on the interface. Similarly we adapt the system to general domain applications by first providing an analysis of this domain and exposing the main changes found in this domain. Again, this leads us to identifying the modules to modify in the design along with a new module to add (either a stack, or a queue) to achieve a new system in general domain (with both depth-first and breadth first access methods). Again these modifications are then reflected to the interface. Finally, we provide an analysis of the similarity-search domain and follow the same steps to identify and apply the changes needed in the affected modules.

Using a modular design for the index system has the advantage of making it easier to adapt the system to work in different database domains. The analysis of the domain determines the modules that need changes (or replacement), and the kind of changes (or modules) required. This means that the other modules need not be changed. This greatly reduces the efforts needed to modify the system. The complexity of modification is also reduced since the developer does not need to know about the details of all modules, but only of those modules to be changed along with an overview of the system. The adoption of STL approach adds great advantage of having a wealth of off-the-shelf standard modules that can be simply used to replace system modules in the process of modifying the system. This promotes code reuse and thus increase readability, user-friendliness and reduces time and money overheads incurred during the application development process.

9.1 Future Work

Our ultimate goal is to build an index system to handle any type of data, key, query or access method in a real database application. One important issue in database is concurrency and locking mechanisms to allow for multiple access to the same piece of information while guaranteeing data integrity. This issue needs to be addressed according to the classes suggested in the design. Another important work that still needs to be done is the construction of concrete classes as shown in the design to obtain a working index.

The design started with tree indexes and we see that it has a potential to handle other index types such as inverted lists, hash tables, and parallel search. These types of indexes deserve further investigations, and we hope to see the design extended to cover them as well.

Bibliography

[AKKS99] M. Ankerst, G. Kastenmüller, H.-P. Kriegel, T. Seidl. **3D Shape Histograms for Similarity Search and Classification in Spatial Databases**. Proceedings of the 6th International Symposium on Large Spatial Databases (SSD'99), Hong Kong, China, in: Lecture Notes in Computer Science, Vol. 1651, Springer, 1999, pp. 207-226.

[AKKS99-2] M. Ankerst, G. Kastenmüller, H.-P. Kriegel, T. Seidl. **Nearest Neighbor Classification in 3D Protein Databases**. Proceedings of the 7th International Conference on Intelligent Systems for Molecular Biology (ISMB'99), Heidelberg, Germany, AAAI Press, 1999, pp. 34-43.

[Aok98] P. M. Aoki. **Generalizing "Search" in Generalized Search Trees**. Proceedings of the 14th IEEE International Conference on Data Engineering, Orlando, FL, Feb. 1998, pp. 380-389.

[Aus99] Matthew H. Austern. **Generic Programming and the STL**. Addison-Wesley, 1999.

[BBB+97] S. Berchtold, C. Böhm, B. Braunmüller, D.A. Keim, H.-P. Kriegel. **Fast Parallel Similarity Search in Multimedia Databases**. Proceedings of the ACM SIGMOD International Conference on Management of Data AZ, 1997, Best Paper Award, pp. 1-12.

[BBBK00] C. Böhm, B. Braunmüller, M. Breunig, H.-P. Kriegel. **High Performance Clustering Based on the Similarity Join**. Proceedings of the 9th International Conference on Information and Knowledge Management (CIKM 2000), Washington DC, 2000, pp. 298-313.

[BBKM00] C. Böhm, S. Berchtold, H.-P.Kriegel, U. Michel. **Multidimensional Index Structures in Relational Databases**. in: Journal for Intelligent Information Systems, Vol. 15, 2000, pp. 51-70.

[BCC+02] 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**, to appear in Domain Oriented Systems Development: Perspectives and Practices, Kiyoshi Itoh, Satoshi Kumagai (eds), Gordon Breach Science Publishers, UK, 2002.

[Bch94] Grady Booch. **Object-Oriented Analysis and Design**. The Benjamin/Cummings Publishing Company 1994.

[BD99] G. Butler and P. Dénommée. **Documenting Frameworks**, in Building Application Frameworks: Object-Oriented Foundations of Framework Design. M. Fayad, D. Schmidt, R. Johnson (eds). John Wiley and Sons, New York, September 1999, pp. 495-504.

[BEK+98] S. Berchtold, B. Ertl, D. A. Keim, H.-P.Kriegel, T. Seidl. **Fast Nearest Neighbor Search in High-dimensional Space**. Proceedings of the 14th International Conference on Data Engineering (ICDE'98), Orlando, FL, 1998, pp. 209-218.

[BKM00] G. Butler, R.K. Keller, H. Mili. **A Framework for Framework Documentation**. ACM Computing Surveys 32,1 (March 2000) electronic symposium.

[BKS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, Bernhard Seeger. **The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles**. Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990. ACM Press 1990, pp. 322-331.

[BRI99] Grady Booch, James Rumbaugh, Ivar Jacobson. **The Unified Modeling Language User Manual**. Addison-Wesley 1999.

[Bry00] Ulrich Breymann. **Designing Components with the C++ STL: A New Approach to Programming**. Addison-Wesley 2000.

[But99] Greg Butler. **Developing Frameworks by Aligning Requirements, Design, and Code**. Proceedings of the 9th Workshop on Software Reuse (WISR-9), Austin, Texas, January 1999, 5 pages.

[BX01] Greg Butler, Lugang Xu. **Cascaded Refactoring for Framework Evolution**. Proceedings of the 2001 Symposium on Software Reusability. ACM Press, 2001, pp. 51-57.

[Dsi90] Bipin C. Desai. **An Introduction to Database Systems**. St. Paul West Publishing 1990.

[EN94] Ramez Elmasri, Shamkant B. Nanathe. **Fundamentals of Database Systems**, second edition. Benjamin/Cummings Publishing Company 1994.

[FBF+94] C. Faloutsos, R. Barber, M. Flickner, J. Hanfner, W. Niblack, D. Petkovic, W. Equitz. **Efficient and Effective Querying by Image Content**. Journal of Intelligent Information Systems, Vol 3 (1994) pp.231-262.

[FZ92] Michael J. Folk, Bill Zeollick. **File Structures**, Second Edition. Addison-Wesley 1992.

[Gdm93] John Goodman. **Memory Management for All of Us**. Sams Publishing 1993.

[GG98] V. Gaede, O. Guenter. **Multidimensional Access Methods**. ACM Computing Surveys, Vol. 30, No. 2, 1998. pp. 170-231.

[GHJV95] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. **Design Patterns**. Addison-Wesley 1995.

[Gut84] Antonin Guttman: R-Trees. **A Dynamic Index Structure for Spatial Searching**. Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, June 18-21, 1984. ACM Press 1984, pp. 47-57

[HCB+99] Joseph M. Hellerstein, Chad Carson, Serge J. Belongie, Megan C. Thomas and Jitendra Malik. **Blobworld: A System for Region-Based Image Indexing and Retrieval**. Proceedings of the 3rd International Conference on Visual Information Systems, VISUAL 99. Amesterdam, Netherlands, June 1999, published as lecture notes in computer science, Springer-Verlag, vol. 1614, pp. 509-516.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton and Avi Pfeffer. **Generalized Search Trees for Database Systems**. Proceedings of the 21st International Conference on Very Large Data Bases, Zurich, September, 1995.

[HSE+95] J. Hafner, H. S. Sawhney, W. Equitz, M. Flickner, W. Niblack. **Efficient Color Histogram Indexing for Quadratic Form Distance Functions**. IEEE Trans. On Pattern Analysis and Machine Intelligence, Vol 17, No. 7. IEEE Press (1995) 729-736.

[JD88] A. K. Jain, R.C. Dubes. **Algorithms for Clustering Data**. Prentice-Hall 1988.

[KKS98] G. Kastenmüller, H.-P.Kriegel, T. Seidl. **Similarity Search in 3D Protein Databases**, Proceedings of the German Conference on Bioinformatics (GCB'98). Köln, 1998.

[Knt73] Donald E. Knuth. **The Art Of Computer Programming**, Volume2 / Seminumerical Algorithms. Addison-Wesley 1973.

[Knt73-2] Donald E. Knuth. **The Art Of Computer Programming**, Volume3 / Sorting and Searching. Addison-Wesley 1973.

[KS97] Norio Katayama, Shin'ichi Satoh. **The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries**. Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (May 1997) pp. 369-380.

[LH96] Jesse Liberty, J. Mark Hord. **ANSI C++**. Sams Publishing 1996.

[Lip96] Stanley B. Lippman. **Inside The C++ Object Models**. Addison-Wesley Longman, 1996.

[LL98] Stanley B. Lippman, Josee Lajoie. **C++ Primer**, Third Edition. Addison-wesley 1998.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Loensen. **Object-Oriented Modeling and Design**. Prentice Hall 1991.

[Res99] Steven P. Reiss. **A Practical Introduction to Software Design with C++**. John Weily and Sons. 1999.

[Rbn81] John T. Robinson. **The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indexes**. Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, April 29 - May 1, 1981. ACM Press 1981 , pp. 10-18

[Rob00] Robert Robson. **Using the STL: the C++ Standard Template Library**, second edition. Springer-Verlag 2000.

[SK01] T. Seidl, H.-P. Kriegel. **Adaptable Similarity Search in Large Image Databases**, in: R. Veltkamp, H. Burkhardt, H.-P.Kriegel (eds.): State-of-the Art in Content-Based Image and Video Retrieval, Kluwer Publishers, 2001, pp. 297-317.

[SK95] T. Seidl, H.-P. Kriegel. **Solvent Accessible Surface Representation in a Database System for Protein Docking**. Proceedings of the 3rd International Conference On Intelligent Systems for Molecular Biology (ISMB), 1995, pp. 350-358.

[SK97] T. Seidl, H.-P. Kriegel. **Efficient User-Adaptable Similarity Search in Large Multimedia Databases**, Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97), Athens, Greece, 1997, pp. 506-515.

[SKH99] Mehul Shah, Marcel Kornacker, Joseph M. Hellerstein. **amdb: A Visual Access Method Development Tool**. User Interfaces for Data Intensive Systems (UIDIS). Edinburgh, 1999.

[Slz88] Betty J. Salzberg. **File Structures**. Printice-Hall 1988.

[Som95] Ian Sommerville. **Software Engineering**, Fifth Edition. Addison-Wesley 1995.

[SRF87] Timos K. Sellis, Nick Roussopoulos, Christos Faloutsos. **The R+-Tree: A Dynamic Index for Multi-Dimensional Objects**. Proceedings of the 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England, pp. 507-518.

[Str00] Bjarne Stroustrup. **The C++ Programming Language**, special edition. Addison-Wesley 2000.

[Str94] Bjarne Stroustrup. **The Design and Evolution of C++**. Addison-Wesley 1994.

[WJ96] D. A. White and R. Jain. **Similarity Indexing with the SS-tree**. Proceedings of the 12th International Conference on Data Engineering, New Orleans, USA (Feb. 1996), pp. 516-523.

Appendix A Flow of Events

Expert developer “designs a database system”
Pre-condition: None
Main flow: The developer studies the different functional and non-functional requirements of the system and puts a design to satisfy them.
Post-condition: A design for a database and index system, including the primary and secondary indexes.

Expert developer “Implements a database system”
Pre-condition: An integrated design of the system.
Main flow: The developer implements the different components of the system and test it on a platform.
Post-condition: A core code for a database system including the indexes to help access them.

Database Developer “provide a query ”
Pre-conditions: An index core code must be present.
Main flow: Database Developer writes the code for a new index query class to be used by an existing system.
Post-condition: A customized index code that supports a new query.

Database Developer “provides a data type ”
Pre-conditions: An index core code must be present.
Main flow: Database Developer writes the code for a new index data type class to be used by a new system. This data type will give a new index capable of dealing with database that uses the new data type.
Post-condition: A customized index code that supports an application with a new data type.

Database Developer “defines new access method”
Pre-conditions: An index core code must be present.
Main flow: Database Developer writes the code for building a new index
required by a new system. The new index structure will be capable of using a new access method concept to traverse the tree.
Post-condition: A customized new index code that supports a new access method concept.

Database Developer “sets index parameters ”
Pre-conditions: An index core code must be present, and any customizable parts (extensions) must be implemented.
Main flow: The extension programmer sets the index parameters (like the page fill factor), the tree order to adjust the index implementation to the particular application.
Post-condition: A customized index code that has specific values for its parameters suitable for the application and the system variables.

Database Administrator “defines scheme”
Pre-conditions: A suitable design for data tables and a Data Definition Language (DDL).
Main flow: The DBA uses the DDL tools to define the format of the tables used to store the data. The definition will include table name, the name and type of each field and the constraints applied to them.
Post-condition: A definition of the tables used in the data and the relations between them.

Database Administrator “builds data references”
Pre-conditions: Physical data files carrying the bulk of data to be used in the database must be available in the data tables.

Main flow: Build data groups / partitions (if necessary), and define references to their physical locations, as well as suitable keys describing those partitions.

Post-condition: Creation of data references that will be used by the index.

Database Administrator "build index"

Pre-condition: A customized index for a particular application must be available. Data scheme must be defined and Physical data must be loaded. Reference file comprising data keys and references to their physical locations must be available.

Main flow; using a bottom up technique: Create an empty index file, sort the reference file, then build the index using the bottom up technique.

Alternative flow; using the top down technique: Create an empty index file then build the index using top down technique.

Post-condition: Creation of an index to the data.

Database Administrator "dynamically fine-tune index"

Pre-conditions: An index file built on the physical data, and an index object that can successfully open an index file (same type)

Main flow: The DBA runs the system by using an index object to open an existing index file of the same type, then dynamically fine-tune the system by monitoring the performance while the index is being used and adjusting the system parameters (bucket/page size, fill factor etc.) to reach the optimal performance.

Post-condition: An index that is optimized for the applications using it.

Database application "uses index"

Precondition: A customized index suitable for this application and a cursor must be available. A previously created index object must exist, that can be opened by the application, connected to an index file built with the same extension type.

Main flow: Search data

The application will search for some data using a query. First it initializes an iterator (cursor) object to accept pairs of key, data reference as an output from the search operation. The search algorithm traverses the index using the query to select the candidate data references at leaf level that satisfy the given key. These will be returned to the cursor object, where the application can access them sequentially.

Alternative flow: Insert data

The application will insert some data in the form of key, data reference. The application will provide the key, data reference pair to be inserted. The index insertion algorithm will perform it in two steps: First it will use the key in the search algorithm to find a suitable location on the index leaf level for insertion. Second the insertion algorithm is used to insert the key, data reference pair in the resulting location. The necessary adjustments to the index are then performed.

Alternative flow: Delete data

The application will attempt to delete some data from the database using a key. The application provides a key for data to be deleted. This is done in two steps: First the delete algorithm will use the key to try and locate the data to be deleted using the search algorithm. Second if the index find the location of candidate data for deletion, the delete will be applied on data occupying that location. The necessary adjustments to the index are then performed.

Post-condition: The index is adjusted to reflect the current status of the data.

Appendix B Interfaces

Page public type members and data members.

page <Key, T, Compare, Allocator> :: **iterator** ;   //internal iterator class

page <Key, T, Compare, Allocator> :: **const_iterator** ;
//An iterator class that allow read-only access to the container elements

page <Key, T, Compare, Allocator> :: **reference** ;
//type of the reference to the **contents** of the container

page <Key, T, Compare, Allocator> :: **const_reference** ;

page <Key, T, Compare, Allocator> :: **key_type**;
//typedef, allow for user defined key types

page <Key, T, Compare, Allocator> :: **mapped_type**;   //typedef of the class T

page <Key, T, Compare, Allocator> :: **value_type**; //typedef, the pair <const Key, T>

page <Key, T, Compare, Allocator> :: **size_type**;

page <Key, T, Compare, Allocator> :: **difference_type**;

page <Key, T, Compare, Allocator> :: **allocator_type**;

page <Key, T, Compare, Allocator> :: **key_compare**;

page <Key, T, Compare, Allocator> :: **page_key**;

Page public member functions.

page <Key, T, Compare, Allocator> :: **page** (const Compare& comp = Compare ( ),
const Allocator& = Allocator ( ));
//constructs an empty page

template <class InputIterator>
page <Key, T, Compare, Allocator> :: **page** (InputIterator first,
InputIterator last,
const Compare& comp = Compare ( ),
const Allocator& = Allocator ( ));
//constructs a page and inserts the values in the range first to last.

page <Key, T, Compare> :: **page** (const page <Key, T, Compare, Allocator> & x);
//copy constructor

size_type page <Key, T, Compare, Allocator> :: **size ( )** const;   //returns current size

bool page <Key, T, Compare, Allocator> :: **is_leaf ( )** const;   //true if page is a leaf

bool page <Key, T, Compare, Allocator> :: **is_full ( )** const;//true if page is full

bool page <Key, T, Compare, Allocator> :: **is_sparse ( )** const;   //true if page is sparse

bool page <Key, T, Compare, Allocator> :: **is_dirty ( )** const;
//true if page contents have been modified.

size_type page <Key, T, Compare, Allocator> :: **max_size ( )** const;
//returns maximum size

iterator page <Key, T, Compare, Allocator> :: **parent ( )**;

const_iterator page <Key, T, Compare, Allocator> :: **parent ( )** const ;

iterator page <Key, T, Compare, Allocator> :: **begin ( )**;

const_iterator page <Key, T, Compare, Allocator> :: **begin ( )** const ;

iterator page <Key, T, Compare, Allocator> :: **end ( )**;

const_iterator page <Key, T, Compare, Allocator> :: **end ( )** const;

iterator page <Key, T, Compare, Allocator> :: **find** (const key_type& x);

const_iterator page <Key, T, Compare, Allocator> :: **find** (const key_type& x) const;
//find first object equal to x

iterator page <Key, T, Compare, Allocator> :: **find_if** (query& Query);

const iterator page <Key, T, Compare, Allocator> :: **find_if** (query& Query);
//this algorithm is using an external comparison query.

pair <iterator, bool> page <Key, T, Compare, Allocator> :: **insert**
(const value_type& x);
//inserts a value (the pair <const Key, T) in its proper place.

iterator page<Key, T, Compare, Allocator>:: **insert**
(iterator position, const value_type& x);
//inserts a value in its ordered place but start searching for the place from position.

void page <Key, T, Compare, Allocator> :: **erase** (iterator position);
//delete object at position

size_type page <Key, T, Compare, Allocator>::**erase** (const key_type& x);
//delete all occurrences of object x and returnthe number of objects deleted (zero or one
//for unique key

void page <Key, T, Compare, Allocator> :: **erase** (iterator first, iterator last);
// delete all objects in the range from first to last.

T& page <Key, T, Compare, Allocator> :: **operator [ ]** (const key_type & x);
//allow for directly indexing the objects by their keys

key_compare page <Key, T, Compare, Allocator> :: **key_comp** ( ) const;
// return the functor used in the page (for comparing keys)

iterator page <Key, T, Compare, Allocator> :: **find_split_point** ( );
//return an iterator to the best position to split a page if it was full.

key_type page <Key, T, Compare, Allocator> :: **find_page_key** ( );
//each page is able to extract the key of its contents

Index public type members and data members.

typedef page <Key, T, Compare, Allocator> :: key_type   **key_type**;
//the key type used in the index container is obtained from the page container

index < page, Allocator, Container> :: **iterator**;   //An iterator to the container

index < page, Allocator, Container> :: **const_iterator**;

index < page, Allocator, Container> :: **reference**;
//type of the reference to the contents of the container: a page reference

index < page, Allocator, Container> :: **const_reference**;

index < page, Allocator, Container> :: **data_type**;   // the class page

index < page, Allocator, Container> :: **allocator_type**;

const index < page, Allocator, Container> :: **page_fill_factor**;

Index public member functions.

reference index < page, Allocator, Container> :: **root_page( )**;

```
reference index < page, Allocator, Container> :: find (key_type& x, cursor &results);
//key_type is from page container.

const_ reference page < page, Allocator, Container> :: find
                                    (const key_type& x, cursor &results) const;

reference index < page, Allocator, Container> :: find_if (query &Query , cursor results);
// searching the index using an external query

const_reference index < page, Allocator, Container> :: find_if
                                                     (query &Query , cursor results);

reference index < page, Allocator, Container> :: insert
                                       (const key_type& x, const page :: mapped_type );

reference index < page, Allocator, Container> :: delete (const key_type &x);

reference index < page, Allocator, Container> :: operator [ ] (const page &p);
```

Some index private member functions.

```
page&  index < page, Allocator, Container> :: create_page ( );

page&  index < page, Allocator, Container> :: check_to_borrow
                                                           (page& left, page& right);

void index < page, Allocator, Container> :: borrow (page &from, page &to);

page& index < page, Allocator, Container> :: check_to_merge
                                                     (page &left, page &right);

void index < page, Allocator, Container> :: merge (page &from, page &to);

void index < page, Allocator, Container> :: fix_page (page &p );

iterator index < page, Allocator, Container> :: split_page (page &p );

iterator index < page, Allocator, Container> :: fix_sparse_page (page &p );

iterator index < page, Allocator, Container> :: pin_page (page &p );

void index < page, Allocator, Container> :: flush_page (page &p );

//For a B+ tree, we need the following additional methods :
```

```
iterator page <Key, T, Compare, Allocator> :: left_sibling ( );

const_iterator page <Key, T, Compare, Allocator> :: left_sibling ( ) const ;

iterator page <Key, T, Compare, Allocator> :: right_sibling ( );

const_iterator page <Key, T, Compare, Allocator> :: right_sibling ( )  const ;

iterator page <Key, T, Compare, Allocator> :: rbegin ( );

const_iterator page <Key, T, Compare, Allocator> :: rbegin ( ) const ;

iterator  page <Key, T, Compare, Allocator> :: rend ( );

const_iterator  page <Key, T, Compare, Allocator> :: rend ( ) const;
```

Interface example: B+ tree with integer.

We can implement the complete class from scratch, or include an internal STL container as an implicit container and just use it.

Interface for complete implementation method

```
#ifndef   page_H
#define   page_H
#include  <….>

template <class Key, class T, >  //using default comparison and default allocator
class page

        {
        public :
                // Constructors
                page ( )            { }      //constructs an empty page

                page (  InputIterator first,  InputIterator last ) { . . . }
                //constructs a page and inserts values in the range first to last.

                page (const page <Key, T>& x ) {  }
                //copy constructor

        class iterator
        {
          public:
                iterator (pair<Key, T>  * Initial = 0) :current (Initial ) { } //constructor
                T&  operator* ( )
                {
                 return current -> second ;
                }

                const T&  operator* ( ) const
                {
                  return current -> second ;
                }

                bool operator = = (const iterator&  x) const
                {
                 return current = = x . current;
                }


        iterator& operator ++ ( ) { }

                // . . .  with other operators and functions

        private:
                pair <Key, T> *  current;
                // . . . possibly with other data members

         } //  end iterator

        class const_iterator {  . . . }
        //An iterator class that allow read-only access to the container

        typedef  pair<const Key, T>&  reference;
        //type of the reference to the contents of the container

        typedef const pair<const Key, T>&  const_reference;

        typedef  Key  key_type;
        //typedef, allow for user defined key types

        typedef T mapped_type;
        //typedef of the class T

        typedef  pair<const Key, T> value_type;
        //typedef, the pair <const Key, T>

        typedef  ptrdiff_t  size_type;

        typedef  ptrdiff_t  difference_type;
        typedef  Allocator  allocator_type;

        typedef  less<T>  Compare key_compare;

        public:

        page ( )          {
            //construct an empty page
                          }

        page ( InputIterator first, InputIterator last )
                  {
                  //constructs a page and inserts the values in the range first to
                  //last.
                  }

        page (const page <Key, T, Compare, Allocator> & x )
        { //copy constructor }

        size_type  size ( ) const
        { return size ;}

        bool is_full ( ) const
        { return is_full ;}

        bool is_sparse ( ) const
        {return is_sparse ;}

        bool is_dirty ( ) const
        {return is_dirty ;}

        size_type max_size ( ) const
        { return maximum_size ;}

        iterator parent ( )
        {return parent ; }

        const_iterator parent ( ) const
        {return parent ;}

        iterator begin ( )
        {// return iterator to the first element }
        const_iterator begin ( ) const
        { //return iterator to the first element }
```

```
iterator end ( )
{ //return iterator to location after last element}
const_iterator end ( ) const{ }

iterator find (const key_type& x) { }
const_iterator find (const key_type& x) const { }

iterator  find_if (query& Query);
const_iterator  find_if (query& Query) const;

pair <iterator, bool> insert (const value_type& x){ }

iterator insert (iterator position, const value_type& x) { }

void erase (iterator position) { }        // delete object at position

size_type erase ( const key_type& x) { }
//delete all occurrences of object x and return the number of objects deleted

void erase (iterator first, iterator last) { }
// delete all objects in the range from first to last.

T& operator [ ] (const key_type & x) { }
//allow for directly indexing the objects by their keys

key_compare key_comp ( ) const{ }
// return the functor used in the page (for comparing keys)

iterator find_split_point ( ) { }
// return an iterator to the best position to split a page.

key_type find_page_key ( ) { }
//each page is able to extract the key of its contents

private:

size_type  size = 0;
bool is_full = false;
bool is_sparse = false;
bool is_dirty = false;
size_type max_size ;
iterator parent ;
```

Interface for implicit container method

We can design the page to be a class that use an implicit container as one of its private data members, namely the map, and thus use the existing map iterator and methods in the implementation of the page (easier).

```
template<class Key, class T>
class page
{
  public:

          typedef map <Key, T> Container

          //import these definitions from map to page. Now they'll apply to the page.
          typedef Container::iterator  iterator;
          typedef Container::const_iterator  const_iterator;
          typedef  Container::key_type   key_type;
          typedef  Container::mapped_type mapped_type;
          typrdef  Container::value_type value_type;
          typedef Container::size_type  size_type;

 private:
          bool is_full;
          bool is_sparse;
          bool is_dirty;
          size_type max_size;
          size_type current_size;

          iterator parent ;

          Container C;               //implicit map container

  public:
          page ( ) : is_full (0), is_sparse(0), is_dirty(0), current_size (0) { }
          //constructs an empty page

          page (  InputIterator first,  InputIterator last ) : is_full (0), is_sparse(0),
          is_dirty(0) , current_size (0) { }
          //constructs a page and inserts the values in the range first to last.

          page (const page&  x ) ;

          size_type  size ( ) const { return current_size ;}

          bool is_full ( ) const { return is_full ;}

          bool is_sparse ( ) const {return is_sparse ;}
```

```
    bool is_dirty ( ) const {return is_dirty ;}

    size_type max_size ( ) const { return max_size ;}

    iterator parent ( ) { return parent ; }

    const_iterator parent ( ) const { return parent ;}

    iterator begin ( ) {return C.begin ( ) ;}

    const_iterator begin ( ) const {return C.begin ( ) ;}

    iterator end ( ) {return C.end ( ) ; }

    const_iterator end ( ) const{ }{return C.end ( ) ;}

    iterator find (const key_type& x) ;

    const_iterator find (const key_type& x) const ;        //find first object equal to x

    pair <iterator, bool> insert (const value_type& x);

    iterator insert (iterator position, const value_type& x) ;

    void erase (iterator position) ;

    size_type erase ( const key_type& x) ;

    void erase (iterator first, iterator last) ;

    T& operator [ ] (const key_type & x) ;

    key_compare key_comp ( ) const ;

    iterator find_split_point ( ) ;

    key_type find_page_key ( ) ;

} //end class page
```

Usage examples.

```
//to create an index of pages:
index < page <int, smart_pointer>, Physical_Alloc>   An_index ;

//Internally, the index creates pages and bilk-load data into them:
page <int, smart_pointer>  page_1 ;

          //bulk-load the page
          for ( int i = 1 ,  i = 200 ,  i ++)
          {
                     . . .  "read data (a_key, a_smart_pointer) "
                  pair <int, smart_pointer >  p = make_pair  (a_key, a_smart_pointer) ;
                  page_1 . insrt (p) ;
          }
          . . .

//To insert a pair into the index:
          int  key_x = 515;
          smart_pointer  sp_x ( physical_data_reference ) ;
          . . .  =  An_index . insert ( key_x, sp_x ) ;

//To search for a key:
          int  key_y = 515 ;
          cursor  result ;
          An_index . find ( key_y , result) ;
          int  First_result =  * result ++ ;
          int  Second_result = * result ++ ;
```