AN EFFICIENT B+-TREE IMPLEMENTATION IN C++ USING THE STL STYLE

Jingxue Zhou, Bin Nie, Greg Butler

Department of Computer Science, Concordia University
1455 de Maisonneuve Blvd. West, Montreal, Quebec, H3G1M8, Canada

Keywords: B+-tree, STL style, design patterns, database index

Abstract: Database indexes are the search engines for database management systems. The B+-tree is one of the most widely used data structures and provides an efficient index. An efficient implementation is crucial for a B+-tree index. Our B+-tree index is designed to be a container in the style of the C++ Standard Template Library (STL) and implemented efficiently using design patterns and generic programming techniques.

1 INTRODUCTION

A database is any organized collection of information. An index is an auxiliary data structure intended to help speed up the retrieval of information in response to certain search conditions. To achieve this goal, a specialized handcrafted index is a good way to support a specific database application in a specific domain using domain-specific access methods. However, these specialized access methods are usually hand-coded from scratch. A specialized index may have better code efficiency and performance but the tradeoffs are development time and cost. The effort required to implement and maintain them is high.

Another choice is to develop a framework for a family of indexes, and reuse it to develop different indexes for different applications. A framework is a software infrastructure that may be tailored for building domain-specific applications, typically resulting in increased productivity and faster time-to-market. Therefore, an index framework should largely reduce the cost of providing a new index.

The Generalized Index Search Tree (GiST) (Hellerstein et al., 1995) is an existing framework of a generalized index system. It can be adapted to different key types and access methods. However, GiST has tried to satisfy all the possible needs of the future members of a family of applications, so it leads to code that is larger and less user-friendly. In addition, the source code itself is largely influenced by the C programming language and has poor object-oriented style.

As the framework development methodologies improve, these problems are being recognized and addressed. The Know-It-All Framework (Butler et al., 2002) is an object-oriented framework for database management systems. The Tree Index Framework is a subproject. It is being developed in C++ to conform to the style of the Standard Template Library (STL) collections and iterators. The index subframework covers tree-based indexes which include multi-dimensional trees and similarity-based retrieval. It also covers sequential queries, exact match queries, range queries, approximate queries, and similarity queries.

The B+-tree Index is designed to be a container that provides an iterator to its contents. The only way to interact with the container is through its iterator. Allocators are responsible for the memory management issues, and the Proxy mechanism is used to load a page from disk on demand and maintain the reference to the loaded page.

The index subframework (Nie, 2003) covers tree-based indexes such as B+-tree, R-tree, X-tree, SS-tree and their variants (Gaede and Günther, 1998). In the future we plan to include hash indexes and inverted file indexes as well. We report only the inital work on B+-trees (Zhou, 2003).

The main sections of the paper cover the design of the B+-tree, the implementation of the B+-tree, and the testing of the B+-tree and its performance. Before those main sections we present the background material, and after the main sections we conclude.

Zhou J., Nie B. and Butler G. (2004).
AN EFFICIENT B+-TREE IMPLEMENTATION IN C++ USING THE STL STYLE.
In *Proceedings of the Sixth International Conference on Enterprise Information Systems*, pages 163-168
DOI: 10.5220/0002613901630168

2 BACKGROUND

2.1 B+-tree

A database management system (Garcia-Molina et al., 2000) is a set of programs that allow users to define the type of data they want to store and manages that data by providing efficient retrieval. Efficient retrieval is done by using appropriate data structures such as B-tree, B+-tree or hash files as indexes within the database. An index is "a data structure that allows for random access to arbitrary data within a field, or a set of fields. In particular, an index lets us find a record without having to look at more than a small fraction of all possible records." (Garcia-Molina et al., 2000) From this definition, we can see that an index (Bayer and McCreight, 1972) consists of "index elements which are pairs (x, a) of fixed size physically adjacent data items, namely a key x and some associated information a. The key x identifies a unique element in the index, and the associated information is typically a pointer to a record or a collection of records in a random access file." All indexes are based on the same basic concept — Key and Reference to Data.

The B-tree and its variant B+-tree are efficient data structures that are widely used as tree-based multilevel indexes in database systems. They had already become so widely used (Comer, 1979) that "the B-tree is, de facto, the standard organization for indexes in a database system". However, B+-trees can support true indexed sequential access as virtual trees, and possibly compress separators and potentially produce an even shallower tree than B-trees (Folk and Zoellick, 1992). A B-tree (Bayer and McCreight, 1972) is a multi-way search tree designed to solve how to access and maintain efficiently an index that is too large to hold in memory, so the index itself must be external and is organized in pages that are blocks of information transferred between main memory and backup storage like hard disks. The power of B-trees lies in the following significant advantages:

1. Storage utilization is guaranteed to be at least 50% and should be considerably better in the average (Bayer and McCreight, 1972).
2. The balance is maintained dynamically at a relatively low cost. No overly long branches exist, and random insertions and deletions are accommodated to maintain balance (Folk and Zoellick, 1992).

The B+-tree retains the search and insertion efficiencies of the B-tree but increases the efficiency of searching the next record in the tree from O(log N) to O(1).

The B+-tree supports equality queries and range queries efficiently. Range queries use the forward or backward pointers in the leaf nodes to get all the records in the requested range.

2.2 The STL Style

The Standard Template Library (STL) (Stepanov and Lee, 1995) is a template-based C++ library of generic data structures and algorithms that work together in an efficient and flexible fashion. "The Standard Template Library provides a set of well-structured generic C++ components that work together in a seamless way. Special care has been taken to ensure that all the template algorithms work not only on the data structure in the library, but also on built-in C++ data structures."

There are six components in the STL organization. Three components, in particular, can be considered the core components of the library: template-based container classes, iterators and generic algorithms (template functions). The remaining three components of the STL are also fundamental to the library and contribute to its flexibility and portability: allocators, adapters and functors (function objects).

We adopt the STL style to design and implement B+-tree index because the STL supports good programming practices and addresses several problems with previous C++ container libraries in a new and innovative way. There are a number of advantages to using the STL:
1. "Standard" and "template": The STL is made up of "standard components". Each of them has a clear standard interface and a well-defined functionality. This makes all the components easy to understand and to reuse. Also new components may be added with the same look as standard ones. Programming with "templates" is a compiler-supported mechanism to take generic data structures, such as arrays and lists, and generic algorithms, such as sort and binary search, and make them independent of the type of data being manipulated.
2. Reuse: The STL supports the generic programming paradigm, whose goal is to design algorithms so they are fundamentally independent from the types they act upon. The STL provides reusable components to achieve code reuse based on templates, rather than class inheritance. A large number of components 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.
3. Smaller source codes: The STL is easy-to-learn because the library is quite small owing to the high degree of generality.
4. Flexibility: The use of generic algorithms allows algorithms to be applied to many different structures. Furthermore, the STL's generic algorithms also work

on native C++ data structures such as strings and arrays. The STL framework has a flexible design by adopting a complete component replacement policy. No component is made mandatory to the design. In other words all the components that make up the system are replaceable.
5. Efficiency: The STL is efficient because "Much effort has been spent to verify that every template component in the library has a generic implementation that performs within a few percentage points of the efficiency of the corresponding hand coded routine" as described by Alexander Stepanov and Meng Lee in the STL specification. STL containers are very close to the efficiency of hand-coded, type-specific containers. The STL has been already written, debugged, and tested.

2.3 Design Patterns

"Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" (Gamma et al., 1994). The purpose of design patterns is to reuse solutions and establish common terminology. Patterns are an attempt to describe successful solutions to common software problems by experts in software architecture and design.

We introduce the design patterns in our design of the B+-tree index.
1. Casting-method: The intent of the Casting-method pattern (Meyers, 1992) is to dynamically and quickly obtain a type-safe reference to a subclass in an inheritance hierarchy. The Casting method pattern uses inheritance to allow subclasses to return references to themselves. This pattern is applicable when there is a need to obtain a downcasted class reference from a base class and when real-time constraints require a fast and safe solution.
2. Composite: The intent of the composite pattern (Gamma et al., 1994) is to compose objects into tree structures to represent whole-part hierarchies in a manner that lets clients treat atomic objects and compositions uniformly. As a consequence this simplifies the Client and makes changes or the additions to the component very simple.
3. Proxy: The intent of the Proxy design pattern (Gamma et al., 1994) is to provide a surrogate or placeholder to control access to an object. Proxies provide a level of indirection to specific properties of objects, so they can restrict, enhance or alter these properties. Proxy is applicable whenever there is a need for a versatile or sophisticated reference to an object.

Smart pointers (Alexandrescu, 2001) are objects that look and feel like pointers, but are smarter. It is an application of the Proxy design pattern. To look and feel like pointers, smart pointers need to have the same interface that pointers do: they need to support pointer operations like dereferencing (operator $*$) and indirection (operator $->$). To be smarter than regular pointers, smart pointers need to do things that regular pointers do not. Probably the most common bugs in C++ (and C) are related to pointers and memory management: dangling pointers, memory leaks, allocation failures, locking and others.
4. Singleton: The intent of the Singleton design pattern (Gamma et al., 1994) is to ensure a class has only one instance and provide a global point of access. The Singleton class hides the operation that creates the instance behind a static member function. This member function, traditionally called *Instance(),* returns a pointer to the sole instance. Clients access the singleton by calling the static instance function to get a reference to the single instance and then using it to call other methods

3 DESIGN

The B+-tree index is designed to be an associative container like multimap in the C++ STL. The index container will be composed of pairs (Key, DataRef), where the Key is the access key type and DataRef is a reference to the true location of data. Both Key and DataRef are passed to the index as template parameters. A B+-tree index index pages and leaf pages, but they are invisible for users. What users operate on are not pages but pairs. The index and leaf page are also containers on a smaller scale. The elements of a leaf page container are pairs (Key, DataRef). The elements of an index page are pairs of the form (separator, child-pointer) where a child-pointer is the address of a lower page and a separator provides information about the boundaries between the two pages in the sequence set of a B+-tree, so child-pointers have one more than separators. A separator may be a prefix from page key or an exact copy of the page key of the lower page that the child pointer points to. In our design, the page key of a leaf page is the first key but the page key of an index page is the page key of the leftmost leaf page if the child pointer is treated as the root of a subtree.

While a B+-tree index typically resides on hard disk, a page is small enough to fit in memory. Whenever a page is needed, it is retrieved from the hard disk into memory through a proxy. At this point the page can perform its tasks of searching for a key in its contents, accepting new entries, and deleting some existing ones.

The major classes are BplusTree, which represents the B+-tree as a whole, the abstract class Page and its concrete subclasses IndexPage and LeafPage. Figure 1 shows the main interface of B+-tree index. It

is important for the BplusTree, Page, IndexPage, and LeafPage containers to conform to a set of abstract concepts provided by the STL.

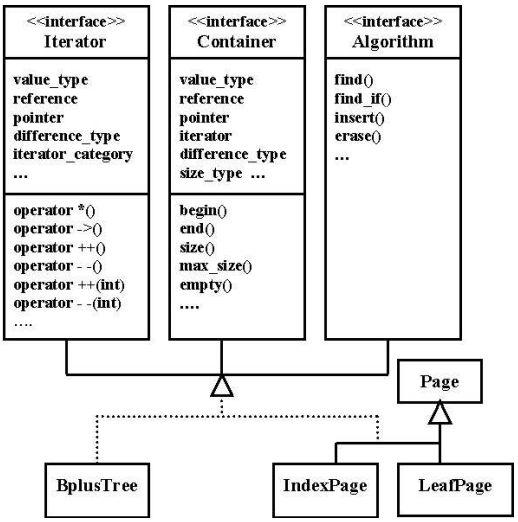

Figure 1: STL Interfaces for the B+-Tree Index.

4 IMPLEMENTATION

The design of the major classes is shown in Figure 2. We investigated several designs in order to cleanly and uniformly treat all pages the same, including the issue of loading pages from disk. This design here combines the composite pattern with the casting-method pattern to resolve the issues.

4.1 Page Class

Page is an abstract class which defines an interface for its subclasses: IndexPage and LeafPage. This base class uses casting methods to obtain a type-safe reference of an object in the class. As a result, BplusTree only holds a pointer to a Page but it can get references to the index page and leaf page through this pointer, and then can invoke the class-specific functions such as *begin(), end()* and *insert()* through these references.

IndexPage and LeafPage are designed to be template container classes in the spirit of the STL, so they must conform to all STL interface characteristics. 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.

4.2 LeafPage Class

The LeafPage class is an associative container that supports elements with duplicate keys. Many STL containers can be used, but we use multimap because it has efficient retrieval, and bidirectional iterators.

4.3 IndexPage Class

An index page is also designed to be an associative container. The IndexPage container is invisible to applications. It is created and managed by the tree. There is a mismatch in number between the separators (keys) and the child pointers that make up the pairs for an IndexPage: there is one more child pointer than separator. This complicates our view of an IndexPage as a container of pairs. In our implementation, two vectors are needed: one for keys (called the key container), and the other for child pointers (called the child pointer container). There are two kinds of iterators provided by two vectors respectively but the index page container uses the iterator of the child pointer container as its external iterator. The iterator of the key container is only used as an internal iterator.

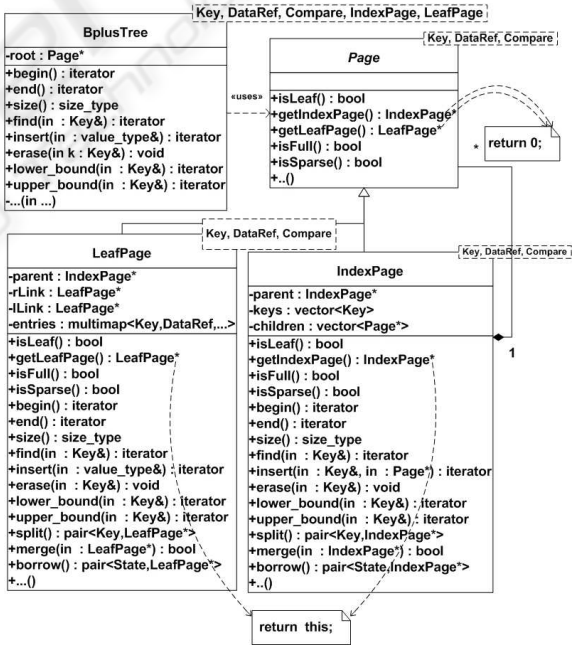

Figure 2: Main Classes of B+-Tree Index.

4.4 B+-Tree

The implementation of the B+-tree index container is based on Leaf Page and Index Page containers. The

B+-tree index is initialized from an empty LeafPage container. However, the index will dynamically grow or shrink with insertions or deletions. The B+-tree index container is designed and implemented to be an associative container, so it supports equality and range-searches efficiently.

The B+-tree index container only holds a pointer to a root page. In the operations of B+-tree index, the required pages are loaded on demand through a proxy mechanism.

4.4.1 Iterator

An iterator is the only way to access to elements within B+-tree index containers. The Iterator is a nested class defined within a B+-tree index container class and is a friend to this container. A B+-tree index container is a doublely-linked list of leaf pages, but the elements that B+-tree iterators are iterating over are pairs of key and data reference. Therefore, a B+-tree iterator should point to a pair: a leaf page pointer to the page where the pair is stored, and the leaf page iterator that points to this pair.

4.4.2 Proxy

The B+-tree index uses a Proxy mechanism to manage access to the storage of the index. Only the root of a B+-tree is loaded initially, and resides in memory until the B+-tree is destroyed. Each access to a non-root page checks if the page is in memory. If yes, the Proxy returns a smart pointer to the tree. Otherwise, the Proxy will check if the Cache has a reference to the page. If the page reference is in the Cache, the tree algorithm can quickly get a smart pointer to the page. If not, the Proxy needs to read the page object from the storage.

4.4.3 Cache

The global cache management consists of two processing components (Huang and Stankovic, 1990): allocation and replacement. Allocation distributes global buffer space among concurrent transaction and replacement is responsible for accessing of the global buffer and page replacement operations. The life span of a page, except the root, in memory depends on the replacement strategy of the Cache. When a page is removed from the cache, it will be destroyed in the memory heap. In our B+-tree index, we use a Least Recently Used (LRU) replacement strategy.

4.4.4 Storage and Serialization

The Storage class is mainly responsible for managing and controlling accesses to the index files on disk. In the B+-tree index, a block is the basic unit for I/O operation. When a new page object is needed, Storage allocates a block for it on the physical storage. When a page object is deleted from the files, Storage garbage-collects the block used by the page and reallocates it.

Serialization is used to read or write a page to or from the index files. The basic idea of serialization is that a page should be able to write its current state to persistent storage. Later, the page object can be re-created by reading, or deserializing, the object's state from the disk.

5 TESTING

5.1 Correctness Testing

Correctness testing of the B+-tree index focuses on testing the insert, delete and find operations. We use the Berkeley DB (Olson et al., 1999) test suite which is a complete test suite for relational databases, not just indexes. Besides the existing test cases for B-tree index testing in this test suite, we also create some special test cases for illegal inputs, large inputs, and values smaller or larger than the specified range.

5.2 Performance Testing

Performance is always a great concern for database indexes. The goal of performance testing can be performance bottleneck identification for code tuning and optimization, or for performance comparison and evaluation. We did both. We used a benchmark dataset from GiST, as well as randomly generated datasets.

Our platform for performance testing was a Sun-Fire 280R with two UltraSparc-III+ CPUs running at 900MHz, 4GB memory, using the Solaris 9 operating system and the GNU g++ 3.2 compiler. The files were on a Network Appliances file server accessed via a Gigabit Ethernet.

The first dataset was a dataset provided with GiST. The dataset contains 10,000 random integers as keys. For the test, we set the size of a page to be 8KB, which is is the size of a block on the test platform; and we set the buffer (cache) to hold at most 16 pages. Then we recompiled GiST v1.0 and our KIA B+-tree index. For this set up, a page will contain at least 500 keys and at most 1000 keys if the Data Reference is treated as an integer. The B+ tree should have at least 21 pages in two levels. The test performed each of the following three tasks and timed them for ten separate runs, reporting the average of the ten repetitions.

1. Insert each (key, pointer) pair in the dataset;
2. Find the first position with a key $\geq$ 20000; and

Table 1: Performance Comparison (Time in μsecs)

	KIA B+-tree Time	Gist B+-tree Time	Ratio KIA:Gist
10,000 keys, GiST dataset			
Insertion	260,385.9	626,468.4	0.4
Search	25.3	64.5	0.4
Deletion	734.4	2,389.1	0.3
100,000 keys, random dataset			
Insertion	2,250,000	223,362,386	0.1
Search	35	4,095	0.008
Deletion	2,270,000	297,000	7

3. Delete all the elements where the key $<$ 20000.

The second dataset contained 100,000 keys which were random integers in the range 0..32767. We set the buffer to hold 128 pages, with the page size still set at 8KB.

Table 1 shows the test results that are the average time (microseconds) of 10 tests under the same conditions. Except for the anomaly of the deletion time for the second dataset (which we still do not fully understand), our implementation is significantly more efficient than that of GiST. We did attempt a comparison using one million keys: our B+-tree worked fine, but we could not repeat the test using the GiST implementation.

6 CONCLUSION

In this paper we describe how to build a B+-tree index using C++ template mechanisms, design patterns, and the STL style in order to achieve flexibility and efficiency. The index can easily handle arbitrary keys and data references. The index is extremely efficient.

The adoption of the STL style promotes code reuse, increases readability and user friendliness, and reduces time and cost overheads incurred during the application development process. Design patterns simplify the design complexity by separating design concerns at the micro-architecture level. The combination of the STL style and design patterns makes our B+ tree index general and reusable. Several design patterns such as Composite, Casting method, Proxy, and Singleton were used because they provided a model of how to solve our design issues, many of which dealt with introducing extensibility into the design in order to make it more reusable.

REFERENCES

Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison Wesley Professional, London, 2nd edition.

Bayer, R. and McCreight, E. (1972). Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189.

Butler, G., Chen, L., Chen, X., Gaffar, A., Li, J., and Xu, L. (2002). The Know-It-All project: A case study in framework development and evolution. In *Domain Oriented Systems Development: Perspectives and Practices*, pages 101–117. Taylor and Francis Publishers.

Comer, D. (1979). The ubiquitous B-tree. *Computing Surveys*, 11(2):121–137.

Folk, M. J. and Zoellick, B. (1992). *File Structures*. Addison Wesley.

Gaede, V. and Günther, O. (1998). Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Garcia-Molina, H., Ullman, J. D., and Widom, J. (2000). *Database System Implementation*. Prentice-Hall.

Hellerstein, J. M., Naughton, J. F., and Pfeffer, A. (1995). Generalized search trees for database systems. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 562–573.

Huang, J. and Stankovic, J. A. (1990). Buffer management in real-time databases. COINS Technical Report 90-65, University of Massachusetts.

Meyers, S. (1992). *Effective C++*. Addison-Wesley.

Nie, B. (2003). A tree index framework for databases. Master's thesis, Concordia University.

Olson, M. A., Bostic, K., and Seltze, M. (1999). Berkeley DB. Software.

Stepanov, A. and Lee, M. (1995). *The Standard Template Library*. Hewlett-Packard.

Zhou, J. (2003). A B+-tree index for the Know-It-All database framework. Master's thesis, Concordia University.