Portland State University

PDXScholar

Dissertations and Theses Dissertations and Theses

5-23-1996

Abstract Index Interfaces

Muralidharan Janakiraman
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Computer Sciences Commons

Let us know how access to this document benefits you.

Recommended Citation
Janakiraman, Muralidharan, "Abstract Index Interfaces" (1996). *Dissertations and Theses.* Paper 5288.
https://doi.org/10.15760/etd.7161

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

THESIS APPROVAL

The abstract and thesis of Muralidharan Janakiraman for the Master of Science degree in Computer Science were presented May 1, 1996, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

Leonard Shapiro, Chair

Goetz Graefe

Sergio Antoy

Kent Lall
Representative of the office of Graduate Studies

DEPARTMENT APPROVAL:

John McHugh, Chair
Department of Computer Science

****

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by on 23 May 1996

ABSTRACT

An abstract of the thesis of Muralidharan Janakiraman for the Master of Science degree in Computer Science presented May 1, 1996.

Title : Abstract Index Interfaces

An index in a database system interacts with many of the software modules in the system. For systems supporting a wide range of index structures, interfacing the index code with the rest of the system poses a great problem. The problems are an order of magnitude more for adding new access methods to the system. These problems could be reduced manifold if common interfaces could be specified for different access methods. It would be even better, if these interfaces could be made database-system independent.

This thesis addresses the problem of defining generic index interfaces for access methods in database systems. It concentrates on two specific issues: First, specification of a complete set of abstract interfaces that would work for all access methods and for all database systems. Second, optimized query processing for all data types including user-defined data types.

An access method in a database system can be considered to be made up of three specific parts: Upper interfaces, lower interfaces, and type interfaces. An access method interacts with a database system through its upper interfaces, lower interfaces and type interfaces. Upper interfaces consist of the functions an index provides to a database system. Lower interfaces are the database-system dependent software modules an index has to interact with, to accomplish any system related functions. Type interfaces consist of the set of functions an index uses, which interpret the data type. These three parts together characterize an access method in a database system.

This splitting of an access method makes it possible to define generic interfaces. In this thesis, we will discuss each of these three different interfaces in detail, identify functionalities and design clear interfaces. The design of these interfaces promote development of type-independent and database-system independent access methods.

ABSTRACT INDEX INTERFACES

by

MURALIDHARAN JANAKIRAMAN

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
1996

Acknowledgments

I would like to thank my advisor, Dr. Goetz Graefe, for his guidance and direction in focusing the content of this thesis and ensuring its scholarship. I would also like to thank Dr. Leonard Shapiro for his insights, guidance, for the time and effort spent in reviewing the contents of the thesis and his acceptance to take the official advisory role in the absence of Dr. Goetz Graefe to ensure its completion. Thanks also to the members of the thesis committee for their time and effort in reviewing the contents of this thesis.

Table Of Contents

Chapter 1	**Introduction**	1
1.1	ADTs in Database Systems	1
1.2	Indices in Database Systems	2
1.3	About this Work	2
1.3.1	Design Goals	3
1.3.2	Design Limitations	3
1.3.3	Benefits of this Work	4
1.4	Related Work	4
1.5	Organization of the Rest of the Chapters	6
Chapter 2	**Access Methods**	7
2.1	Access Methods - A Survey	7
2.2	An Index in a Data base System - A Modular Perspective	12
Chapter 3	**Type-Dependent Interfaces**	15
3.1	Design Considerations	15
3.2	Interface Specification	17
3.2.1	KEY class	17
3.2.1.1	KEY Record	18
3.2.1.2	KEY Class Definition	20
3.2.1.3	Detailed Specification and Member Function Definition	22
3.2.2	Match Class	42
3.2.2.1	Class Definition	42
3.2.2.2	Detailed Definition	42

3.3	Usage	44
3.3.1	Implementation Class - An Example	44
3.3.1.1	STRING_KEY Implementation	45
3.3.1.1.1	Class Definition	45
3.3.1.1.2	Class Implementation	46
3.3.1.2	STRING_MATCH Implementation	52
3.3.1.2.1	Class Definition	52
3.3.1.2.2	Class Implementation	52
3.3.2	Application Design	53
3.3.3	Index Interfaces Implementation	53
3.4	Discussion	53
Chapter 4	**Upper Interfaces**	55
4.1	Data Definition and Data Manipulation Language Interfaces	55
4.1.1	Interface Specification	55
4.1.2	Discussion	64
4.2	Optimizer Support Interfaces	65
4.2.1	Design Considerations	65
4.2.2	Interface Specification	66
4.2.3	Discussion	67
4.3	Log Support Interfaces	69
4.3.1	Interface Specification	69
4.3.2	Discussion	71
Chapter 5	**Lower Interfaces**	72
5.1	Design Considerations	72
5.1.1	Software Modules an Index Interacts With in a DBMS	73

5.2	Disk Manager	74
5.2.1	Interface Specification	75
5.2.2	Discussion	78
5.3	Record Manager	78
5.3.1	Interface Specification	79
5.3.2	Discussion	83
5.4	Buffer Manager	84
5.4.1	Interface Specification	85
5.4.2	Discussion	85
5.5.	Log-Recovery Sub-System	85
5.5.1	Interface Specification	87
5.5.2	Discussion	87
5.6	Lock Manager	88
5.6.1	Interface Specification	89
5.6.2	Discussion	90
5.7	Transaction Manager	90
5.8	Catalog Manager	93
5.8.1	Index Header Record	93
5.8.2	Interface Specification	94
5.8.3	Discussion	94
Chapter 6	**Prototype Implementation**	96
6.1	Cascades	96
6.1.1	Index Interfaces of Cascades	97
6.1.1.1	Data Structures	97
6.1.1.2	File Management	99
6.1.1.3	Record Management	101

6.2	Access Methods Implementation	105
6.2.1	B+-tree Implementation	106
6.2.2	R-tree Implementation	106
6.2.3	Testing	107
Chapter 7	**Summary and Conclusions**	109
7.1	Conclusions	110
7.2	Future Work	110
7.2.1	Single Interface	110
7.2.1	Templates	111
7.2.2	Multi-threading	111
Appendix A	List of Access Methods Considered for this Work	113
Appendix B	Data Structure Definitions	114
References		120
Tables	Table 1	11
Figures	Figure 1	13

1 INTRODUCTION

1.1 ADTs in Database Systems

The collection of built-in data types such as integers, floats, characters, and built-in operators such as plus, minus, multiplication, and division, in a database management system were motivated by the needs of business data processing applications. These built-in data types are not adequate for new and emerging database applications including information and knowledge-based systems, engineering test and measurement, hardware and software design, and geographical applications[Stonebraker 1983; Stonebraker 1986].

For example, a geographical application may be better served by a database management system that includes points, lines, circles, and polygons as its basic data types and operators such as intersection, containment, and overlapping as its basic operators along with its standard built-in data types and operators. A scientific application may require complex numbers and time series with appropriate operators. Even a business application can make better use of user-defined data types. For example, it may be advantageous to define data types like date or time, and to define addition, subtraction and comparison operations on them. Most of the commercial database systems have started to support date, time and even interval as their built-in data types. Detailed discussions on new types in relational database systems can be found in [Stonebraker 1983; Stonebraker 1986].

While it is clear that addition of non-conventional data types to the basic type collection and facilities for user-defined data types do provide many advantages to the user of a database system, one might ask how these new data types affect the storage structures and indexing in the system.

1.2 Indices in Database Systems

Most of the commercial and research systems implement B-tree[Comer 1979] and hashing as access methods for better retrieval performance. These access methods work very well for the standard built-in data types. These may also be extended to support some user-defined data types, such as date and time without affecting the retrieval performance. However, these classical one-dimensional indexing structures are not appropriate for multi-dimensional spatial searching.

A number of index structures have been proposed in the literature for handling multi-dimensional data, and a survey of methods can be found in [Bentley 1979]. Index structures such as R-trees, KDB-trees etc., promise better retrieval efficiency for multi-dimensional data types[Chapter 2 discusses this in detail]. As different access methods work well for different data types, it becomes necessary for a database system supporting a wide collection of data types to use appropriate indexing structures, to provide better overall retrieval performance. Moreover, a type extensible system may also have to provide facilities to add new access methods to the existing collection.

Supporting many different access methods in the system brings a host of challenges to the access method designers. We discuss these issues in the next section.

1.3 About This Work

An access method in a database system interfaces with many of the software modules in the system. For systems that support many different access methods, interfacing the index code with the rest of the system poses a great challenge. More than fifty percent of the complexity and effort [Stonebraker 1986] in any access method implementation, results from interfacing the index code with the rest of the system. The problems are an order of magnitude greater when adding new access methods to the system.

Moreover, for systems that support a wide collection of built-in data types and user-defined data types, the access method implementations have to provide a type-independent interface, in order that they can be made use of without any modifications to the original code.

These problems could be reduced manifold, if generic interfaces that would work for all access methods and for all data types could be specified for access methods in database management systems. In this work, we address the problem of defining generic type-independent interfaces for all access methods in database systems. We start with specifying the design goals.

1.3.1 Design Goals

Following are the major design goals we intend to meet in this work.

- The interface set must work for all access methods(Subject to the limitation discussed in Section 1.3.2) and must be type-independent.
- The interface set must be complete. It should provide a complete set of operations that can be provided by an access method to a database system and it should include a complete set of interfaces for interfacing the index code with the rest of the database system to accomplish all database-system dependent functionalities..
- The interface design should not target any specific database system, and must be generic enough to work for any database system with little or no changes.
- Design should not sacrifice query optimization effectiveness for generality.

1.3.2 Design Limitations

In Chapter 3, we will identify a set of type-dependent functions, some of which are access method specific. One such example is a hash function. Although, we do not know of any other access method requiring a special type-dependent operator such as hashing, the

existence of access-method dependent type operators limit the scope of our design. The finite set of type-dependent operators we identify in Chapter 3 is based on the needs and requirements of a finite set of access methods. The access methods we have taken into consideration for the interface design are listed in Appendix A. Our interface is designed to work only for this finite set of access methods. However, as existence of access method specific type operators are limited, we expect our interface design to work for most of the available access methods. Any reference to the term "all access methods" in this thesis, only means all the access methods listed in Appendix A.

1.3.3 Benefits Of This Work

Some of the major benefits of our work are :

- It provides uniform interfaces to all access methods in the system.
- It allows usage of existing access methods for user-defined data types.
- It permits optimized query processing for user-defined datatypes as well as built-in datatypes.
- It makes it easier to add new access methods to the system as implementation of an access method is not tied to a specific database system.

1.4 Related Work :

The concept of abstract index interfaces has already been discussed in the literature. Some research database systems have implemented these concepts in varying degrees of abstraction. We expect the current commercial systems to have implemented some level of abstraction in their index interface design.

Modular and extensible database systems such as Postgres, Genesis, and Exodus have discussed and implemented these concepts in greater detail. Of these three systems, Postgres more closely supports the concepts discussed in this thesis. Though these systems

support extensibility in the areas of adding new datatypes, new access methods and extending existing access methods to new data types, we believe their basic motivation was not to identify “Generic Index Interfaces” that would work for many database systems. Their index interfaces are tied to their respective systems. In the next paragraphs of this Section, we discuss some of the major differences between our work and the existing work that has motivated us to take up this research.

The first difference has to do with the functionality provided by the existing index interfaces and the completeness of the existing interfaces. Most of these systems provide index interfaces to create, destroy, open or close an index and to insert, retrieve, delete and modify records in the index. We believe an index could provide more functionality than that. In this thesis, we identify two other areas: support for the optimizer, and support for the log-recovery sub-system which could be provided by an index to any database system.

The next major difference is in interfacing the index with the rest of the system. In Exodus, the index sits on top of the storage manager. In Genesis, the index (the FILE manager) sits on top of the NODE manager. Both these systems are page-oriented. In other words, the access methods are not involved in transaction or logging. Though this architecture is simple, it is not efficient. For instance, one can not do event logging or take advantage of special concurrency protocols possible on different index structures. On the other hand, Postgres provides all this features. A new access method could be added to the system by implementing 13 new functions. But Postgres interfaces are tied to the system. One has to be aware of Postgres page-structure, Postgres way of pinning and unpinning pages from buffer and Postgres locking techniques. In our work, we try to remove this dependency on the system-related details.

The third major difference has to do with extending the existing access methods to work for new data types. In this area, we believe Postgres provides the best support for extensibility. But, Postgres also has some drawbacks. First, one can not extend an index for

any constructed type. Indices work only on basic types. Second, extending an index to a new type is not trivial. There are many system catalog entries to be made. The system could crash if the entries happen to be wrong. In this thesis, we try to provide extensibility without any limitation. In addition, our solution provides more type-safety and it lets the compiler do most of the job with very minimal dependency on the system catalog.

1.5 Organization of the Remaining Chapters

In the next chapter, we discuss in detail various access methods and try to identify different characteristics that uniquely distinguish the many different access methods. Chapter 2 also defines a modular perspective for access methods in database systems, and identifies three major layers of interfaces. Chapters 3, 4, and 5 discuss each of these three layers in detail, identify functionality to be supported in each layer, and specify clear interfaces for each of the identified functions. In Chapter 6, we provide an overview of our prototype implementation. Chapter 7, summarizes the discussion and concludes the thesis.

2 ACCESS METHODS

In order to reduce the number of accesses to secondary storage, which is relatively slow compared to main memory, most database systems employ associative search techniques in the form of indices that map key or attribute values to locator information, with which database objects can be retrieved. Typically, in relational systems, an attribute value is mapped to a tuple or record identifier.

A number of index structures have been described in the literature, e.g.,[Bayer 1972; Becker 1991; Beckmann et al. 1990; Bentley 1975; Finkel 1974; Guenther 1991; Gunther 1987; Gunther 1989; Guttman 1984; Henrich 1989; Hoel 1992; Hutflesz 1988a, 1988b, and 1990; Jagadish 1991; Kemper 1987; Kolovson 1991; Kriegel 1987, and 1988; Lomet 1990a; Lomet 1992; Neugebauer 1991; Robinson 1981; Samet 1984]. Although all indices are based on the same basic concept - keys and reference fields - the wide variety of access methods can be described by some characteristics that pertain to the physical structure of the index, the layout, and use of pages on disk. These characteristics distinguish them from one another making each index uniquely suitable for different applications.

In this chapter, we survey a number of access methods based on their characteristics and define a modular perspective for indices in database management systems.

2.1 Access Methods - A Survey

[1] The best known and most-often used database index structure is the B-tree[Bayer 1972; Comer 1979]. A large number of extensions to the basic structure and its algorithms have been proposed, e.g., B+-trees for faster scans, fast bulk loading from a sorted file, increased fan-out and reduced depth by prefix and suffix truncation, B*-trees for better space utilization in random insertions, and top-down B-trees for better locking behavior through

[1] Most of the discussion in this section has been taken from [Graefe 93].

preventive maintenance[Guibas 1978]. B-trees are still a active research topic, in particular with respect to concurrency control[Srinivasan 1991], improved space utilization[Baeza-Yates1989], parallelism[Seeger 1991], and on-line creation of B-trees for very large databases. On line re-organization and modification of storage structures become a important topic as databases become larger and larger and are spread over many disks and many nodes in parallel and distributed systems.

While B-trees are the most commonly used index structure in database systems, there exists a wide variety of index structures, which have potential for use in database systems. The large variety of index structures can be described by the following characteristics.

First, is the index structure static or dynamic?. Either the index structure allocates a fixed number of buckets when it is first created and resorts to overflow pages if buckets cannot hold all data items that logically belong in them, or it reorganizes itself incrementally as items are inserted and deleted. Database systems prefer dynamic structures such as B-trees as they adapt gracefully to growing data volumes. Static structures such as ISAM do have a distinct advantage with respect to concurrency control and recovery, since their internal nodes do not change on insertions and deletions, except during re-organization.

Second, is the index support range retrievals and ordered scans, or only exact-match equality retrievals? This is the main difference between sort-based indices such as B-trees and hash-based indices. Indices that support ordered key domains tend to have logarithmic insertion, deletion, and search costs, while index and storage structures based on hashing typically have constant average maintenance complexity.

Third, does the index structure support only single-dimensional data or also data representing multiple-dimensions? There are a number of application areas where it is very common to perform searches using the values of several attributes. Examples of such areas include geographic or geometric data, VLSI design, and certain kinds of document retrieval.

Where multiple-attribute searches are the rule and single-attribute searches the exception, there are advantages to using one multi-attribute index compared with several single-attribute indexes[Note that a multi-attribute index can be implemented using one single multi-dimensional index or with several single-dimensional index structures]. First, the clustering of index terms and data on disk can dramatically reduce the number of I/O accesses needed for the search. Second, when new records are inserted, a multi-attribute organization needs only a single update of its index. Multiple single-attribute indices require multiple updates. True multi-dimensional indices support all dimensions as equals. Another issue closely related with dimension is, do the indices support point data or range data? Range data have two data points in each dimension; the standard example is the case of two-dimensional rectangles.

A number of structures have been proposed for handling multi-dimensional point data and range data and a survey of methods can be found in [Bentley 1979]. Cell methods[Bentley 1977; Guttman 1982; Yuval 1975] are not good for dynamic structures because the cell boundaries must be decided in advance. Quad trees[Finkel 1974] and k-d trees[Bentley 1975] do not take paging of secondary memory into account. K-D-B trees are designed for paged memory but are useful only for point data. The use of index intervals has been suggested in[Wong 1977], but this method cannot be used in multiple dimensions. Corner stitching [Ousterhout 1982] is an example of a structure for two-dimensional spatial searching suitable for data objects of non-zero size, but it assumes homogeneous primary memory and is not efficient for random searches in very large collections of data. Grid files [Hinrichs 1983] handle non-point data by mapping each object to a point in a higher-dimensional space. H-B trees [Lomet 1990] are derived from K-D-B trees and k-d trees. R-tree [Guttman 1984] is one of the most general multi-dimensional index structure that has been actually implemented in a complete database management system - Postgres[Stonebraker 1990].

Finally, most index implementations can be switched to accept or reject duplicate keys. Thus, indices are used to enforce uniqueness constraints, particularly for identifying keys in database systems.

Table 1 shows some example index structures classified on the above basis.

Classification of Some Access Methods

Structure	Ordered	Dynamic	Multi-dim Data	Range	References
ISAM	Yes	No	No	No	[Larson 1981]
B-trees	Yes	Yes	No	No	[Bayer 1972; [Comer 1979]
Quad-trees	Yes	Yes	Yes	Yes	[Finkel 1974]
KD-trees	Yes	Yes	Yes	No	[Bentley 1975]
KDB-trees	Yes	Yes	Yes	No	[Robinson 1981]
HB-trees	Yes	Yes	Yes	No	[Lomet 1990a]
R-trees	Yes	Yes	Yes	Yes	[Guttman 1984]
Extendible Hashing	No	Yes	No	No	[Fagin et al 1979]
Linerar Hashing	No	Yes	No	No	[Litwin 1980]
Grid-Files	Yes	Yes	Yes	No	[Nievergelt 1984]

2

Table 1 : Classification of Some Access Methods.

[2] Source : [Graefe 93]

2.2 An Index in a Database System - A Modular Perspective

In the last section, we discussed various index structures and compared them using different characteristics that pertain to their physical structure, the layout, and use of pages on disk. In this section, we will analyze how an index or access method fits in a database system.

An index in a database system interacts with many of the software modules in the system. On one hand, it provides a set of DDL and DML functions to the higher-level software in the system for defining and manipulating the underlying data. On the other hand, it interacts with various lower level system modules for physical manipulation of data and for maintaining database consistency. In this section, we attempt to classify this set of modules into three specific layers and outline the interaction of the index with each layer. The classification we define here forms the basis for designing abstract interfaces. Chapters 3, 4 and 5 will discuss each layer in detail.

An access method in a database management system can be considered to be made up of three major parts: Upper interfaces, lower interfaces, and type-dependent interfaces. Figure 1 illustrates how an index fits in a database management system and how the three different parts relate to one another and to the index.

Upper interfaces are the set of functions an index provides to the higher level software in the system. A facility to create an index or insert a record in an index are some of the functions that will typically fit in as upper interfaces. This interface depends only on the functionality an index can provide to the system. To design abstract interfaces one must first identify a complete set of functions that an index can provide to the system.

Modular Perspective of An Index In a Database Management System

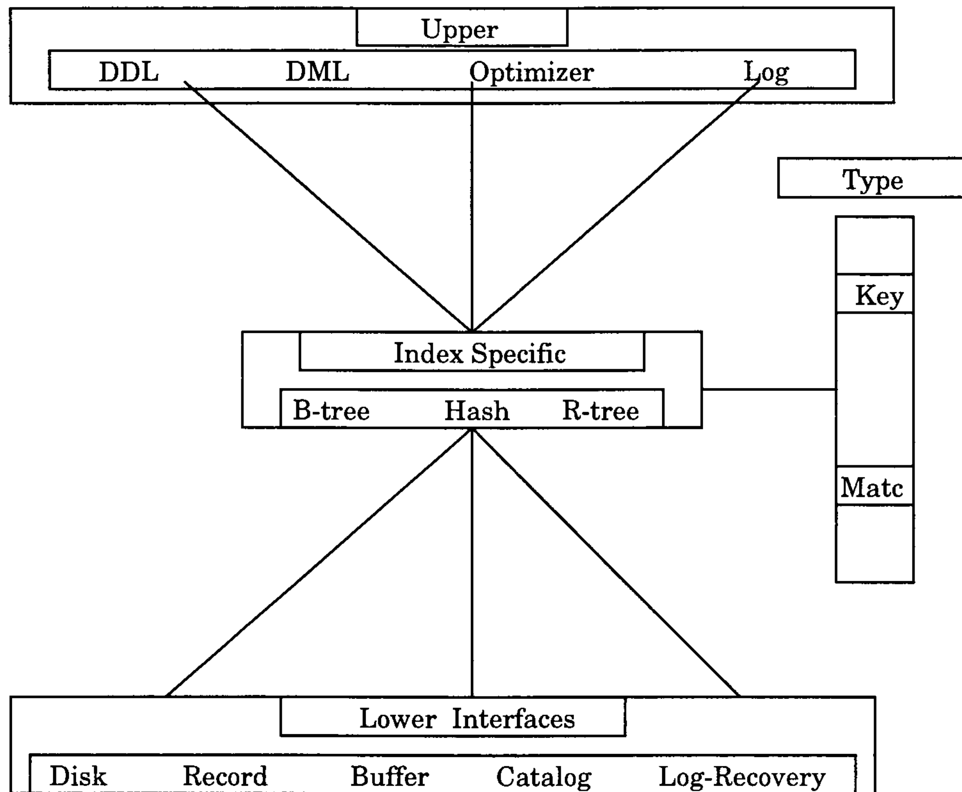


Figure 1

On the lower level, an index interfaces with many of the software modules in the system to accomplish database system related functions such as file management, transactions, logging, etc. We call this set of modules the lower interface. The lower interface depends on the functionality supported by the database system.

An index in any system often handles data of various types. The data types may be standard built-in types such as integers, characters, strings or may be any user-defined data type. To build a type-independent access method one must identify the functions that really need to interpret the data type. For example, a B-tree can be built using only two type-dependent operators: compare and copy. Compare is an operator that takes two keys as its input and determines whether one key is greater, less, or equal to another key. Copy is an operator that takes a key and an address of a memory location as its input and copies the key to the specified memory location. By providing a compare operator and a copy operator for different key types, we could make the B-tree work for all key types. These are only examples and different access methods may require different type-dependent operators. We call this set of functions type-dependent functions.

These three different parts together characterize an index in a database system. An index interacts or communicates with the database system through its upper interfaces, lower interfaces and type-dependent interfaces. We will define generic index interfaces by identifying and defining common interfaces for these three sets.

In the next three chapters, we discuss each of these three interfaces in detail.

3 TYPE- DEPENDENT INTERFACES

Chapter 2 gave a brief introduction to type-dependent interfaces. In this chapter, we will identify a set of type-dependent functions and will show how this set can be organized in the system. The set of type-dependent functions we identify here will be based on the needs and requirements of all access methods listed in Appendix A.

Our success here lies in identifying a correct and complete set of type-dependent functions that would work for all access methods(Section 1.3.2) and for all data types, without having to extend or modify the index code. The design should permit optimized query processing for all data types, including the user-defined data types, as that would be possible if the index has been coded for a particular data type.

Section 3.1 discusses the problem of identifying a finite set of type- dependent functions. In section 3.2., we specify the type-dependent interfaces.

3.1 Design Considerations

The first step towards building a type-independent access method is to identify the functions that really need to interpret the data type. For any access method, the set of functions that need to interpret the data type are dependent on the access method and on the data types the database system deals with. For example, a hash index requires a "hash" operator which is type-dependent. A B-tree requires a "compare" operator. Most multi-dimensional indices such as R-tree, and KDB-tree require an "overlaps" or a "contains" operator to check how two regions covered in a k-dimensional space relate to one another. Like wise, every access method requires its own set of type-dependent operators. Of course, some of the operators may be common to all access methods. For example, almost all indices require a "copy" operator.

In our opinion, the type-dependent operators of an access method can be classified into two different sets. One set that depends on the requirements of the index construction, traversal and maintenance algorithms. The other set that depends on the type of retrieval operations possible for a given key type. For example, let us consider a R-tree consisting of data on two-dimensional boxes. The R-tree can be constructed using an "overlap" and a "compare" operator (Overlap is used to determine whether a given box overlaps the region covered by the other box. Compare determines whether a region covered by one box is greater than the region covered by the other). These two operators are enough to create, traverse and maintain a R-tree.

Let us now examine the retrieval operations possible for this data type. Given that the data type stored in the R-tree is a box data type, all the following operations make sense[Stonebraker 1983].

Retrieve all the boxes that are contained in an unit square.

Retrieve all the boxes that intersect the unit square.

Retrieve all the boxes that are to the left of the unit square,

Retrieve all the boxes that are to the right of the unit square.

For a box data type the above retrieval operations are only a few. There are many more possible and it should be possible to perform retrieval using any operator that is appropriate for this box key type. Note that these retrieval operators depend more on the key type rather than on the access method. If the same R-tree is used to store a single dimensional integer key type, the retrieval options available may not be this many. On the other hand a 4 or a 5 dimensional data type may throw in a lot of different retrieval options.

From the ongoing discussions we could conclude that the set of type-dependent functions cannot only include all the operators that are required by the algorithms of different access methods but should also take into account various retrieval operations possible on any given key type. This distinction is important because, though we may be working with a finite

set of access methods(Appendix A), we are not working with a finite set of data types. That means the set of type-dependent operators are not finite because of the retrieval needs.

3.2 Interface Specification

We will adopt the following strategy to arrive at a finite set of type-dependent operators from the seemingly infinite set of type-dependent operators. We will divide the type-dependent operators into two sets. One set will consist of all the functions that are access method dependent. The other set will consist of a single abstract function which can be used for all retrieval operations for all data types.

These two sets are organized as two classes: KEY class and MATCH class. Both are base interface classes with pure abstract functions. In the base class these abstract functions are defined to provide only default error semantics. The implementation classes that derive from the base classes have to re-define these abstract functions for any meaningful operations. In the following sections, we specify these two classes and discuss them in detail.

3.2.1 KEY class

KEY class abstracts the concept of a "key" in a relational system. It provides a data structure for efficient storage of a key value of any type and defines functions that operate on this value. KEY class defines two sets of functions., one set as part of its public interface and the other as its protected interface. The functions in the public interface define all the functionality provided by the KEY class. These are concrete functions and are defined in the base class itself. Most of the functions in the public interface do only the error checking and call on the implementations of their counterparts in the protected interface for semantic processing.

The functions in the protected part of the KEY class are forwarding functions. This set of functions consist of the access method dependent type operators. These are defined to be no-ops in the base class. Providing an equivalent counter part in the public interface shifts the burden of error checking to the base class, simplifying the implementations.

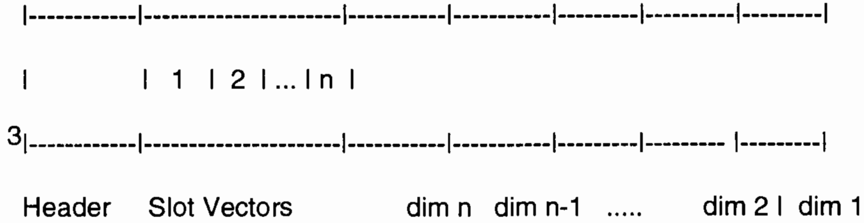
In the design of the KEY class, we have associated each "KEY" type with an unique identifier. It is expected that the system provides an unique identifier for both the built-in data types and for the user-defined data types. For the user-defined data types an unique identifier may be returned when the data types are registered with the system. Note that this unique id has been added only for operational convenience, for example, simplified error checking and it is not a design constraint.

Before specifying the KEY class, we will first present the data structure used for storing a key record. Section 3.2.1.2 defines the KEY class.

3.2.1.1 Key Record

The design of KEY record permits efficient storage for key of any type. Key can be single dimensional or multi-dimensional and can be of range or point values. The design doesn't permit a key to consist of range value in some dimensions and point value in other dimensions. All dimensions have to be of range or point values. The key record contains information only about the key and not about the information associated with a key.

A key record consists of three parts. The first part contains header information. The second part consists of slot vectors that has some information about each dimension. Number of slot vectors depend on the number of dimensions of the key. The third part contains the actual data for all dimensions. The following is a schematic representation of a key record.



The next section details the data structure. We define only the main components of the data structure here. The other structures and typedefs used here are defined in Appendix B.

```
// Header details of a key record
typedef struct KEY_HDR {
        TYPE_ID        type_id;     // Identifier for a type. This is the identifier
                                    // returned by the database system when a
                                    // new type is registered with the system.
        REC_NO         dim;         // Dimensions of a key
        BOOL           range;       // True if self refers to a range value
        KEY_LEN        data_len;    // Total length of the key record including headers
} KEY_HDR;

// Single entry in a dope vector
typedef struct KEY_SLOT {
        KEY_LEN        key_len,     // Length of each dimension
                       offset;      // Offset of a key at a particular dimension.
                                    // offset  from the beginning of data portion.
}KEY_SLOT;
```

[3] This structure is similar to the page structure used in Volcano[Graefe 1993a].

```
// A key record
typedef struct KEY_REC {
        KEY_HDR                 hdr;     // Header details
        union {
                KEY_SLOT        slot[1];  // var - length array
                char            data_[1];// var - length array
        } data;
} KEY_REC;
```

3.2.1.2 KEY Class Definition

This section defines the KEY class. Detailed definitions of the member functions are discussed in the next section.

```
class KEY {

   friend class MATCH;

   public :
        /*
         * Constructors, Destructors, and Validators
         */
        KEY();
        KEY(const KEY& key);
        KEY(const KEY_REC& rec);
       ~KEY();

        BOOL            is_valid() const ;
        Status          error_code() const;
        BOOL            same_types( const KEY& key ) const;

        /*
         * Operators
         */
        void            operator =   (  const KEY& key ) ;
```

```
BOOL        operator == (  const KEY& key ) const;
BOOL        operator !=  (  const KEY& key ) const;
BOOL        operator >= (  const KEY& key ) const;
BOOL        operator  >  (  const KEY& key ) const;
BOOL        operator  <  (  const KEY& key ) const;
BOOL        operator <= (  const KEY& key ) const;
int         compare( const KEY& key ) const;

BOOL        overlaps( const KEY& key ) const;
BOOL        overlaps( const KEY& low, const KEY& high ) const;

BOOL        contains( const KEY& key ) const;
BOOL        contains( const KEY& low, const KEY& high ) const;


/*
 * Mutators
*/
KEY         clone() const;
KEY_LEN     length() const;
KEY_REC      *key_record() const;
REC_NO      dim() const;
BOOL        range() const;
TYPE_ID     type() const;

// Mutators for each dimensions
KEY_LEN     key_length( REC_NO dim ) const;
KEY_LEN     offset( REC_NO dim ) const;
void*       key( REC_NO dim ) const;
int         hash() const;

/*
 * Modifiers
 */
Status      store_data( const KEY_REC& rec );
```

```
        KEY             build_region();
        KEY             build_region( const KEY& key );
        KEY             add_to_region( const KEY& key );
        KEY             subtract_from_region( const KEY& key );

protected :
        /*
         * All the protected are forwarded functions.
         * Sub classes have to re-define these
         */
        virtual int     hash_() const;
        virtual int     compare_( const KEY& key ) const;
        virtual BOOL    overlaps_( const KEY& key1, const KEY& key2, BOOL range ) const;
        virtual BOOL    contains_( const KEY& key1, const KEY& key2, BOOL range ) const;

        virtual Status  store_data_( const KEY_REC& rec );
        virtual KEY     build_region_( const KEY& key );
        virtual KEY     modify_region_( const KEY& key, BOOL add  );

        Status          status_;        // Maintains the state of self.
        KEY_REC         *rec_;          // Specifies self's value.
};
```

3.2.1.3 Detailed Specification and Member Function Definition

In this section, each member function is explained in detail along with the default implementation.

KEY::KEY()

Purpose	:	Default constructor. Produces an uninitialized KEY object.
Input	:	None
Output	:	Uninitialized KEY object
Error	:	None
Details	:	

The state of the produced object will be "notInitialized." This constructor doesn't allocate any storage and the member "rec_" is set to NULL. No valid operations will be possible in this state.

Implementation :

```
KEY::KEY() :
        status_ ( notInitialized ),
        rec_ ( NULL )
{ ;}
```

KEY::KEY(const KEY& key)

Purpose	:	Copy constructor. Produces a KEY object similar to that of the specified KEY object.
Input	:	Reference to a KEY object
Output	:	KEY object
Error	:	None
Details	:	

The constructed object will have the same state as that of the input object. This method allocates storage. The produced object will have it's own copy of "rec_".

Implementation :

```
KEY::KEY( const KEY& key) :
        status_( key.error_code() ),
        store_data_( (const)key.data() )
{;}
```

KEY::KEY(const KEY_REC& rec)

Purpose	:	Produces an initialized KEY object.
Input	:	Reference to a KEY_REC structure
Output	:	Initialized KEY object
Error	:	None
Details	:	

Constructor allocates storage and sets values for status_ and rec_. rec_ will have a value of "rec", and the status will indicate the success of the operation that stores "rec".

Implementation :

```
KEY::KEY( const KEY_REC& rec ) :
        status( store_data_( rec ) ){;}
```

KEY::~KEY()

Purpose : Destructor

Input : None

Output : None

Error : None

Details :

Destructor frees the storage.

```
KEY::~KEY()
{
        if( rec_ )
            delete rec_;
}
```

BOOL

KEY::is_valid() const

Purpose : Checks the validity of self

Input : None

Output : Boolean

Error : None

Details :

Returns TRUE if the object has a OK status

Implementation :

```
BOOL
KEY::is_valid() const{
        return ( status_ == OK );
}
```

Status

KEY::error_code() const

Purpose : To get the status of the object.

Input : None.

Output : Status.

Error : None.

Details :

Returns the status of self.

Implementation :

```
Status
KEY::error_code const{
	return status_;
}
```

BOOL

KEY::same_types(const KEY& key) const

Purpose : Determine whether two keys are of the same type.

Input : Reference to a KEY object.

Output : TRUE if self and key are of same type.

Error : None.

Details :

Return TRUE if both self and key are of same type.

Implementation :

```
BOOL KEY::same_type( const KEY& key ) const {
	return ( type() == key.type() );
}
```

KEY

KEY::clone() const

Purpose : Returns a copy of self.

Input : None.

Output : KEY object.

Error : None.

Details :

Returns a deep copy of self. Obeys value semantics.

Implementation :

```
KEY KEY::clone() const{
	return key;
}
```

KEY_REC *

KEY::key_record() const

Purpose : Provides access to the data value of self.

Input : None.

Output : Pointer to KEY_REC.

Error : Output will be a NULL reference.

Details :

Returns a pointer to the data value of key. If self is inval id returned pointer will be a NULL reference. It is the responsibility of the caller to check for the validity of the pointer.

Implementation :

```
KEY_REC
KEY::key_record() const
{
        if( is_valid() )
            return ( rec_ );
        return NULL;
}
```

KEY_LEN

KEY::length() const

Purpose : Gets the length of the data value of self.

Input : None.

Output : Length of the data part of self.

Error : Output will be 0 in case of errors.

Details :

Returns the length of self. The length will be 0 if self's status is invalid.

Implementation :

```
KEY_LEN
KEY::length() const
{
        if( is_valid() )
            return rec_->hdr.data_len;
        return 0;
}
```

REC_NO
KEY::dim() const

Purpose : Gets the dimensions of self.
Input : None.
Output : Number of dimensions of self.
Error : Output will be 0 in case of errors.
Details :
Invalid object will return a value of 0.

Implementation :

```
REC_NO
KEY::dim() const{
        if( !is_valid() )
            return 0;
        return ( rec_.hdr.dim );
}
```

BOOL
KEY::range() const

Purpose : Determine whether self represents a range value.
Input : None.
Output : TRUE if self represents a range value.
Error : Output will be FALSE in case of errors.

Implementation :

```
BOOL
KEY::range() const{
        if( !is_valid() )
            return FALSE;
        return ( rec_.hdr.range == TRUE );
}
```

KEY_LEN
KEY::key_length(REC_NO dim) const

Purpose : Get the length of the key at dimension "dim."

Input : Dimension of the key.
Output : Length of the key at dimension "dim."
Error : Output will be 0 in case of errors.
Implementation :

```
KEY_LEN
KEY::key_length( REC_NO dim ) const{
        if( !is_valid() )
            return 0;
        if( dim >= dim() || dim < 1 )
            return 0;
        return ( rec_->data.slot[dim - 1].key_len );
}
```

KEY_LEN
KEY::offset(REC_NO dim) const

Purpose : Get the offset of the key at dimension "dim".
Input : Dimension of the key.
Output : Offset of the key at dimension "dim."
Error : Output will be 0 in case of errors.
Implementation :

```
KEY_LEN
KEY::offset( REC_NO dim ) const{
        if( !is_valid() )
            return 0;
        if( dim >= di m() || dim < 1 )
            return 0;
        return ( rec_->data.slot[ dim -1].offset );
}
```

void *
KEY::key(REC_NO dim) const

Purpose : Get a pointer to the key at dimension "dim."
Input : Dimension of the key.
Output : Pointer to the key at dimension "dim."
Error : Output will be NULL in case of errors.

Implementation :

```
void *
KEY::key( REC_NO dim ) const {
        if( !is_valid() )
            return NULL;
        if( dim >= dim() || dim < 1 )
            return NULL:
        return ( (void *)(rec_->data + offset( dim ) ) );
}
```

TYPE_ID

KEY::type() const

Purpose : Return the type identifier of self.

Input : None.

Output : Type identifier of self.

Error : Output will be noType in case of errors.

Implementation :

```
TYPE_ID
KEY::type() const{
        if( !is_valid() )
            return noType;
        return rec_.hdr.type_id;
}
```

void

KEY::operator=(const KEY& key)

Purpose : Assignment operator.

Input : Reference to a KEY object.

Output : None.

Error : None.

Details :

Assigns key to self. Self will reflect key in all respects. Obeys value semantics.

Implementation :

```
void
KEY::operator=( const KEY& key ) :
```

```
        status_( key.error_code() ),
        store_data_( key ) {
}
```

BOOL

KEY::operator==(const KEY& key) const

Purpose : Check for equality.

Input : KEY object.

Output : TRUE if self and key are alike.

Error : Output will be FALSE if any of the object is invalid.

Implementation :

```
BOOL
KEY::operator==( const KEY& key ) const{
        return ( compare( key ) == 0 && compare( key ) != argErr );
}
```

BOOL

KEY::operator!=(const KEY& key) const

Purpose : Check for inequality.

Input : KEY object

Output : True if self and key are not alike.

Error : Output will be FALSE if any of the object is invalid.

Implementation :

```
BOOL
KEY::operator!=( const KEY& key ) const{
        return( compare( key ) != 0  && compare( key ) != argErr );
}
```

BOOL

KEY::operator>=(const KEY& key) const

Purpose : Check whether self is greater than or equal to key.

Input : KEY object.

Output : TRUE if self is greater than or equal to key.

Error : Output will be FALSE if any of the object is invalid.

Implementation :

```
BOOL
KEY::operator>=( const KEY& key ) const{
        return ( compare( key ) >= 0 && compare( key ) != argErr );
}
```

BOOL

KEY::operator>(const KEY& key) const

Purpose	:	Check whether self is greater than key.
Input	:	KEY object.
Output	:	TRUE if self is greater than key.
Error	:	Output will be FALSE if any of the object is invalid.
Implementation	:	

```
BOOL
KEY::operator>( const KEY& key ) const{
        return( compare ( key ) > 0 && compare( key ) != argErr );
}
```

BOOL

KEY::operator<(const KEY& key) const

Purpose	:	Check whether self is less than key.
Input	:	KEY object.
Output	:	TRUE if self is less than key.
Error	:	Output will be FALSE if any of the object is invalid.
Implementation	:	

```
BOOL
KEY::operator<( const KEY& key ) const{
        return( compare( key ) < 0 && compare( key ) != argErr );
}
```

BOOL

KEY::operator<=(const KEY& key) const

Purpose	:	Check whether self is less than or equal to key.
Input	:	KEY object.
Output	:	TRUE if self is less than or equal to key.
Error	:	Output will be FALSE if any of the object is invalid.

Implementation :

```
BOOL
KEY::operator<( const KEY& key ) const{
        return( compare( key ) <= 0 && compare( key ) != argErr );
}
```

int

compare(const KEY& key) const

Purpose : Determine whether key is greater, equal to or less than self.

Input : A reference to KEY object.

Output : -1 if self is less than key.
0 if both are equal.
1 if self is greater than key.

Error : Error state of the object.

Details :

In case of errors the error code will be returned as return value.

implementation :

```
int compare( const KEY& key ) const{
        if( !is_valid() )
            return argErr;
        if( !key.is_valid() )
            return argErr;
        if( !same_type( key ) )
            return argErr;
        return ( compare_( key ) );
}
```

Status

KEY::store_data(const KEY_REC& rec)

Purpose : Set the rec_ value of self.

Input : A reference to KEY_REC structure.

Output : Status of the operation.

Error : Check the state of self for error.
noMemory

Details :

Sets the value of rec_ to rec. Uses the virtual function "store_data_" to do the job. Storage will be allocated. Any previous reference of rec_ will be lost. Status of self will reflect any errors. This method will return error and not do any operation if the state of self is other than ok or notInitialized.

Implementation :

```
Status
KEY::store_data( const KEY_REC& rec ){
        if( !is_valid() && error_code != notInitialized )
            return status_;
        if( rec_ )
            delete rec_;
        return( store_data_( rec ) );
}
```

int

KEY::hash() const

Purpose : Returns a hash value for self.

Input : None.

Output : Hash value.

Error : Error state of self.

Details :

Uses the virtual method hash_() to do the job. Execution of this method will not affect the status of self. If self is invalid, self status will be returned as return value.

Implementation :

```
int
KEY::hash() const{
        if( !is_valid() )
            return (int)status_;
        return( hash_() );
}
```

BOOL

KEY::overlaps(const KEY& key) const

Purpose : Check whether self overlaps key.

Input : KEY object.

Output : TRUE if self overlaps key.

Error : Output will be FALSE in case of invalid objects.

Details :

Returns TRUE if self overlaps the region occupied by key. That is one of the points of self falls in the region bounded by key. This will return FALSE, if self or key is invalid or low doesn't contain a region.

Implementation :

```
BOOL
KEY::overlaps( const KEY& key ) const{
	if( !is_valid() || !key.is_valid() || !range() || !same_type(key) )
		return FALSE;
	return( overlaps_( key, NULL, FALSE ) );
}
```

BOOL

KEY::overlaps(const KEY& low, const KEY& high) const

Purpose : Check whether self overlaps the region bounded by low and high.

Input : Two KEY objects.

Output : TRUE if self overlaps the region bounded by low and high together.

Error : Output will be FALSE if any of the object is invalid.

Implementation :

```
BOOL KEY::overlaps( const KEY& low, const KEY& high ) const{
	if( !is_valid() || !low.is_valid() || !high.is_v alid() || !same_type(low) ||
	   !same_type(high) || low.dim() != high.dim() )
		return FALSE:
	return ( overlaps_( low, high, TRUE )  );
}
```

BOOL

KEY::contains(const KEY& key) const

Purpose : Check whether self is contained in key.

Input : KEY object.

Output : TRUE if self is contained in key.

Error : Output will be FALSE if any of the object is invalid.

Details :

Containment is TRUE when all points of self lies within the region bounded by key.

Implementation :

```
BOOL
KEY::contains( const KEY& key ) const{
        if( !is_valid() || !key.is_valid() || !same_type(key) || !key.range() )
            return FALSE;
        return ( contains_( key, NULL, FALSE ) );
}
```

BOOL

KEY::contains(const KEY& low, const KEY& high) const

Purpose : Check whether self is contained within the region bounded by low and high.

Input : Two KEY objects.

Output : TRUE if self is contained within the region bounded by high and low.

Error : Output will be FALSE if any of the object is invalid.

Implementation :

```
BOOL KEY::contains( const KEY& low, const KEY& high ) const{
        if( !is_valid() || !low.is_valid() || !high.is_valid() || !same_type(low) ||
           !same_type(high) || low.dim() != high.dim() )
            return FALSE;
        return( contains_( low, high, TRUE ) );
}
```

KEY

KEY::build_region()

Purpose : Build a region bounded by self.

Input : None.
Output : A KEY object that refers to a range value.
Error : Check the status of the returned object.
Details :

Builds a region that specifies the region bounded by self. The output will be similar to self if self already refers a region. The status of the returned object will reflect the success of the operation. State of self will remain unchanged.

Implementation :

```
KEY
KEY::build_region(){
        if( !is_valid() )
            return *this;
        return( build_region_( clone() );
}
```

KEY
KEY::build_region(const KEY& key)

Purpose : Build a region bounded by self and key.
Input : KEY object.
Output : KEY object that refers to a range value.
Error : Check the status of the returned object.
Details :

Builds a region that specifies the region bounded by self and key. This method will succeed if and only if both self and key refer to "non-regions" or "regions". This will fail if one of them is a region and the other is not. The state of self will remain unaffected in all cases.

Implementation :

```
KEY
KEY::build_region( const KEY& key ){
        if( !is_valid() )
            return *this;
        if( !key.is_valid() )
            return key;

        KEY key;
```

```
        if( dim() ! key.dim() || range() != key.range()  || ! same_type(key) ){
                    key.status_ = errorCreation;
            return key;
        }
        return( build_region_( key ) );
}
```

KEY

KEY::add_to_region(const KEY& key)

Purpose : Add the value of key to self.

Input : KEY object.

Output : KEY object.

Error : Check the status of the returned object.

Details :

This method returns a region that covers the region bounded by self and key. This method will fail if either self or key or both is not already a region. State of self will remain unaffected in all cases.

Implementation :

```
KEY KEY::add_to_region( const KEY& key ){
        if( !is_valid() )
            return *this;
        if( !key.is_valid() )
            return key;

        KEY k;
        if( !same_type(key) || !range() || !key.range() || dim() != key.dim() )
            return k;
        return( modify_region_( key, TRUE ) );
}
```

KEY

KEY::subtract_from_region(const KEY& key)

Purpose : Subtract the region occupied by key from self.

Input : KEY object.

Output : KEY object.

Error : Check the status of the returned object.

Details :

This method returns a region that covers the region bounded by self and not by key. This method will fail if either self or key or both is not already a region. State of self will remain unaffected in all cases.

Implementation :

```
KEY
KEY::subtract_from_region( const KEY& key ){
        if( !is_valid() )
            return *this;
        if( !key.is_valid() )
            return key;

        KEY k;
        if( !same_type(key) || !range() || !key.range() || dim() != key.dim() )
            k.status_ = errorCreation;
            return k;
        }
        return( modify_region_( key, FALSE ) );
}

/*
 * Following are forwarded functions.
 * These are defined to be no-ops in the base class.
 */
```

Status

KEY::store_data_(const KEY_REC& rec)

Purpose : Set the value of self to that of rec.

Input : Reference to a KEY_REC structure.

Output : Status of the operation.

Error : State of the object, noMemory.

Details :

This method will allocate necessary storage and will store 'rec' in the data member 'rec_'. For any errors this method will change the status of self to reflect the error status. Value returned will correspond to error status.

Default Implementation :

```
Status
KEY::store_data_( const KEY_REC& rec ){
        return noOp;
}
```

int

KEY::compare_(const KEY& key) const

Purpose	:	Compare self to key.
Input	:	Reference to a KEY object.
Output	:	-1 if self is less than key, 0 if self is equal to key, 1 if self is greater than key.
Error	:	Error state of the object, notSameTypes
Details	:	

It is expected that this method will provide a comparison logic for all types of keys.

Default Implementation :

```
int
KEY::compare_( const KEY& key ) const{
        return noOp;
}
```

int

KEY::hash_() const

Purpose	:	Return a suitable hash value of self.
Input	:	None.
Output	:	Hash value.
Error	:	Error state of the object.
Details	:	

It is up to the implementation to decide on the hashing technique.

Default Implementation :

```
int KEY::hash_() const{
        return noOp;
}
```

KEY

KEY::build_region_(const KEY& key)

Purpose : Build a region bounded by self and key.

Input : Reference to a KEY object.

Output : KEY

Error : Check the status of returned KEY object.

Details :

This method returns a region bounded by self and key.

Default Implementation :

```
KEY
KEY::build_region_( const KEY& key ){
	return key;
}
```

KEY

KEY::modify_region_(const KEY& key, BOOL add)

Purpose : Modify the region bounded by self by adding or subtracting the region bounded by key.

Input : Reference to a KEY object.

Output : KEY object.

Error :

Details :

If the flag add is TRUE the region returned will reflect the region bounded by self and key. If not, the resulting region will reflect the region covered by self and not by key.

Default Implementation :

```
KEY KEY::modify_region_( const KEY& key, BOOL add ){
	return key;
}
```

BOOL

KEY::overlaps_(const KEY& low, const KEY& high, BOOL range) const

Purpose : Check whether self overlaps the region bounded by self and low or low and high together.

Input : Reference to KEY objects.
Boolean condition.

Output : TRUE if self overlaps.

Error : Output will FALSE in case of invalid objects.

Details :

If range is TRUE, this method will return TRUE if self overlaps the region bounded by low and high. If not, "high" will be ignored and will return TRUE if self overlaps the region occupied by low.

Overlapping is TRUE if any of the points of self falls in the region bounded by low or low and high together.

Default Implementation :

```
BOOL
KEY::overlaps_( const KEY& low, const KEY& high, BOOL range ) const{
        return FALSE;
}
```

BOOL

KEY::contains_(const KEY& low, const KEY& high, BOOL range) const

Purpose : Determine whether self is contained in low or low and high together.

Input : Reference to two KEY objects.
Boolean condition

Output : TRUE if self is contained in.

Error : Output will be FALSE in case of invalid objects.

Details :

If "range" is TRUE this method will return TRUE if self contains the region bounded by low and high. If not, high will be ignored and containment will be checked with the region bounded by low. Contains is TRUE if and only if all the points of self fall with in the region covered by or low and high together.

Implementation :

```
BOOL  KEY::contains_( const KEY& low, const KEY& high, BOOL range ) const{
        return FALSE;
}
```

3.2.2 MATCH CLASS

Match class consists of a single match function defined in the public interface that can be used for different retrieval options possible for different key types. As in the case of KEY class, the public match function has an equivalent counter part in the protected interface of MATCH class. The public part does all the error checking and calls on the implementation of protected counter part for semantic processing. The protected interface is defined to be a pure abstract function and it is a no-op in the base class. Classes that derive from MATCH has to provide the necessary implementation. As MATCH class is defined to be a friend of KEY class, direct access to the data is available to this class.

3.2.2.1 Class Definition

```
class MATCH
{
  public :
        BOOL match(  const KEY& key1, const KEY& key2  ) const;
  protected :
        virtual BOOL match_(  const KEY& key1, const KEY& key2  ) const;
};
```

3.2.2.2 Detailed Definition

BOOL

MATCH::match(const KEY& key1, const KEY& key2) const;

Purpose : Determine whether key1 matches key2. Matching criteria depends on the

implementation of the protected function "match_()".

Input : Reference to KEY objects.

Output : Return,
TRUE if key1 matches key2,
else FALSE.

Error : Output will be FALSE in case of invalid objects.

Details :

This method determines whether one key matches another key. This method does only error checking and depends on the implementation for matching criteria. If this method returns TRUE, the record associated with that key will be returned. This method will return FALSE for all errors.

Implementation :

```
BOOL
match( const KEY& key1, const KEY& key2 ) const{
        if( !key1.is_valid() || !key2.is_valid() )
            return FALSE;
        if( ( key1.dim() != key2.dim() )  || ( key1.ran ge() != key2.range() ||
           !same_type( key ) )
            return FALSE;
        return match_( key1, key2 );
}
```

BOOL

MATCH::match_(const KEY& key1, const KEY& key2) const;

Purpose : Determine whether key1 matches key2.
The matching criteria is not specified. It is up to the implementation to incorporate appropriate matching criteria.
Typically the matching criteria will depend on the kind of operations possible for the key type.

Input : Reference to KEY objects.

Output : Return,
TRUE if key1 matches key2,
else FALSE.

Error : Output will be FALSE in case of invalid objects.

Details :

This method determines whether one key matches another key. The criteria for matching is up to the implementation. The access method implementor makes use of this method to determine whether a key in the index matches the search key. If this method returns TRUE, the record associated with that key will be returned. Hence, it is upto the user of the index to determine what matching criteria would satisfy his query and accordingly implement this method.

Default Implementation :

```
BOOL
match_( const KEY& key1, const KEY& key2 ) const{
        return FALSE;
}
```

3.3 Usage

All the type dependency needs of an index can be met using the KEY class and MATCH class member functions. In this section, we will show how an access method designer and the application programmer could make use of these functions to meet the various needs of different access methods and applications. We will start with building an implementation for both KEY and MATCH class.

3.3.1 Implementation Class - An Example

As noted earlier, the implementation of the base KEY and MATCH classes provide only default error semantics. For any meaningful operation, a KEY or a MATCH object has to be associated with a concrete implementation. In the typical design, the database system will provide implementation for all the built-in data types and the application programmer will provide implementations for all the user-defined data types. In this section, we will show how the designer of a database system or an application programmer could go about building an

implementation for a particular data type. In this example, we will build an implementation for a single dimensional "string" data type.

3.3.1.1 STRING_KEY Implementation

STRING_KEY class is a concrete class providing an implementation for base class KEY for single dimensional "string" data type. The class definition includes all forwarding functions of the base class and its own utility functions.

3.3.1.1.1 Class Definition

```
class STRING_KEY : public KEY {
   public :
         STRING_KEY();
         STRING_KEY( const STRING_KEY& key);
         STRING_KEY( const KEY_REC& rec );
         ~STRING_KEY()
   protected :
         int compare_( const KEY& key ) const;
         BOOL overlaps_( const KEY& key1, const KEY& key2, BOOL range ) const;
         BOOL contains_( const KEY& key1, const KEY& key2, BOOL range ) const;

         Status store_data_( const KEY_REC& rec );
         KEY build_region_( const KEY& key );
         KEY modify_region_( const KEY& key, BOOL add  );
   private :
         void copy( KEY& k, const KEY& k1, const KEY& k2, REC_NO i );
         void add(  KEY& k, const KEY& k1, const KEY& k2);
         void subtract(  KEY& k, const KEY& k1, const KEY& k2);
         BOOL between( char *k, char *low, char *high,
                          KEY_LEN k_len, KEY_LEN low_len, KEY_LEN high_len ) const;
};
```

3.3.1.1.2 Class Implementation

STRING_KEY::STRING_KEY

```
STRING_KEY::STRING_KEY()
	: KEY()
{;}
```

STRING_KEY::STRING_KEY

```
STRING_KEY::STRING_KEY(const STRING_KEY& key )
	: KEY( key )
{;}
```

STRING_KEY::STRING_KEY

```
STRING_KEY::STRING_KEY(const KEY_REC& rec )
	: KEY( rec )
{;}
```

STRING_KEY::~STRING_KEY

```
STRING_KEY::~STRING_KEY()
{;}
```

STRING_KEY::compare_() const

```
int
STRING_KEY::compare_( const KEY& key ) const
{
		return strncmp( (char *)key_record(1), (char *)key.key_record(1),
					max( key_length(1), key.key_length(1)  );
}// STRING_KEY::compare()
```

STRING_KEY::hash_() const

```
int STRING_KEY::hash_() const
{
		long ret;
		for( int i = 1, dims = dim() ; i <= dims; i++ ){
```

```
            char *key = (char *)key_record(i);
            for( j = 0; j <= key_length(i); j++ ){
                  ret += key[j];
            }
      }
      return ( ret / length() );
}// STRING_KEY::hash_()
```

STRING_KEY::store_data_()

```
Status
STRING_KEY::store_data_( const KEY_REC& rec )
{
      rec_ = (KEY_REC * )new [ rec.hdr.data_len ];
      if( !rec_ ){
         status_ = noMemory;
         return status_;
      }
      rec_->hdr.dim = rec.hdr.dim;
      rec_->hdr.range = rec.hdr.range;
      rec_->hdr.data_len = rec.hdr.data_len;

      // This is a generalized version. Can copy multiple dimensions of
      //  "string" types.
      for( int i = 0; i < rec_->hdr.dim ; i++ ){
            rec_->data.slot[i].key_len = rec.data.slot[i].key_len;
            rec_->data.slot[i].offset = rec.data.slot[i].offset;
            strncpy( rec_->data + rec_->data.slot[i].offset,
                     rec->data + rec->data.slot[i].offset,
                     key_length(i+1) );
      }
      return status_;
}// STRING_KEY::store_data_()
```

STRING_KEY::overlaps_() const

```
BOOL STRING_KEY::overlaps_( const KEY& low, const KEY& high, BOOL range ) const
{
        low.build_region( high );
        for( int i = 1, j = 1, dims = dim(); i <= dims; i++ ){
              if( between( (char *)key_record(i), (char *)low.key_record(i) ),
                           (char *)low.key_record(j+1), key_length(i), low.key_length(i),
                            low.key_length(i+1) ){
                    return TRUE;
              }
              if( (i % 2) == 0 )   j++;
        }// for loop
        return FALSE;
}//STRING_KEY::overlaps_()
```

STRING_KEY::contains_() const

```
BOOL
STRING_KEY::contains_( const KEY& key1, const KEY& key2, BOOL range ) const
{
        low.build_region( high );
        for( int i = 1,  j = 1,  dims = dim(); i < dims ; i++ ){
              if( between( (char *)key_record(i), (char *)low.key_record(i),
                           (char *)low.key_record(j+1), key_length(i),
                           low.key_length(i), low.key_length(i+1) ){
                    return TRUE;
             }
              if( (i % 2) == 0 )
                   j++;
        }
        return FALSE;
}// STRING_KEY::contains_();
```

STRING_KEY::build_region_()

```
KEY
STRING_KEY::build_region_( const KEY& key )
```

```
{
        KEY k;
        k.rec_  =( KEY_REC *) new char[ length() ;
        offset = length();
        for( int i = 1, dims = dim(); i <= dims ; i += 2 ){
                if( strncmp( (char *)key_record(i), key.key_record(i),
                                max( key_length(i), key.key_length(i) )
                                <= 0 ){
                        copy( k, *this, key, i );
                }
                else{
                        copy( k, key, *this, i );
                }
        }// for loop
        return k;
}//STRING_KEY::build_region()
```

STRING_KEY::modify_region_()

```
KEY
STRING_KEY::modify_region_( const KEY& key, BOOL add  ){
        int length = max( length(), key.length() );
        k.rec_ = (KEY_REC *)new[ length ];
        k.rec_->hdr.dim = dim();
        k.rec_->hdr.range = TRUE;
        k.rec_->hdr.length = length;
        int offset = length - sizeof( KEY_HDR );
        if ( add ){
            add( &k, *this, key );
        }
        else
            subtract( &k, *this, key );
        }
        return k;
}//STRING_KEY::modify_region_()
```

STRING_KEY::copy()

```
void
STRING_KEY::copy( KEY& r, const KEY& k1, const KEY& k2, int i )
{
        int offset = r.length() - sizeof( KEY_HDR );
        r.rec_->slot[i].keylen = k1.key_length(i);
        offset -= r.key_length(i) ;
        r.rec_->slot[i].offset  = offset ;
        strncpy( r.rec_->data. + r.offset(i), k1.key_record(i),  r.key_length(i) );

        if( key2 ){
            r.rec_->slot[i+1].keylen = k2.key_length(i+1);
            offset -= r.key_length(i+1) ;
            r.rec_->slot[i+1].offset  = offset ;
            strncpy( r->data. + r.offset(i+1),  k2.key_record(i+1), r.key_lenth(i+1) );
        }
}//STRING_KEY::copy()
```

STRING_KEY::add()

```
void
STRING_KEY::add( KEY& k, const KEY& k1, const KEY& k2 ){
        for( int i = 1, dims = dim(); i <= dims ; i++ ){
                // Low dimension
                if ( i % 2 != 0 ){
                {
                    if( strncmp( (char *)k1.key_record(i), (char *)k2.key_record(i),
                                max( k1.key_length(i), k2.key_length(i) ) <= 0 ){
                        copy( k, *k1, NULL, i-1 );
                    }
                    else {
                        copy( k, k2, NULL, i-1 );
                    }
                }
                else{ // Inverse is TRUE for higher level ranges.
                    if( strncmp( (char *)k1.key_record(i), (char *)k2.key_record(i),
```

```
                        max( k1.key_length(i), k2.key_length(i) ) <= 0 ){
                    copy( k, k2, NULL, i-1 );
                }
                else {
                    copy( k, k1, NULL, i-1 );
                }
            }
        }// For loop
}// STRING_KEY::add
```

STRING_KEY::subtract()

```
void STRING_KEY::subtract( KEY& k, const KEY& k1, const KEY& k2 ){
    for( int low = 1, high = 2, dims = dim(); high <= dims; low++, high++ ){

        /*
         * If the region to be subtracted falls within - adjust the
         * current position. Else leave it as it is.
         */
        if( between( k2.key_record(low), k1.key_record(low), k1.key_record(high)
                    k2.key_length(low), k1.key_length(low), k2.key_length(low) ){
            copy( k, k2, NULL, low )
        }
        else{ copy( k, k1, NULL, low )
        }
    }
}// STRING_KEY::subtract
```

STRING_KEY::between()

```
void
STRING_KEY::between( char *k, char *low, char *high, KEY_LEN k_len,
                    KEY_LEN low_len, KEY_LEN high_len ){

    if( ( strncmp( k, low, min( k_len, low_len ) ) > 0 ) &&
      ( strncmp( k, high, min( k_len, high_len )) < 0 ) ){
        return TRUE;
```

```
        }
        return FALSE;
}//STRING_KEY::between();
```

3.3.1.2 STRING_MATCH Implementation

The last section provided an implementation for class KEY. Here we will provide an example implementation for MATCH class. While there is only one implementation for a KEY class for any single data type, there may be many implementations for a MATCH class. This is because, as noted earlier, there may be many retrieval operations possible for a data type and each retrieval operator requires an unique MATCH class implementation.

The following implementation defines an exact match operation.

3.3.1.2.1 Class Definition

```
Class STRING_EXACT_MATCH : public MATCH {
        BOOL match( const KEY& key1, const KEY& key2 ) const;
 };
```

3.3.1.2.2 Class Implementation

```
BOOL
STRING_EXACT_MATCH::match( const KEY& key1, const KEY& key2 ) const
{
        for( int i = 1, dims = key1.dim(); i <= dims; i++ ){
                if( strncmp ( key1.key_record(i), key2.key_record(i),
                        min( key1.key_length(i), key2.key_length(i) )
                         != 0 ){
                return FALSE;
           }
        }
        return TRUE;
}//::match()
```

3.3.2 Application Design

Having provided an implementation for KEY and MATCH classes, applications can call on the index interfaces with the appropriate "KEY" type. For example, to insert a key of "STRING_KEY" type in an existing index an application would typically proceed as follows.

```
// Create a key of type STRING_KEY
STRING_KEY string_key( key_rec );

//This is an index interface call. Refer chapter 4 for details.
insert( open_id, string_key, record, rec_len );
```

3.3.3 Index Interfaces Implementation

In this section, we will discuss how an access method designer will make use of these "KEY" and "MATCH" classes to implement a type-independent index.

The access method designer will meet all the type-dependent needs using only the functionality provided by the base interface classes "KEY" and "MATCH". The implementation of these classes(for example, STRING_KEY and STRING_MATCH) are not the concern of access method designer. The functionality provided by the KEY class are to be used for any index construction, loading, traversal and maintenance algorithms. The retrieval operations are to be performed using only the functionality provided by the MATCH class. In a retrieval operation, the match operator of the MATCH class will be applied only to the data records and not for traversal within the index to reach the data record.

3.4 Discussion

In this section, we will discuss how our design supports optimized query processing for user-defined data types, analyze the completeness of the set and try to identify possible weaknesses of the design.

Consider a query seeking to retrieve all the boxes that overlap the unit square, from a multi-dimensional index storing 2-dimensional boxes. If the overlap operator is not directly supported by the access method, the query can be processed only by retrieving all the boxes and then comparing to see whether a retrieved box overlaps the unit square or not. Note that this comparison has to be done outside the index as the index doesn't have direct support for the overlap operator. This will typically be the case for all user-defined data types. In our design, the retrieval is done using a user-supplied implementation of MATCH class. By instantiating the match operator with a suitable implementation class, one could reject all the unwanted records within a single index call. That means, an index call will never produce an unwanted record, resulting in one hundred percent success rate. This will result in fewer index calls. The fewer index calls may result in reduced locking overhead.

Let us now analyze the completeness of our set. We have arrived at this set of type-dependent operators working with a finite set of access methods. The only reason why we have to work with a finite set of access methods, is the existence of access method specific type-dependent operators. But, as noted earlier, we have not come across any access method, other than the hash method that requires a specific type-dependent operator. Though it is not guaranteed, we believe the interfaces will work for other access methods as well, other than those listed in appendix A.

We have been discussing the advantages of our design. In the following, we try to identify weaknesses of the design. The obvious disadvantage is the instantian overhead. Any retrieval call has to instantiate the MATCH class before a retrieval can be done. But given the advantages of type independence and optimized query processing for all data types, this overhead must be overlooked. It is also not possible to avoid this overhead in any effort towards type-independent access methods, irrespective of any organization one may choose to employ.

4 UPPER INTERFACES

We gave a brief introduction to upper interfaces in Chapter 2. In this chapter, we identify and specify a complete set of upper interfaces that would work for all the access methods specified in Appendix A. We have divided upper interfaces into three groups: DDL and DML interfaces, Optimizer Support Interfaces, and finally Log Support Interfaces. DDL and DML interfaces consist of the standard set of functions for performing DDL and DML operations. These are covered in Section 4.1. Optimizer and Log support interfaces are our addition to index functionality. Section 4.2 discusses optimizer support interfaces and Section 4.3 discusses the log support interfaces.

The data structures used in the definition of the upper interfaces are defined in Appendix B. All these interfaces return a "okay" status on success. In case of errors they return the appropriate error code. Appendix B details all the error codes.

4.1 Data Definition and Data Manipulation Language Interfaces

DDL and DML interfaces provide a standard set of functions for creating, altering, deleting indices and for inserting, updating and deleting records. The set also includes functions for bulk loading, lookup, and range retrievals. Most of the interfaces are designed as iterators with open, next, and close calls. The following section specifies the interfaces.

4.1.1 Interface specification

Create

Status create

(TRANSACTION_ID tid, const CREATE_PARAM& create_param, FILE_ID& file_id);

Purpose	:	Create a new index with the specified parameters.
Input	:	tid - Transaction identifier.
		create_param - Specifies all create parameters.
Output	:	file_id - Permanent file identifier for this index -

A handle for this index that can be used for open and destroy operations.

Details :

This method creates a new index by creating a new file, using the "create_file" interface of the disk manager. Create_param contains the "file_param" for the "create_file" interface. The "index_type" field of create_param determines the type of index to be created. This method also creates a new index header entry in the index catalog using the "write_catalog" interface of the catalog manager. Part of the details for the index header entry are given by the "index_param" field of "create_param".

The "file_id" returned by this method is the same as the "file_id" returned by the "create_file" interface. This file identifier will remain valid until the index is destroyed. The file identifier serves as the handle for any further operations on this index.

Destroy

Status destroy(TRANSACTION_ID tid, FILE_ID file_id);

Purpose : Remove an existing index from the system.

Input : tid - transaction identifier.
file_id - Permanent id of the index returned by a create call.

Output : Status code.

Details :

This method deletes an index from the database by deleting the index header entry in the index catalog and by freeing all the relevant data blocks. This method makes use of the "delete_file" interface of disk manager for freeing the data blocks.

Open

Status open
(TRANSACTION_ID tid, FILE_ID file_id, MODE mode, OPEN_FILE_ID& open_id);

Purpose : Open an existing index for reading, writing or both.

Input : tid - Transaction identifier.
file_id - Permanent file id returned by a create call.
mode - Indicates the opening mode for the index.
No other operations other than those for which the file is opened will be allowed.

Output : open_id - An handle for the open file that is to be

used for subsequent read, write, and close operations.

Details :

This method opens the index using the "open_file" interface of disk manager. The "open_id" returned is the same as the "open_id" returned by the "open_file" interface. Before returning, this method will associate this open_id with the index header entry of this index.

The "open_id" returned will remain valid till the index is closed with an explicit close call. An index can be opened any number of times and each open call will return an unique "open_id".

Close

Status close(OPEN_FILE_ID open_id);

Purpose : Close an open index.

Input : open_id - File handle returned by an open call.

Output : Status code.

Details :

This method calls on the "close_file" interface of disk manager to close the file. The "open_id" will be disassociated from this index making it available for other open calls. Closing an index that was never opened will not result in error.

Insert

Status insert

(OPEN_FILE_ID open_id, const KEY& key, const RECORD record, REC_LEN rec_len);

Purpose : Insert a single record in an open index, opened for writing.

Input : open_id - Valid file handle returned by an open call.
key - Key of the record to be inserted.
record - Pointer to the record to be inserted.
rec_len - Length of the record to be inserted.

Output : Status code.

Details :

The new record to be inserted must meet the integrity constraints specified. If the index permits duplicate records, insertion of any new record associated with an already existing key will always be appended to the end of the records associated with the same key. If duplicates are not supported an error will be returned.

This method will modify the index header record to reflect the current status of the index.

Delete

Status delete(OPEN_FILE_ID open_id, const KEY& key);

Purpose	:	Delete an existing record from an open index, using a key value.
Input	:	open_id - Valid file handle returned by an open call. key - Key of the record to be deleted.
Output	:	Status code.
Details	:	

This method deletes an existing record from the index. The value of "key" will be used for finding and deleting the record. If the index contains duplicate records the first record matching the "key" will be deleted. The other records associated with the same key will remain unaffected. All the records pertaining to a key can be deleted by successive calls until the Status returned is "recordDoesNotExist."

Whether the deleted space is immediately reclaimed or not would vary from implementation to implementation. We do not specify any constraints here and it is left to the implementation. This method will modify the index header record to reflect the current status of the index.

Status delete(OPEN_FILE_ID open_id, REC_ID rec_id);

Purpose	:	Delete an existing record using a record identifier.
Input	:	open_id - Valid file handle returned by an open call. rec_id - Identifier of the record to be deleted.
Output	:	Status code.
Details	:	

Deletes using this method are expected to be faster than the deletes using a "key" value as this method doesn't do a search to find a record. Apart from efficiency considerations, this method is the only way a "particular" record can be deleted in case of duplicate records. This method will modify the index header record to reflect the current status of the index. All other constraints listed for the delete method with the "KEY" option holds for this delete method as well.

Modify

Status modify

(OPEN_FILE_ID open_id, const KEY& key, const RECORD new_record,
REC_LEN new_rec_len);

Purpose	:	Modify the data portion of an existing record from an open index, opened for writing.
Input	:	open_id - Valid file handle returned by an open call. key - Key of the record to be modified. new_record - New data portion of the record. new_rec_len - Length of the new record.
Output	:	Status code.
Details	:	

This method modifies the data portion of an existing record. Modifications involving changes to the key value can not be done through this method. Key value changes can be achieved through a delete followed by an insert call.

This method uses the value of "key" for finding the record to be modified. In case of duplicate records, the first record that matches the key will be modified and the call will return. Use the modify method with the "rec_id" option for modifying a specific record. Scan can be used to get rec_id details.

In cases where the modified record length is less than the existing record, the reclamation of space is left to the implementation. It is also left to the implementation to deal appropriately, if the modified record length is more than the existing record.

This method will modify the index header record to reflect the current status of the index.

Status modify

(OPEN_FILE_ID open_id, REC_ID rec_id, const RECORD new_record,
REC_LEN new_rec_len);

Purpose	:	Modify the data portion of an existing record from an open index, opened for writing.
Input	:	open_id - Valid file handle returned by an open call. rec_id - Identifier of the record to be modified. new_record - New data portion of the record. new_rec_len - Length of the new record.
Output	:	Status code.
Details	:	

This method is similar to the modify method with the "KEY" option in all respects except that this uses a "rec_id" to find the record to be modified. Use of this method is recommended, if a specific record is to be modified. Scan can be used to get rec_id details. Modification using this method may be faster than modifying using a key value.

Lookup

Status lookup

(OPEN_FILE_ID open_id, const MATCH& match, const KEY& key, RECORD& record, REC_LEN& rec_len, REC_ID& rec_id);

Purpose	:	Retrieve the record associated with the specified key from an open index.
Input	:	match - Reference to the MATCH library implementing the match criteria. open_id - Valid file handle returned by an open call. key - Key value of the record required.
Output	:	record - Pointer to the data portion of the record. rec_len - Length of the record retrieved. rec_id - Record identifier of the record retrieved.
Details	:	

This method looks up in the index and returns the record associated with the specified key value. The type of association is determined by the specified "MATCH" criteria and not by the implementation.

In case a matching record is found, the output parameters record, rec_len and rec_id will be set to reflect the appropriate values. This method doesn't make any assumption about the type of the record and it is up to the caller to appropriately interpret it.

In case of duplicate records, the first record that matches the key will be returned. Scan can be used if details of all the records associated with a key are required.

The "record pointer" returned points to the buffer page containing the record and no copying is done. This pointer is guranteed to be valid untill a next call on this file. The next call can be any index interface call that uses the same "open_id" used in this lookup call.

Open_scan

Status open_scan

(TRANSACTION_ID tid, FILE_ID file_id, const MATCH& match, const KEY& interval, BOOL forward, OPEN_SCAN_ID& open_scan_id);

Purpose	:	Open an existing index for a range search.
Input	:	tid - Transaction id. file_id - Permanent file id returned by a create call. match - Pointer to the MATCH class library specifying the criteria for matching. interval - Range interval for the scan. This represents the region covered between low and high scan ends inclusive of low and high values. forward - If TRUE records will be returned in the increasing order of key value.
Output	:	open_scan_id - Handle for further scan operations.
Details	:	

This method opens an index for doing a range search. The validity of the interval is determined by existence of atleast one record falling in the specified range. The "open_scan_id" returned can only be used for subsequent scanning operations, such as "next_scan" and / or "close_scan". No other operations can be carried out using this "open_scan_id".

The boolean argument "forward" determines whether the scan is to be done in the increasing order(from low to high) or in the decreasing order. This however may not make sense for all access methods. For example, in a multi-dimensional index like R-tree the next physical record may not be the next logical one in terms of increasing or decreasing order. On performance reasons, it would make perfect sense to return the next physical record if it falls in the scan region instead of finding and retrieving the next record that logically falls in order. The order in which records are returned will only apply to access methods that truly support ordering.

This method makes use of the "open_file" interface of the disk manager to physically open the index. The "open_scan_id" returned will pertain to a new entry created by this method in the "open_scan_table". This new entry in the "open_scan_table" will associate with it, the "open_id", and the index header record pertaining to this index. It is the responsibility of this method to fill up the other fields of "open_scan_table" and to do whatever is necessary to facilitate a "next_scan" call. A call to the "next_scan" returns the next record that falls within the scan range.

Constraints relating to allowing multiple "open_scans" are left to the implementation. The design of "open_scan_table" permits multiple scans to run concurrently on the same index.

Next_scan

Status next_scan

(OPEN_SCAN_ID open_scan_id, RECORD& record, REC_LEN& rec_len,
REC_ID& rec_id);

Purpose : Retrieve the next record for the current scan.

Input : open_scan_id - Handle returned by an open_scan call.

Output : record - Pointer to the record retrieved.
rec_len - Length of the record.
rec_id - Identifier of the retrieved record.

Details :

This method returns the next record in the current scan. An "endOfFetch" status will be returned on reaching end of scan. End of scan is marked by reaching the "scan_end_interval" or end of file.

This method maintains the internal state of the scan across invocations. The state is maintained until a "close_scan" is called. The "SCAN_LIST" field of the "open_scan_table" serves to maintain the internal state. Maintaining the state of the scan prevents multiple traversal of the index, thereby improving the scan efficiency.

The “record” pointer returned points directly into the buffer page that contains the record and no copying is done for performance reasons. This pointer is guranteed to be valid until a next call that uses the same “open_scan_id” used in this call.

Close_scan

Status close_scan(OPEN_SCAN_ID open_scan_id);

Purpose : Close a currently running scan.

Input : open_scan_id - Id returned by an open_scan call.

Details :

Closing a scan is marked by closing the index, deleting the entry in the "open_scan_table", and by releasing the "open_scan_id" and "open_id". The index will be closed by calling on the "close_file" interface of the disk manager.

Any scan initiated by a call to "open_scan" has to be explicitly closed. There is no automatic closure of scans.

Open_load

Status open_load

(TRANSACTION_ID tid, FILE_ID file_id, OPEN_LOAD_ID& open_load_id);

Purpose	:	Open an index for bulk loading operations.
Input	:	tid - Transaction identifier.
		file_id - Permanent file identifier returned by a create call.
Output	:	open_load_id - Handle for further loading operations.
Details	:	

This method opens an existing index for bulk loading operations. The "open_load_id" returned by this method can only be used for subsequent "next_load" and "close_load" operations. No other operations can be carried out using the "open_load_id".

Similar procedure followed by the "open" method will be followed by this "open_load" method for physically opening the index. In other words, this method will make use of the "open_file" interface of the disk manager for opening the index. The "open_id" returned by the "open_file" interface will be associated with the index header record pertaining to this index. The major difference between "open" and "open_load" method is the "open_load_id" returned is not the same as "open_id". This method creates a separate entry in the "open_load_table". The "open_load_id" returned pertains to this entry and it is associated with the "open_id".

This extra effort is required for the following reasons. First, it is not possible to open an index multiple times simultaneously for loading. Second, no other data manipulations can be made possible during a loading operation. In addition to this, each access method may also impose some more loading constraints. For example, a B-tree index may not allow loading in an existing index. On the other hand, in an access method such as R-tree, loading may be carried out using an insert algorithm, in which case loading in an existing index can be allowed.

For the above reasons, we will not specify here the required conditions for a successful open_load call. The success of an open_load call will differ from access method to access method and will depend on the implementation. Implementations may choose to disallow loading depending on the state of the index.

This method also sets up the other fields of "open_load_table" which are used to maintain the internal state of the load. It is important to maintain the internal state of load, as loading can be spread over many invocations of "next_load".

Next_load

Status next_load
(OPEN_LOAD_ID open_load_id, const KEY& key, const RECORD record,
REC_LEN rec_len);

Purpose	:	Load a single record in the index.
Input	:	open_load_id - Load handle returned by an open_load call.
		key - Key of the record to be loaded.
		record - Data portion of the record.
		rec_len - Length of the record portion.
Output	:	Status code.
Details	:	

This method loads a single record in an index opened for loading. It is the responsibility of this method to maintain the internal state of loading. This will be done by updating the appropriate fields of "open_load_table". This method will modify the index header record to reflect the current status of the index.

Close_load

STATUS close_load(OPEN_LOAD_ID open_load_id);

Purpose	:	Close loading.
Input	:	open_load_id - Handle for loading returned by an open_load call.
Output	:	Status code.
Details	:	

Close load closes the index opened for loading with an explicit "open_load_call". As in the case of "close" interface, this method will make use of the "close_file" interface of disk manager to physically close the index. The "open_load_id" and the "open_id" will be disassociated from the index and will be made available for further use.

4.1.2 Discussion

The DDL and DML interfaces have been designed to provide ease of operation and complete functionality. The interface design works for all access methods. However, it is possible that not all access methods support all these interfaces. For example, the original

paper on R-tree does not talk about bulk loading on R-trees. In cases like this, each access method implementation may either return a "notSupported" status or can indirectly map one method to another. As an example, loading can always be implemented using an insert method, though it will not be as efficient as in the case of direct algorithmic support for loading.

The decision to maintain separate open tables and provide unique handles for loading and scanning is based on the following reasons. Many access methods that have direct support for loading may not allow other data manipulation operations in the index during loading. Also, loading and scanning have to maintain the current state, as these both can be spread over many invocations of next calls. Maintaining the internal state improves the efficiency as it reduces the traversal of the index for each call. It becomes easier to provide these facilities using unique handles, instead of using the same handle for all data manipulation operations.

4.2 Optimizer Support Interfaces

In this section, we specify optimizer interfaces. Optimizer interfaces consist of cost and selectivity functions that support the optimizer in deciding on the most economical access path for evaluating a query. Optimizer interfaces are not one of the standard set of interfaces, one provides as part of index functionality. In our desire to provide a complete set of index interfaces, we try to identify a set of functions that an index on its own can provide to an optimizer.

4.2.1 Design Considerations

A basic query block is represented by a SELECT list, a FROM list, and a WHERE tree, containing respectively the list of items to be retrieved, the relation(s) referenced, and

the Boolean combination of predicates specified by the user. A single SQL statement may have many query blocks because a predicate may have one operand, which is itself a query.

In the conventional query processing model, an optimizer, a separate software module that is not part of the index code, determines the most economical access path based on the current status of referenced relations. It is obvious that the optimizer has to have access to the meta-data of all the referenced relations or all the relations in the database, to make such a decision.

In our case, we are working in the context of a single relation. An index has access to its own meta data only and does not have access to another index's meta data. With that constraint, we will try to identify some interfaces that will make the job of an optimizer easier. Our intention here is to provide the optimizer with all the information, an optimizer may ever want from an index.

In the next section, we specify the optimizer support interfaces. The discussions following that sketches a query processing model using these interfaces.

4.2.2 Interface Specification

Statistics

Status statistics(FILE_ID file_id, INDEX_STATS& index_stats);

Purpose	:	To get the statistical information about an existing index.
Input	:	file_id - Permanent id of the existing index
Output	:	index_stats - Pointer to the structure containing the index statistics.
Error	:	Status code.
Details	:	

This method returns a pointer to an "INDEX_STATS" data structure that contains the statistical information about an index.

Cost_and_selectivity

Status cost_and_selectivity

(FILE_ID file_id, PREDICATE &predicate, REC_NO& selectivity, COST& cost);

Purpose	:	Get the selectivity and cost of accessing this index for satisfying the given predicate.
Input	:	file_id - Id of the existing index. predicate - Pointer to the structure containing the predicate details.
Output	:	Selectivity and Cost will be set.
Details	:	

This method returns selectivity and cost details for the given predicate. Selectivity is the fraction of records expected to satisfy a given predicate. A predicate is considered to be valid if and only if it can be handled within the context of this index. For example, a predicate having a reference to a column of another index will be considered to be invalid.

We have not specified here the unit of cost. There are different ways to model the unit of cost. Some systems may find the number of I/O's to be the unit of cost. For some, the time spent may also be a factor in determining the unit of cost. We feel it makes more sense to leave it to the implementation to incorporate the appropriate cost model as they deem fit.

4.2.3 Discussion

The statistics method provides access to some statistical information about an index to the caller. The cost and selectivity provide cost and selectivity details for evaluating a predicate using this index. In this section, we will try and analyze the changes these interfaces bring to the query processing model and the additional benefits they provide.

Note that whether these methods exist or not, a typical optimizer would in any case work out these details for all the relations a query predicate may reference. By making "statistics()," "cost_and_selectivity()" part of the index interfaces, we have shifted a portion of the job of query optimization from the optimizer to every access method in the system. Abstraction and modularity are the foremost benefits of this design. An optimizer could be designed and developed without being aware of where the meta data of an index are stored or how they are stored. These information could be obtained by simple calls to the index

manager. That means the design of the optimizer need not be tied to the storage structure of a system.

The meta data of an index is part of that index and can be accessed only through its interfaces. It provides freedom to the access method designer in choosing where to keep this information, and what kind of structure to use. A decision on these issues can be made on how fast and frequently the information can be kept updated to reflect the most recent state of an index. The design and implementation of selectivity and cost methods have become access method specific. While we have not analyzed the immediate benefits of these, it looks as though more accurate information could be provided to the optimizer than that would be possible by following a generic cost model for the entire system.

Inclusion of optimizer support interfaces as part of index functionality doesn't result in major changes to the existing query processing model. The optimizer would start the optimization process, evaluating the predicates. If the predicates are of the form "Column(s) Comparison-Operator Value(s)," they can be directly passed to the appropriate indices for evaluation. For complex predicates involving many boolean combinations and referencing columns from many indices, the optimizer has to break down the predicates to simple predicates that can be handled by a single index. The predicates that can be handled by a single index are limited to the predicates that are of the form "Column(s) Comparison-Operator Value(s)" where the column or columns are all part of the same index. For each simple predicate, calls have to be made to the appropriate index for any cost or selectivity information. The final cost for the entire predicate has to worked out by the optimizer, using the details available for each simple predicate. The same model of processing will hold good for complex predicates of any type, including queries involving sub-queries and co-related queries.

4.3 Log Support Interfaces

In Section 5.5, we discuss in detail index lower interfaces for Log-Recovery sub-system and issues related to physical logging and logical logging. In the case of logical logging, we will argue in Section 5.5 that each access method provides a redo and undo procedure for each event, to help the log manager in the recovery phase. In this section, we will design interfaces for these two procedures to support the log manager during crash recovery.

Most of the data definition and data manipulation operations result in changes to the database. Section 5.5 describes all the events that result in changes to the database. For example an insert or a delete call changes the data content of an index. Apart from the changes to the data pages, an insert or a delete might also affect the structure of the index (As an example, the internal nodes in a B-tree index might also get affected). Note that we have taken insert or delete as an example only. Other events may also affect the structure of the index. The degree of changes to the storage structure of the index depend on the state of the index and on the algorithms that do the insert or delete. In other words, these are access method specific. We believe it would be more appropriate to make each access method provide interfaces for redoing or undoing an event and let the log manager of the database system call on these interfaces, when ever an event is to be re-done or un-done. The next section defines these interfaces.

4.3.1 Interface Specification

The interfaces we design here pertain only to logical logging. In our opinion physical logging can be carried out transparent to the index. For detailed discussions refer to section 5.5.

Redo

Status redo

(FILE_ID file_id, EVENT_ID event_id, RECORD log_record, REC_LEN log_rec_len);

Purpose	:	Redo the event.
Input	:	file_id - Permanent file identifier. event_id - Type of event for which redo is required. log_record - This is the information provided by the index to the log manager, when the event was logged. log_rec_len - Length of the log record.
Output	:	Status code.
Details	:	

This method idempotently re-does the operation indicated by "event_id". The format and content of the "log_record" should remain the same as it was, when that log record was created by this index. The implementation of this method is specific to the access method. It is also up to the access method to store appropriate information in the log record, during the creation of log record, to carry out a successful redo.

Undo

Status undo

(FILE_ID file_id, EVENT_ID event_id, RECORD log_record, REC_LEN log_rec_len);

Purpose	:	Undo the event and restore the state of the index.
Input	:	file_id - Permanent file identifier. event_id - Type of event for which redo is required. log_record - This is the information provided by the index to the log manager, when the event was logged. log_rec_len - Length of the log record.
Output	:	Status code.
Details	:	

This method idempotently un-does the event and restores the index to the previous state that existed before the occurrence of the event that is being re-done. The format and content of the "log_record" should remain the same as it was, when that log record was

created by this index. Implementation of this method is access method specific. It is also up to the access method to store appropriate information in the log record, during the creation of log record, to carry out a successful undo.

4.3.2 Discussion

In the above two interfaces, we have specified two constraints: first, the idempotency of the operations and second, the immutability of the log record. The implementation of these two functions are left to the access method designer as these interfaces are also access method specific as the other upper interfaces. As we will see in section 5.5, it is the responsibility of the access method to create appropriate log records for each event. The access method designer should take care to store all relevant information necessary to perform a redo or undo. As this log record will be interpreted only by the access method that created it, it is the choice of the access method designer to have his own structure for the log record.

The design of the log-support interfaces reduces the burden on the log-recovery subsystem. During the recovery phase, the log manager has to only call on the redo or undo interface of the appropriate index depending on whether a transaction has committed or aborted. Since the access methods create their own log records using the lower log manager interface, the log manager's responsibility is limited to associating these log records with the relevant transaction, and storing and retrieving these log records as and when needed.

In this chapter, we discussed upper interfaces. The inclusion of support for the optimizer and the log-recovery sub-system completes the abstraction of an index in the system promoting the concept of an index ADT.

In the next chapter, we discuss the lower interfaces that lead to development of system-independent access methods.

5 LOWER INTERFACES

An index in a database system interacts with many of the software modules in the system to accomplish the index file management functions, memory management functions, and some database system specific functions such as logging and recovery. All these functions are database-system dependent and we call these system-dependent software modules, the lower interfaces. In this chapter, we deal with the problem of interfacing the index code with the database-system dependent software modules. We try to identify a set of software modules an index will depend on, in a database system, and design interfaces for all the identified modules. Our success here will depend on identifying the right set of software modules that will facilitate development of system-independent index implementations .

This chapter is organized is as follows. In the next section, we detail the design considerations and give a brief road map of how we are going to design generic lower interfaces. In the rest of the chapter, we will discuss and specify interfaces for each of these software modules in detail.

5.1 Design Considerations

The set of lower interfaces an index in a database system interacts with, depend on the functionality supported by the database system. The functionalities a database management system support vary from system to system. A sophisticated system may include support for partitioning, clustering, logical and physical logging, crash-recovery, and transaction processing. On the other hand, a less sophisticated system may just support buffer management and file management. In some cases, even buffer manager and file manager may not exist. In these cases, the index has to directly interact with the operating system for all its file management and memory management activities.

As our goal is to design generic interfaces that will work for all database systems, we will adopt the following strategy. We will first identify a superset of functions a sophisticated database system supports. This set will be decided on the basis of functionalities supported by various commercial and research database systems. From this set, we will identify the set of software modules an index will depend on to implement those system-dependent functionalities. Finally, we will design interfaces to interface the index code with the set of software modules we identified.

We will design our interfaces as simple procedure calls that are callable from the index code. We will leave it to the access method designer to either incorporate them or not to incorporate them based on whether the functionality in question is supported by the database system or not. Note that if a functionality is not supported by a database system and if it is not incorporated in the index code, then the same index code will not work for another database system that supports the functionality. The alternate and correct choice would be to implement the index with all the lower interfaces as listed in this chapter and provide stubs for unsupported functionalities where ever necessary. This would lead to development of portable index code.

5.1.1 Software Modules An Index Interacts With in a DBMS

We have arrived at a superset of database functionalities based on the functionalities supported by the following commercial and research systems: Informix, Oracle, Sybase, XPRS[Stonebraker 1988], System-R[Astrahan 1976], Ingres[Held 1975], O2[Deux 1990], and Volcano[Graefe 1993c]. Based on the analysis, we note that most of the systems include support for logical & physical logging, concurrent transaction processing, crash recovery, different kinds of partitioning, clustering, and buffer management. Including support for cataloging, abstract file operations (Section 5.2), and abstract record operations (Section 5.3), the software modules an index will depend on in a high end database system are as follows.

Disk Manager.

Record Manager.

Buffer Manager.

Log/Recovery Sub-system

Lock Manager.

Transaction Manager.

Catalog Manager.

In the following sections, we discuss in detail the operations of the above modules.

5.2 Disk Manager

Our upper interfaces provided four different DDL operations such as "create()" for creating a file, "destroy()" for deleting a file, "open()" for opening a file and finally "close()" for closing a file. To accomplish these operations, an index has to directly interact with the operating system, if no other abstraction is provided by the database management system. If the access methods are to be generic and uniform, it is necessary to isolate the index from these operating system related details. The disk manager performs this part of the work by providing the index with an abstract set of file operations, isolating the index from the operating system related details.

The "create" function of our upper interfaces returned an unique permanent file identifier for every new index in the system. As every index will be mapped to a separate operating system file at the lower level, we will make it the responsibility of the disk manager to set unique permanent identifiers for each new file created in the system. The "open" function of the upper interfaces returned an unique open file handle for every open call. This open file handle associated the transaction that opened the file with it. As in the case of "create," we will make it the responsibility of the disk manager to set up unique identifiers for every open file and associate the transaction that opened the file with the open file handle.

This means that the disk manager, apart from providing a complete set of file operations, has to maintain appropriate data structures to support permanent file identifiers and open file identifiers.

The next section specifies the interfaces provided by the disk manager.[4]

5.2.1 Interface Specification

Create_file

Status create_file

(TRANSACTION_ID tid, const FILE_PARAM & file_param, FILE_ID& file_id);

Purpose	:	Create a new file with the specified file parameters.
Input	:	tid - Id of the transaction creating the file.
		file_param - Specifies system related parameters for the file.
Output	:	file_id - Unique file handle for the created file.
Details	:	

This method creates a new file in the database and assigns an unique handle for the file, which can be used for further operations on this file. The unique handle returned by this method should remain valid until a "destroy" call, and the handle should facilitate "open" and "destroy" operations on this file. While it is specified that an unique handle be assigned to the file, we do not specify the other associations of this handle. The system may choose to incorporate any other associations with this handle as may be demanded by the system's file structure.

Destroy_file

Status destroy_file(TRANSACTION_ID tid, FILE_ID file_id);

Purpose	:	Remove an existing file from the system.
Input	:	tid - Id of the transaction destroying the file.
		file_id - Permanent id of the file to be destroyed.
Output	:	Status Code.
Details	:	

This method removes a file from the database. Removing a file frees up the disk space occupied by the file. The "file_id" will be made available for reuse.

[4] The disk manager interfaces are almost similar to Cascades[Chapter 6] interfaces.

Open_file

Status open_file

(TRANSACTION_ID tid, FILE_ID file_id, MODE mode, OPEN_FILE_ID& open_id);

Purpose : Open an existing file for reading, writing, or for both.

Input : tid - Id of the transaction opening the file.

file_id - Permanent file id returned by a create call.

mode - Mode in which the file is to be opened.

Output : open_id - File handle that would permit read and write operations on this file.

Details :

This methods opens an existing file for reading, writing, or both. On success, this method is expected to return an unique identifier that would remain valid until a "close_file" call is made with that identifier. The identifier returned would serve as a handle for all data manipulation and lookup operations on this file. The identifier must associate the "file_id" with this "open_id".

This method is also expected to associate the "open_id" with the identifier of the transaction that opened the file. The database system may have further associations with this "open_id." Note that this method should permit multiple opens for the same transaction.

Close

Status close_file(OPEN_FILE_ID open_id);

Purpose : Close an open file.

Input : open_id - File handle returned by an open call.

Output : Status code.

Details :

This method closes the file. In case, the file has been opened multiple times, the file will not get closed until the closure of all opens. Irrespective of whether the file is closed or not, the "open_id" will be disassociated from this file and this will be available for re-use. For all practical purposes, the "open_id" will be no longer valid.

Alloc_page

Status alloc_page(OPEN_FILE_ID open_id, PAGE_ID& page_id);

PURPOSE : Allocate a new disk page and attach it to the

specified file.

Input : open_id - File handle returned by an open call.

Output : page_id - Logical page id of the new page.

Details :

This method attaches a new disk page to the specified file and returns a logical page identifier, which can be used for referring to this new page. This "page_id" must remain valid until the page is deallocated. It must also be possible to write to this page or read from this page directly using this "page_id".

Note that we are only specifying here that a new page be added to the file on each call. On efficiency reasons it may be necessary to allocate contiguous pages to a file. Depending on the type of file system, it is possible that a fixed number of contiguous pages may get allocated to each file when the file gets created. When ever this is the case, the disk manager must maintain a repository of free pages for each file and allocate from the free list a new page to the caller of this method.

Dealloc_page

Status de_alloc_page(OPEN_FILE_ID open_id, PAGE_ID page_id);

Purpose : Disassociate and release a disk page from the given file.

Input : open_id - File handle returned by an open call.
page_id - valid page_id returned by an alloc_page call.

Output : Status code.

Details :

This method disassociates a page from the given file.

Get_id

Status get_id(OPEN_FILE_ID open_id, FILE_ID& file_id, TRANSACTION_ID& tid);

Purpose : Get permanent file id and transaction id associated with the specified open_id.

Input : open_id : Identifier returned by an open call.

Output : file_id : Permanent file identifier.
tid : Id of the transaction that opened the file.

Details :

This method returns the permanent file identifier and the transaction identifier associated with the specified open file identifier. Providing this detail isolates the index from the open file table structure which is database-system dependent.

5.2.2 Discussion

With the inclusion of alloc_page and de_alloc_page interfaces, disk manager provides a complete set of interfaces, isolating the index from operating system related details. The abstraction provided by the disk manager is a necessary requirement for our interface design. The structure details of "file_param", and the associations of "file_id" and "open_id" have not been specified intentionally, as these will vary from system to system.

5.3 Record Manager

Record manager is not one of the standard modules found in every database system. In order to isolate the page structure details from index, we have designed record manager as a layer of abstraction in between the buffer manager and the index. Record manager provides a rich set of record level operations, hiding the page structure details from the index.

Our success here will depend on identifying a complete set of operations that an index may need to interpret the page structure. Page oriented index structures such as B-tree, R-tree etc., operate at the level of pages by writing or reading records from a specific page. These indices would also require page related information such as number of records in a page, and free space in a page to determine whether or not to split a page. Any design should take these factors into consideration. Another constraint to be addressed here is I/O efficiency. Providing record level operations can not lose out on the I/O efficiency possible on page level operations.

The following section specifies the interfaces provided by the record manager.[5]

---

[5] Some of the record manager interfaces are similar to Cascades[Chapter 6] interfaces.

5.3.1 Interface Specification

Read_record

Status read_record

(OPEN_FILE_ID open_id, REC_ID rid, RECORD& record, REC_LEN& rec_len);

Purpose	:	Read a record from an open file.
Input	:	open_id - File handle returned by an open call.
		rid - Id of the record to be read.
Output	:	record - Pointer to the record read.
		rec_len - Length of the record read.
Details	:	

This method reads a record from an open file. The input param "rid" specifies the identifier of the record to be read. On success, the output parameters "record" will point to the beginning of the record and "rec_len" will specify the length of the record.

In addition to reading a record, this method also fixes the corresponding page in memory by increasing the "fix_count" of that page. This page will remain in buffer unless and until the page is explicitly unfixed with an "unfix" call. This automatic fixing of page avoids redundant copying of the record and improves I/O efficiency.

Append_record

Status append_record

(OPEN_FILE_ID open_id, PAGE_ID page_id, const RECORD record, REC_LEN rec_len, BOOL compact, REC_ID& rid)

Purpose	:	Append a new record at the end of the specified page.
Input	:	open_id - File handle returned by an open call.
		page_id - Id of the page where the record is to be written.
		record - Pointer to the record to be written.
		rec_len - Length of the record to be written.
		compact - set to TRUE if page is to be compacted.
Output	:	Identifier of the new record.
Details	:	

This method appends a new record at the end of the specified page and returns a new identifier for the inserted record. The append will fail if there is no enough free space in the page. If "compact" is set to FALSE, free space at the end of the page will only be considered for appending the record. If "compact" is TRUE and if the new record can not be

accommodated within the available free space at the end of the page, the page will be compacted and then the append will be tried.

Compaction will not be done if there is no necessity even if "compact" is TRUE. This method calls the "compact" method of record manager for compacting a page.

Each "append_record" call fixes the page in the buffer by increasing the fix count of the page by one. The caller must explicitly unfix the page. This automatic fixing is done to ensure the page is readily available, if another write is to be done immediately.

Insert_record

Status insert_record

(OPEN_FILE_ID open_id, REC_ID rid, const RECORD record, REC_LEN rec_len, BOOL compact)

Purpose	:	Insert a new record in the specified page at the specified slot.
Input	:	open_id - File handle returned by an open call.
		rid - Expected id of the record.
		record - Pointer to the record to be written.
		rec_len - Length of the record to be written.
		compact - set to TRUE if page is to be compacted.
Output	:	Status code.
Details	:	

This method inserts a new record in the specified page at the specified slot. If the new record gets inserted in between existing records, all the records after the new record will get shifted maintaining the same ordering of records. This shifting doesn't mean physical shifting of the records. Only the record numbers of the shifted records will go up by one. Insert record guarantees that no record will ever get shifted out of the page.

Insert will fail if there is no free space to accommodate the record. This method by default will only look at the free space available at the end of the page for insertion. This behavior can be changed by setting the flag "compact" to be TRUE. Look at the "append_record" method for compaction semantics.

Each "write_record" call fixes the page in the buffer by increasing the fix count of the page by one. The caller must explicitly unfix the page.

Write_record

Status write_record

(OPEN_FILE_ID open_id, REC_ID rid, const RECORD record, REC_LEN rec_len);

Purpose : Modify the information content of an existing record.

Input : open_id - File handle returned by an open call.

rid - Expected id of the record.

record - Pointer to the record to be written.

rec_len - Length of the record to be written.

Output : Status code.

Details :

This method modifies the information or data content of an existing record. If the modified record is smaller than the original record it will result in internal fragmentation. No compaction will be done in this case. If the modified record is longer than the existing record, records in the page may get shifted. When ever this happens the page will be automatically compacted. The write will fail if there is no enough space, even after compaction, to accommodate the modified record. In case of failures the record to be modified will still contain the old information though the page may have been compacted.

Each "write_record" call fixes the page in the buffer by increasing the fix count of the page by one. The caller must explicitly unfix the page.

Delete_record

Status delete_record

(OPEN_FILE_ID open_id, REC_ID rid, BOOL reclaim_space);

Purpose : Delete a record from an open file, opened for writing and reclaim space, if specified.

Input : open_id - File handle returned by an open call.

rid - Id of the record to be deleted.

reclaim_space - Can be set to TRUE or FALSE. If TRUE the space occupied by the deleted record will be reclaimed immediately.

Output : Status code.

Details :

This method removes the specified record from the file. By default, this method will return after marking the deleted record invalid and adjusting the header information to reflect the current status of valid and invalid records. If the "reclaim_space" option is TRUE, in addition to the default behavior all the other records following the deleted record will be shifted forward removing any internal fragmentation caused by the delete. This method also

fixes the page in the buffer to facilitate subsequent operations on the page. The caller must explicitly unfix the page.

Get_page_details

Status get_page_details

(OPEN_FILE_ID open_id, PAGE_ID page_id, REC_NO& total_recs, REC_NO& valid_recs, REC_LEN& free_bytes);

Purpose	:	Get page related details.
Input	:	open_id - File handle returned by an open call.
		page_id - Id if the page for which details are required.
Output	:	total_recs - Reference to the total number of records in the page including invalid records.
		valid_recs - Reference to the number of valid records in the page.
		free_bytes - Total number of free bytes in the page.
Definitions	:	total_recs - Includes valid records and invalid records.
		valid_recs - Only valid records.
		invalid_records - Records that are deleted but for which, space has not yet been reclaimed.
		free_bytes - Means only the contiguous free bytes at the end of a page. Doesn't include the space occupied by invalid records.
Details	:	

This method gets certain relevant details about a page. "Free_bytes" indicates only the contiguous free bytes. Internal fragmentations are not taken into account. The reason for this is because, all the three "write" methods (append, insert and write) by default will only look for contiguous free bytes at the end of a page, to write a new record or modify a record on the page.

Compact_page

Status compact_page(OPEN_FILE_ID open_id, PAGE_ID page_id);

Purpose	:	Compact a page by reclaiming all the space occupied by all invalid records in the page and make them contiguous.
Input	:	open_id - file handle returned by an open call.

page_id - Id of the page to be compacted.

Output : Status code.

Details :

This method compacts a page by removing all the internal fragmentations and grouping all valid records together. After a "compact_page" call the total number of records and the valid records in the page will be same and the free bytes will truly reflect the total free bytes in the page. This method guarantees the ordering of records in the page, though the record numbers of the records might possibly get changed.

Unfix

Status unfix(OPEN_FILE_ID open_id, PAGE_ID page_id, REC_NO count);

Purpose : Unfix a page from buffer.

Input : open_id - File handle returned by an open call.

page_id - Id of the page to be unfixed.

count - Specifies the unfix count.

Output : Status code.

Details :

This method unfixes the "fix_count" of a page by the specified count. If the "fix_count" becomes zero, the page is set free to be flushed to the disk. An error will be returned if "count" exceeds the existing "fix_count" on that page. In this case, no unfixing will be done.

5.3.2 Discussion

The interface set of record manager meets all the I/O needs of all access methods. It is expected that the access method designer will route all his I/O calls using the record manager interfaces. Except the unfix interface, all the other interfaces of record manager fixes the read page in the buffer by increasing the fix count on that page by one. This mechanism is provided for I/O efficiency. The access method designer can make use of it to his advantage by completing all possible operations on the page before unfixing the same. The "reclaim_space" option in the "delete_record" interface has also been provided for efficiency reasons. Reclamation of deleted space is an expensive operation as it may involve

moving records within the page. Instead of reclaiming the deleted space immediately after every delete, the access method designer may choose to reclaim space, only when it is really necessary, improving the overall delete performance.

Get page details interface is provided for page oriented index structures. The information provided by "get_page_details" can be made use of to take a decision on compacting a page or splitting a page. Unfix provides a facility to unfix all the fix-counts in a single call.

No specific interface has been provided for updates involving modifications to the key value. We expect this updates to be carried out by a delete followed by an insert call.

5.4 Buffer manager

The basic idea of buffer management in any system is to increase I/O efficiency. The basic services of a buffer manager include reading a disk page in to memory, writing a buffer page to disk, fixing a page in memory or unfixing a page, and flushing all memory pages to disk in some pre-determined ordering.

In our desire to keep the page structure details isolated from the index, we have designed a layer of abstraction between the buffer manager and the index, the record manager. As all our I/O operations are routed through the record manager, the interaction of an index with the buffer manager is rather limited. Of course, the record manager has to interact with the buffer manager for all its I/O needs.

In the following section, we just provide one interface to allow the index to set the order in which the pages are to be written to disk. This is provided for supporting transactions. Database systems that support transactions have to log the changes to the data pages and these log pages have to be flushed to the disk prior to the corresponding data pages. The "order" interface of buffer manager is designed to serve this purpose.

5.4.1 Interface Specification

Order

Status order(OPEN_FILE_ID open_id, PAGE_ID first, PAGE_ID second);

Purpose : Determine the ordering of buffer pages for disk write.

Input : open_id - File id returned by an open call.
first - Id of the page to be written to disk first.
second - Id of the page to be written to disk second.

Output : Status code.

Details :

An error will be returned if the specified pages are not in memory or the pages do not belong to the specified file.

5.4.2 Discussion

The buffer manager interface is provided for the access method designer to force the log pages to stable storage before the data pages are written to disk, a necessary requirement for write-ahead logging. The buffer manager interface is also to be used for maintaining the consistency of the index structures. For example, in case of updates involving modifications to the index structure, it is always better to write the data pages first to disk before the internal pages are written to disk, since writing bottom up avoids dangling pointers. The access method designer can make use of the "order" function to achieve the index consistency. Note that this consistency has to be maintained by the access method irrespective of whether it is a logging database or a non-logging database.

5.5 Log-Recovery Sub-System

In this section, we discuss the interfaces for logging and crash recovery. If the database management system or the underlying operating system supports transaction by physically logging pages, the access method need not concern itself with transaction management and logging. Higher-level software will begin and end transactions, and the

access method can freely read and write records using the interfaces provided by the record manager.

[6]However, most of the systems have a variety of special code to perform logical logging of events rather than physical logging of changes of bits. There are at least two reasons for this method of logging. First, changes to the schema(e.g. Create an index) often require additional work besides changes to the system catalog(e.g. Creating an OS file). Undoing a create command because a transaction is aborted will require deletion of the newly created file. Physical back out cannot accomplish such extra function. Second, some database updates are extremely inefficient when physically logged. For example, if a relation is modified from a hash to B-tree, then the entire relation will be written to the log. It may be written more than once depending on the implementation of the modify utility. This costly extra I/O can be avoided by simply logging the command that is being performed. In the event that this must be re-done because of a crash, the command can be re-run to make the changes anew. For undo recovery, of course, the inverse of the command must be executed. In our case, we must have a modify utility to modify the relation from B-tree to hash. That means, such inverse commands or DML statements must be generated automatically as part of the regular recovery activity. Event logging, of course, sacrifices performance at recovery time for a compression of the log by several orders of magnitude.

As designers of generic index interfaces, we are not interested in guiding the selection of a particular type of logging. On the other hand, we are interested in providing an interface that works for any type of logging the system may choose to employ. For systems performing physical logging of pages , no separate interfaces with the log/recovery sub-system are necessary. Logging can be done transparent to the index. The record manager can log the modified pages without the index being aware of the logging process. For event logging however, an index has to interface with the log/recovery sub-system.

---

[6] The content of the discussion in this paragraph are taken from Stonebraker[1986].

The following section specifies our log interface.

5.5.1 Interface Specification

Log

Status log

(FILE_ID file_id, TRANSACTION_ID tid, EVENT_ID event_id, const RECORD log_record, REC_LEN log_rec_len)

Purpose : Write a new log record to the log.

Input : file_id - Permanent file identifier.
tid - Id of the transaction logging the event.
event_id - Id of the event to be logged.
log_record - Pointer to the log record to be written to log.
log_rec_len - Length of the log record.

Output : Status code.

Details :

This method writes a new log record to the system's log file. The system may add to the contents of the log record as demanded by the system's recovery logic. For example, systems may choose to chain the log records of a transaction for faster recovery.

5.5.2 Discussion

Viewed from a macro level, the index operations or events that result in schema or data changes can be summarized as follows.

Create Index.

Destroy Index.

Insert Record.

Delete Record.

Modify Record.

Load Records.

Each of the above operations will be identified with an unique identifier. As seen in Section 4.3, each access method in the system provides a redo and undo method for each of

the events noted above. The access method designer will make use of the log interface to create appropriate log records. As the information stored in the log record will only be interpreted by the access method that created it, each access method in the system can decide the content and format of this log record. The event_id identifies the event to be re-done or un-done during recovery.

The design of the log interfaces and log support interfaces greatly simplify the log manager's responsibilities, which is reduced to associating the log records with transactions, storing and retrieving the log records, and invoking the appropriate redo or undo interfaces of the index during a recovery. The log manager can also choose to store additional information as part of the log record to facilitate recovery. For example, the log manager may want to chain all the log records that pertain to a single transaction.

5.6 Lock manager

In a multi-user environment, concurrent access of processes to an index structure must be supported. The problem of concurrent access is that of allowing a maximum number of processes to operate on the index, without impairing the correctness of their operations.

A simple-minded solution for the problem of concurrent access would be to strictly serialize all updaters, by requiring each updater to gain exclusive control of the index - e.g., by placing an exclusive lock on the whole index, thus, preventing all other updaters and readers from altering or reading the index while the specific update takes place. Readers, on the other hand, could access the structure concurrently with other readers. Serializing access to index structures can create an unacceptable bottleneck for the entire system. Thus, there is a need for locking protocols that can assure integrity for each access, while at the same time providing a maximum possible degree of concurrency. Another feature required from these protocols is that they be deadlock free, since the cost to resolve a deadlock may be high.

Locking protocols differ for different access methods. Most access methods require more control over concurrency decisions. For example, most B-tree implementations do not hold write locks on index pages which are split, until the end of the transaction that performed the update. It appears easiest to provide specific lock and unlock calls for different granularities and leave it to the access method designer to implement appropriate locking protocols as permitted by that access method. The next section specifies the locking interfaces.

5.6.1 Interface Specification

Lock

Status lock
(OPEN_FILE_ID open_id, OBJECT object, OBJECT_TYPE object_type,
LOCK_MODE lock_mode);

Purpose	:	Lock an object in the specified mode. Object can be a file, a page, or a record. Mode can be any mode, the system permits.
Input	:	open_id - File handle returned by an open call. object - Id of the object to be locked. object_type - Indicates the type of the object. Can be a file, a page, or a record. lock_mode - Mode of the lock required.
Output	:	Status code.
Details	:	

This method locks an object in the specified mode. The object can be a file, a page or a record and it is indicated by "object_type." This method guarantees the following behavior. The caller will be put on wait if the request cannot be satisfied. If the caller is currently holding a lock on the specified object and if the request cannot be satisfied immediately, the caller will continue to hold the current lock and will be put to wait for the requested lock.

Unlock

Status unlock

(OPEN_FILE_ID open_id, OBJECT object, OBJECT_TYPE object_type);

Purpose	:	Unlock an object.
Input	:	open_id - File handle returned by an open call.
		object - Id of the object to be unlocked.
		object_type - Type of the object to be unlocked.
Output	:	Status code.

5.6.2 Discussion

As discussed earlier, special case concurrency possible on an index depends on the access method, and each access method requires more independent control over concurrency decisions. We believe that any effort to identify uniform locking protocols that can work for all access methods can only have serious limitations on the degree of concurrency and will fail to exploit any special-case parallelism possible on many access methods. Leaving the locking protocols to the access method designer and providing just mechanisms for locking and unlocking allows development and usage of more appropriate locking protocols that can provide maximum degree of concurrency without deadlocks for each different access method.

For the same reason, we have not defined the locking modes. Leaving it to the implementation supports development and usage of appropriate protocols.

5.7 Transaction manager

One of the hard problem of index interface design is interfacing the index code with the transaction management code[Stonebraker 86]. The major issues involved in transaction management are concurrency control, logging, crash recovery and interfaces to support begin, commit and abort semantics. We have already covered part of the issues of transaction management: the concurrency issues and log-recovery issues in the lock

manager and log/recovery sub-system sections. In this section, we will address the higher-level components of transaction management: begin, commit, and abort, and analyze whether it is necessary for an index to support begin, commit, and abort.

For systems that use physical page level logging, and executing one of the popular concurrency control algorithms for page size granules[Brown 81; Popek 81; Spector 83; Stonebraker 85] the index need not concern itself with transaction management. Locking can be implemented through the record manager interface, as an open file id uniquely maps to a transaction. Higher-level software will begin and end transactions and the access method need not be aware of the concept of a transaction. For these systems, the access method does not have to log the events and no special case concurrency need be employed using our lock manager interface. It is only in the case of event logging and finer granularity locking, one need to think whether begin, commit, and abort should become part of the access method interface.

There are two ways to look at this. On one hand, we could treat the index manager as a resource manager and make it a participant in a commit or abort decision. In this case, the index manager is aware of the concept of transactions and all index operations are performed at the level of transactions. That means that on the upper level the index manager should provide a begin, commit, and abort interface to the higher level software in the system. On the lower level, more interfaces with the buffer manager and log/recovery subsystem will become necessary. For example, one may need a FLUSH interface with the buffer manager to flush all the data pages pertaining to a transaction to be flushed to disk, when a commit is to be performed. The index manager will have to maintain appropriate data structures to keep track of active transactions, the data pages associated with each transaction, and the operations performed by each transaction. This scenario will be acceptable if a relation's primary representation is an index structure as in Tandem's Non-Stop SQL and Sybase's SQL Server. But, we could choose to represent some relations as simple files and as well have

index representation for some relations. In this case, making an index manager handle begin, commit, and abort will result in duplication of code and effort. This duplication of work may also lead to inconsistencies if not properly implemented.

On the other hand, we believe it is possible to remove the concept of transaction from the index interface and provide begin, commit, and abort as modules implemented by the higher level softwares in the system. This can be achieved using only our existing lock manager, log manager and buffer manager interfaces. In the next few paragraphs we go back to these interfaces and explain how adding or removing begin, commit and abort do not add or sacrifice any index functionality or performance.

Our lock manager provided two simple interfaces for locking and unlocking at different granularities. These interfaces are provided for taking advantage of any special-case concurrency possible in an access method. The protocol to be used for locking and the granularity of locking is dependent on the access method. One cannot possibly have a better degree of concurrency or no extra benefit will arise by making begin, commit and abort part of the index code.

With regard to event logging, we have made it the responsibility of the access method designer to decide on the nature of events to log, when to create a log record, and what type of information to store in a log record to effectively perform a redo or undo. As redo and undo are part of the index code, the event logging activities are independent of the system. The design of "log" interface of the log manager, the "order" interface of the buffer manager, and the "redo" and "undo" of the upper log support interfaces provide this independence. Making begin, commit and abort as part of index interfaces cannot possibly have any effect on this design.

Based on the above considerations, we believe adding the transaction code (begin, commit, and abort) will only add to the index overhead, and will not add any functionality or performance benefit. It is possible and seems better to make begin, commit and abort as part

of the higher level modules and leave it to the higher level modules to begin and end transactions, removing the concept of transaction from the index interface.

5.8 Catalog Manager

The catalog manager interface is provided for manipulating index catalogs. Here, we will model every index as being associated with an unique index header record that contains index specific information such as : index identifier, index storage type related information(type of index, KEY type), index access information(the root page id, height of the index), index statistical information(record cardinality, key cardinality), and index constraint information(low and high key limits). Except the index storage type related information, all the other information is directly manipulated by the index. Storage type related information is set during index creation and they remain the same as long as the index exists.

In the next section, we specify an index header record that is required to support our interface design, and design catalog interfaces for reading and writing these records. The index header record designed here contains only the minimum required information that is necessary for our interface design. Based on other needs of the system, the system may choose to store more information as part of the index header record. Also, it is not a necessary part of the design to have a separate index catalog. The system may choose to store all the index header records as part of some other relation catalog and do away with the index catalog altogether.

5.8.1 Index Header Record

The index header record is specified as a C structure.

```
typedef struct INDEX_HDR_RECORD
{
        FILE_ID                 file_id; // Permanent index identifier
```

```
        INDEX_PARAM          index_param;  // Holds create parameters and
                                           //    integrity constraint Details.
        INDEX_ACCESS index_access;// Holds access information.
        INDEX_STATS  index_stats;    // Holds statistical information.
} INDEX_HDR_RECORD;
```

INDEX_PARAM, INDEX_ACCESS, and INDEX_STATS are defined in Appendix B.

5.8.2 Interface Specification

Read_catalog

Status read_catalog(FILE_ID file_id, INDEX_HDR *index_hdr);

Purpose	:	Read an index header record from the index catalog.
Input	:	file_id - Permanent identifier of the index returned by a create call.
Output	:	index_hdr - Pointer to the index_hdr record.

Write_catalog

Status write_catalog(FILE_ID file_id, const INDEX_HDR& index_hdr);

Purpose	:	Write an index_header record to the index catalog.
Input	:	file_id - Permanent identifier of the index returned by a create call.
Output	:	index_hdr - Index header record to be written to the catalog.

5.8.3 Discussion

The "read_catalog" and "write_catalog" methods provide access to the catalog information. The index manager and all higher-level software modules can access the catalog information only through the above interfaces.

The design of the catalog manager interface completes our lower interface design. As these interfaces have been designed for a superset of database functionalities, we expect these interfaces to work for most of the database systems. Where ever a database system does not support some or all of the functionalities discussed here, the indices in those systems will not have to make calls to the respective software modules. However, as noted,

system support is required for the "record manager" and "disk manager" interfaces for our interfaces to work correctly.

The design of the lower interfaces completes our abstract index interface design. In the next chapter, we discuss our prototype implementation.

6 PROTOTYPE IMPLEMENTATION

We tested our interface design, implementing two different access methods and interfacing these access methods with a research prototype database management system called Cascades. The next section gives a brief description of Cascades, and Section 6.2 details the access methods implementation.

6.1 Cascades

Cascades is an extensible and modular database management system that was being developed to provide a testbed for database systems education and research at Portland State University, Portland, Oregon. The development of Cascades is currently suspended and as of this writing only the file system has been implemented. The design and implementation of Cascades follows many of the ideas outlined by Volcano[Graefe 1993a], an extensible and parallel dataflow query processing system. Cascades's file system is rather conventional. It includes modules to manage devices, buffer pools, files, records, B+-tree, and R-tree access methods. Support for logging, recovery and transaction has not been implemented. The interaction of indices with the lower level modules are limited to record manager, buffer manager, and disk manager. As there is no support for concurrent transactions and logging, the buffer manager does not support the order interface. The interfaces supported by record manager and disk manager are not exactly similar to the interfaces designed in this thesis. There are some differences and Section 6.1.1 specifies all the index interfaces of Cascades. On the upper level, the B+-tree and R-tree indices currently do not interact with any higher-level modules as no query processing routines have been implemented yet.

6.1.1 Index Interfaces of Cascades

In this section, we specify the lower interfaces supported by Cascades which includes support for file management and record management. The buffer manager interfaces of Cascades are not listed here as the index doesn't interface with the buffer manager. We begin with the definition of the relevant data structures.

6.1.1.1 Data Structures

```
#define MAX_NO_OF_DEVS    (10)     // Limit on size of striped files
typedef int STATUS;                // Return values
typedef int BOOLE;

// Types associated with files
typedef int FILE_ID;               // Permanent file id
typedef int OPEN_FILE_ID;          // Id for open files

// Record id's and related types
typedef short DEV_NO;              // Id of a device
typedef int PAGE_NO;               // Id of a page within a cluster
typedef int REC_NO;                // Id of a record within a page
typedef int REC_LEN;               // Length of a record
typedef int FIX_COUNT;                     // Fix count of a page
typedef int EXT_SIZE;              // Extent size in clusters
typedef short CLU_SIZE;            // Cluster size in pages

// Cluster Identifier
typedef struct CLU_ID
{
   DEV_NO dev_no;
   CLU_SIZE clu_size;
   PAGE_NO page_no;
} CLU_ID;
```

```
// Record identifier
typedef struct REC_ID
{
   CLU_ID clu_id;
   REC_NO rec_no;
}REC_ID;
// File types.
typedef enum TAG
{
   EMPTY, BASE, STRIPE, LIST, RECORD
} TAG;

// Access status of a file
typedef enum MODE
{
   FILE_MODE_WRITE, FILE_MODE_READ
} MODE;

// Create Parameters for a base file
typedef struct BASE_FILE_CR_PARAM
{
   EXT_SIZE          prim_ext_size;
   EXT_SIZE          secnd_ext_size;
   CLU_SIZE          cluster_size;
   DEV_NO            dev_no;
} BASE_FILE_PARAM;

// Create parameters for a stripe file
typedef struct STRIPE_FILE_CR_PARAM
{
   int               no_of_disks;     // # of disks in striped file.
   int               no_of_clusters; // # of clusters per partition
   DEV_NO            stripe_dev_no[ MAX_NO_OF_DEVS ];// dev_no of each disk.
} STRIPE_FILE_CR_PARAM;
```

```
// Create parameters for a list file
typedef struct LIST_FILE_CR_PARAM
{
   DEV_NO            dev_no; // device of component file
} LIST_FILE_CR_PARAM;

// Create parameters for a composite file
typedef struct COMPOSITE_FILE_CR
{
   TAG tag;
   union {
      struct BASE_FILE_CR_PARAM *base_info;
      struct STRIPE_FILE_CR_PARAM *stripe_info;
      struct LIST_FILE_CR_PARAM *list_info;
   } create_param;
   COMP_PTR next_component;
}COMPOSITE_FILE_CR;

// Create parameters for a file of records.
typedef struct RF_CREATE
{
   REC_LEN    threshold;     // Minimum free bytes for space reclamation
   TAG        comp_tag;      // File type
   COMPOSITE_FILE_CR *composite; // Create parameters
} RF_CREATE;
```

6.1.1.2 File Management

Cascades file manager supports four file operations: create, destroy, open and close. These interfaces are almost similar to that supported by our disk manager and they differ only in the data structures. The following section specifies the interfaces.

Rf_create

STATUS rf_create

(RF_CREATE create_param, DEV_NO dev_no, OPEN_FILE_ID *open_id,
FILE_ID *closed_id)

Purpose	:	Create a file of records.
Input	:	create_param - Specifies the parameters for the file to be created.
		dev_no - device where the file resides.
Output	:	
Details	:	

The status of the created file is open for writing. The cluster prepared for writing is the first in the file.

Rf_destroy

STATUS rf_destroy(FILE_ID closed_id, DEV_NO dev_no);

Purpose	:	Destroy a file of records.
Input	:	closed_id - Permanent file id returned by create
		dev_no - Device identifier
Output	:	Status code.
Details	:	

All related data structures are placed on free lists and for all practical purposes are gone. The data can often be restored to valid status if the space originally occupied by the file has not been reclaimed.

Rf_open

STATUS rf_open

(FILE_ID closed_id, DEV_NO dev_no, MODE mode, OPEN_FILE_ID *open_id);

Purpose	:	Open a file for a specified mode.
Input	:	closed_id - Id returned by a create call.
		dev_no - Id of the device.
		mode - Mode for open.
Output	:	open_id - Id of the open file.
Details	:	

Depending on the mode, a cluster from some position within the file is loaded to the buffer and a pointer to this cluster is returned to the caller.

Rf_close

STATUS rf_close(OPEN_FILE_ID open_id);

Purpose	:	Close a file.
Input	:	open_id - Id returned by an open call.
Output	:	Status code.
Details	:	

It is assumed at the time of this call that all clusters belonging to this file have been unfixed in the buffer. Call will fail if this is not true.

6.1.1.3 Record Management

The record manager interfaces supported by Cascades are not similar to that supported by our design. They differ in the variety and type of interfaces, the data structures they use and in the operations they perform.

Cascades interfaces have the concept of implicit current cluster which is non-existent or non-visible to the index in our design. Most of the Cascades interfaces read and write from this "current cluster", when no cluster is specified explicitly. When a file is created or opened, a cluster is loaded on to memory for that file and that is called the current cluster. For an existing file, the last cluster on which an operation was done is called the current cluster. The other major difference has to do with forwarding a record to a new cluster. Most of the interfaces of Cascades that do an append, insert, or update forward the record to another cluster if the specified record can not be accommodated in the current or the specified cluster. The other differences have to do with the data structures and the variety of operations supported.

Rf_append

STATUS rf_append

(OPEN_FILE_ID open_id, REC_LEN length, BOOLE this_clu, char **addr, REC_ID *rid);

Purpose	:	Append a new record to an open file.
Input	:	open_id - Id returned by an open call.

		length - Length of the record.
		this_clu - set to TRUE if to be appended only in the current cluster.
Output	:	addr - Address for the new record
		rid - Id of the new record.
Details	:	

This call, if results in success returns an address for the record to be appended to be copied in. If there is no enough free space and if "this_clu" is TRUE the call will fail. If "this_clu" is not set to TRUE, this method will try the append in other clusters. This method doesn't reclaim free space. The caller must explicitly unfix the cluster.

Rf_append_new

STATUS rf_append_new

(OPEN_FILE_ID open_id, REC_LEN length, char **addr, REC_ID *rid);

Purpose	:	open_id - Id returned by an open call.
Input	:	length - Length of the record.
Output	:	addr - Address for the record.
		rid - Id of the record.
Details	:	

Appends a record to a brand new cluster.

Rf_insert

STATUS rf_insert

(OPEN_FILE_ID open_id, REC_LEN length, REC_ID hint, BOOLE strict_hint,
BOOLE this_clu, char **addr, REC_ID *rid);

Purpose	:	Insert a new record.
Input	:	open_id - Id returned by an open call.
		length - Length of the record
		hint - Specifies the expected id of the record.
		strict_hint - If TRUE call will fail if "hint" can not be accommodated.
		this_clu - set to TRUE if to be inserted in this cluster only.
Output	:	addr - Address for the new record.
		rid - Id of the inserted record.

Details :

This method reclaims space if there is no enough free space to insert the new record. If there is not enough free space even after compaction this method will try to reclaim space from a cluster on the free list. If there is no enough space on the free list clusters, a new cluster will be appended to the file.

The caller may give a "hint" in the form of cluster id as to where the record should be inserted. If "strict_hint" is TRUE call will fail if there is no free space in the specified cluster. The boolean "this_clu" restrict the insert to the current cluster. Call will fail if current cluster doesn't have a free space.

Rf_newreclen

STATUS rf_newreclen
(OPEN_FILE_ID open_id, REC_LEN length, REC_ID rid, BOOLE saveold,
BOOLE this_clu, char **addr);

Purpose : Change the length of an existing record.

Input : open_id - Id returned by an open call.
length - New length
rid - Id of the record.
saveold - If TRUE old data will remain good.
this_clu - If TRUE try within current cluster.

Output : addr - Address for the new record.

Details :

If there is no enough free space in the current cluster then the record is forwarded to a new cluster unless "this_clu" is TRUE.

Rf_delete

STATUS rf_delete
(OPEN_FILE_ID open_id, REC_ID rec_id, BOOLE reclaim);

Purpose : Delete an existing record.

Input : open_id - Id returned by an open call.
rec_id - id of the record to be deleted.
reclaim - If TRUE deleted space will be reclaimed
immediately.

Output : Status code.

Details :

The deleted space will be on free list if not reclaimed. It is possible to undelete a record if the space has not yet been reclaimed.

Rf_scan

STATUS rf_scan

(OPEN_FILE_ID open_id, BOOLE physical_order, char **addr, REC_ID *rid);

Purpose	:	open_id - Id returned by an open call.
Input	:	physical_order - If TRUE forward records are returned.
Output	:	addr - Address of the record.
		rid - id of the record.
Details	:	

Scan a file one record at a time. If the flag "physical_order" is TRUE, forwarded records are read only when the actual data is encountered. If not they will be read when the forwarder is encountered.

Rf_unfix

STATUS rf_unfix(OPEN_FILE_ID open_id, REC_ID rid, REC_NO count);

Purpose	:	Release fix counts on a cluster.
Input	:	open_id - Id returned by an open call.
		rid - Specifies the cluster.
		count - Number of times to unfix.
Output	:	Status code.
Details	:	

Releases "count" rights to the cluster specified in rid.

Slot_info

STATUS slot_info

(OPEN_FILE_ID open_id, CLU_ID clu_id, REC_NO *used_slots, REC_NO valid_slots, REC_LEN *lowest_offset, REC_LEN bytes_avail);

Purpose	:	Get cluster information.
Input	:	open_id - Id returned by an open call.
		clu_id - Id of the cluster for which info is requested.
Output	:	used_slots - Total records in the cluster including invalid ones.
		valid_slots - Total valid records.

lowest_offset - The lowest offset for the next record.
bytes_avail - Free contiguous bytes available.

Rf_read

STATUS rf_read

(OPEN_FILE_ID open_id, REC_ID rid, char **addr, REC_LEN *length);

Purpose	:	Read a record.
Input	:	open_id - Id returned by an open call. rid - Id of the record to be read.
Output	:	addr - Memory address of the record. length - Length of the record.
Details	:	

Get memory address for the specified record. The caller must explicitly unfix the page when the address is no longer needed.

6.2 Access Methods Implementation

We decided to test our design with three different access methods. Our natural choice was to select B+-tree and hash indices as these are the most commonly used access methods in many database systems. For the third, we decided on R-trees, a dynamic, multi-dimensional, page-oriented index structure that is increasingly being used in database systems. Other than the reasons of popularity, the major deciding factors were, first, the type needs of these three different access methods differ from one another and Second, the inclusion of a multi-dimensional index structure gives us an opportunity to test different non-conventional data types. The efficiency of the MATCH class could be thoroughly tested as one could define many different retrieval operations for a multi-dimensional data type.

As of this writing, B+-tree and R-tree indices have been fully implemented. The hash index has not been implemented. The implementation of B+-tree and R-tree indices support the same set of upper interfaces except the optimizer support interfaces and log support interfaces. We have not implemented both the optimizer and log support interfaces as Cascades do not have support for logging and it does not have an optimizer yet. The type-

dependent needs are met through the KEY and MATCH class libraries. The lower interfaces are different from the interfaces specified in Chapter 4. This is because of the differences in Cascades interfaces. As development of Cascades is suspended now, we modified our lower interfaces to that supported by the current implementation of Cascades[Section 6.1.1].

As the underlying system does not include support for transaction, logging and recovery, the current implementation of these two access methods do not support concurrency, logging, and recovery. These access methods can be modified to include support for the above three functionalities with very little effort. This is infact, one of the major advantages of our work. All that, one has to do to include support for concurrency, logging and recovery is to make calls to the respective interfaces at the appropriate places in the existing code. No other change need be made with the existing code.

6.2.1 B+-tree Implementation

The B+-tree implementation supports top-down split. Deletion does not shrink the tree. This is to support better performance at the cost of space. The leaf pages are chained together for fast scanning. A range search in a B+-tree implementation traverses the index from root to leaf only once. Subsequent "next_scans" have to only follow the chain for the next record. The B+-tree implementation allows duplicates. The code of the B+-tree implementation is in both "C" and "C++". Total lines of code is approximately 2500 lines.

6.2.2 R-tree implementation

The R-tree implementation uses "liner-cost" algorithm[Guttman 1984] for splitting nodes. As R-trees do not have direct algorithmic support for bulk-loading, we have implemented loading using the insert algorithms. Scanning in a R-tree is also bit inefficient compared to a B+-tree. In a R-tree, it is not possible to chain the leaf pages, as the next physical record need not be the next logical record as per index ordering. The "scan_list" data

structure helps to bring in some efficiency by storing relevant information to avoid re-traversal all the time. However, we couldn't avoid re-traversal altogether, as moving to the next leaf page involves referring to the parent-node. The R-tree implementation allows duplicates. The code of the R-tree implementation is in both "C" and "C++". Total lines of code is approximately 3500 lines.

6.2.3 Testing

Both the B+-tree implementation and the R-tree implementation were compiled and tested on a "sunos" platform using a "cfront" based "C++" compiler. Our basic testing strategy was to create many different B+-tree and R-tree indices each storing different data types, and to perform data manipulation and retrieval calls through the upper interfaces. As the query processing routines of the underlying system are yet to be implemented, we could only test our upper interfaces by making direct index calls through test programs, and not through a query language interface. The test program code is in both "C" and "C++". Total lines of code is approximately 500 lines. Creating indices with different data types and performing retrieval with different operators were done by providing different implementation classes for KEY and MATCH class libraries, for each tested data type and retrieval operator respectively. The B+-tree implementation was tested by creating three different indices with int, float, and string data types. The R-tree implementation was tested by creating three different indices with three different data types. The three data types are,

1, A "box" data type(a 4 dimensional string type data).

2, A "line" data type(a 2 dimensional int type data).

3, A "point" data type(a 1 dimensional float type data).

We executed all the tests with both duplicate data and non-duplicate data. The test program tested all the upper interfaces supported by both the B+-tree and R-tree

implementations. For the B+-tree implementation selection of records were based on both exact-match and range queries for all the three indices.

The selection criteria for the R-tree implementation differed for different data types. For the "box" data type we used "containment", overlapping", "exact-match", "area greater than", and "area less than" relationships. Area here indicates the area covered by the box. For the line data type we used "containment", "overlapping", "exact-match", "distance greater than", and distance less than" relationships. Distance here indicates the distance between the two data points of the line. For the "point" data type we used both exact-match and range queries. These different selection criterions were imposed through different implementations of MATCH class.

We did not do any performance studies, as our main goal in the prototype implementation was to test the applicability of the interfaces. The tests would have been more complete if we had interfaced the implemented access methods with different database systems. So far, we have not had an opportunity to do so.

7 SUMMARY AND CONCLUSIONS

In this thesis, we have defined type-independent generic index interfaces that would work for a large set of access methods and for many database management systems. We started our work by identifying how an index fits in a database system. We identified three different levels of interfaces that completely characterize an index in a database system. These were: upper interfaces, lower interfaces, and type-dependent interfaces. This modular perspective of indices in database systems led to the possibility of developing type-independent generic interfaces.

Under upper interfaces, we first identified a complete set of functions that an index can provide to a database management system. Support for the optimizer and the log manager completes the set of upper interfaces. Our work in type-dependent interfaces was to identify a complete set of type-dependent functions for a finite set of access methods. We had to work with a finite set of access methods because of the existence of access method specific type-dependent functions. We grouped the identified type-dependent functions into two classes, one that depends on the access method and the other that depends on the retrieval operators.

We designed the lower interfaces, first by identifying a set of software modules an index has to depend on in a database system to accomplish database system related functions, and designed interfaces for interfacing the index code with these modules. We have modelled the index interaction with a database system as simple procedure calls callable from the index code. This design promotes portability of an access method to any database system.

7.1 Conclusions

Our research proves that it is possible to design type-independent abstract index interfaces that would work for a major set of access methods and database systems. There are two major areas that restrict the development of type-independent and database-independent access methods. They are, the type-dependent needs of an access method and the dependency on the database system software to accomplish the database system dependent functions. Our research has shown that it is possible to abstract both these issues with appropriate interface design.

Our design of abstract interfaces makes it possible to develop generic access methods that can be purchased from a third party vendor like any other ready made software component and fit into any database system.

7.2 Future work

In this section, we identify areas where there are possibility for improvements. The current design of KEY class expects all dimensions to be either of range values or point values. It doesn't allow some dimensions to be of range and some to be of point types. This restriction can be overcome by modifying the "KEY_REC" structure and changing some of the protected virtual functions. It may also be necessary to add few more functions to support this combination.

7.2.1 Single Interface

Another change we would like to see in the "KEY" class relates to usage of the class. We discuss this with examples.

As per the current design, an application would declare "KEY" objects of two different types as follows. Supposing class STRING_KEY relates to a string implementation and class

INT_KEY refers to an integer implementation, the objects of both classes would be declared as,

```
STRING_KEY skey;
INT_KEY      ikey;
```

This truly doesn't support the interface-implementation paradigm. We see differences of type at the interface level. Instead, it should be possible to declare a "KEY" object as "KEY key;" and this "key" instance should behave as a "string key" or "int key" depending on the implementation it is associated with. To make this feasible, we need to have a producer, for example a "Key Manager" which will produce keys of required types. This products will have an implementation associated with them and their behavior will depend on this association. It may require more research to identify whether the benefit would worth the effort.

7.2.1 Templates

We have provided type abstraction using C++ virtual functions. This may have some impact on the run time performance as virtual functions will be bound only at run time. Alternatively, it may be possible to provide the same abstraction using "template" classes such as the one supported by C++. Since "templates" are compile-time bound they offer an extra layer of type safety. The factors that will guide the selection of one or the other would be functionality, type safety and run time performance. More research needs to be done to arrive at a decision.

7.2.2 Multi-threading

An important issue we have not addressed so far is multi-threading. In systems that support shared libraries, it is possible that the same implementation of a "KEY" class or a "MATCH" class may be shared by multiple threads. If critical sections are not guarded against concurrent access, the internal data structures of the implementation may

stand corrupted. It is essential that the implementations be coded appropriately for multi-thread safety.

APPENDIX A

Here, we list the access methods taken into consideration for our design. The type dependent interface has been designed based on this list of access methods.

B-tree and all its variants	- Bayer 1972, Comer 1979
Extendible Hashing	- Fagin et al 1979
Grid-Files	- Nievergelt 1984
Hbtrees	- Lomet 1990
ISAM	- Larson 1981
Kdtrees	- Bentley 1975
KDB-trees	- Robinson 1981
Linear Hashing	- Litwin 1980
Quad-trees	- Finkel 1974
R-trees	- Guttman 1984

APPENDIX B

Data Structure Definitions

```
typedef int TRANSACTION_ID;
typedef int FILE_ID;
typedef int OPEN_FILE_ID;
typedef long REC_LEN;
typedef int REC_NO;
typedef short BOOLE;
typedef int CLU_SIZE;
typedef int PAGE_NO;

/* Here, we are modelling cost as number of I/O's performed.
 * It is up to the system to incorporate appropriate cost model.
 */
typedef float COST;
typedef short DEV_NO;
typedef void *RECORD;

// File access.
typedef enum MODE
{
   FILE_MODE_READ, FILE_MODE_WRITE
}MODE;

// Type of events to log
typedef enum EVENT
{
   Create, Destroy, Insert, Delete, Modify, Load
} EVENT;

// Type of objects for locking.
typedef enum OBJECT_TYPE
```

```
{   File, Page, Record } OBJECT_TYPE;
// Contains predicate information.
typedef struct PREDICATE
{
        OPERATOR_ID         opr_id;             // Id of the operator.
        RECORD              column;             // Array of column names
        REC_LEN             *column_length; // Array of column lengths
        REC_NO              column_count;   // Number of columns in the array
        RECORD              values;             // Pointer to the list of values.
        REC_NO              val_count;          // Number of values in the list.
} PREDICATE;

/*
 * Page identifier. This is an example. As page structure and file structure are
 * system dependent, system may choose to have its own page identification.
 */
typedef struct PAGE_ID
{
        DEV_NO          dev_no;
        PAGE_SIZE       page_size;
        PAGE_NO         page_no;
} PAGE_ID;

// Record identifier. Would depend on page_id.
typedef struct REC_ID
{
        REC_NO          rec_no;
        PAGE_ID         page_id;
} REC_ID;

// Object for locking.
typedef union OBJECT
{
        REC_ID          rid;
        PAGE_ID         page_id;
```

```
        FILE_ID         file_id;
}OBJECT;

/* Contains information to perform iterative loading.
 * The design of the structure permits iterative loading in all access
 * methods listed in appendix - A.
*/
typedef struct LOAD_LIST
{
        page_id             page_id;            // Id of the page being loaded.
        RECORD              load_information;   // Any specific load related info.
        REC_LEN             load_rec_len;       // Length of the information.
        struct LOAD_LIST    *next;              // Pointer to the next element.
} LOAD_LIST;

/*
 * This is an example implementation of open_load_table.
 * It is not necessary to have open_load_table if the load_status information
 * can be directly associated with open_id.
 */
typedef struct OPEN_LOAD_TABLE
{
        OPEN_FILE_IDopen_id;        // Load Information
        LOAD_STATUS         *root;
}OPEN_LOAD_TABLE

// Contains information about active scan pages
typedef struct SCAN_LIST
{
        REC_ID              rid;    // Id of the current scan record.
        struct SCAN_LIST    *next;  // Pointer to the next page in the scan.
} SCAN_LIST;

// Example implementation of open scan table.
typedef struct OPEN_SCAN_TABLE
```

```
{
        OPEN_FILE_ID        open_id;          // open_file file handler.
        MATCH               *match;           // Pointer to match library
        KEY                 *scan_interval;   // Scan interval.
        SCAN_LIST           *root;            // Pointer to scan list.
}OPEN_SCAN_TABLE;

//INDEX_PARAM - Contains user supplied information for creating an index.
typedef struct INDEX_PARAM
{
        // User must specify these details during index creation.
        INDEX_TYPE    indexType;
        KEY_TYPE      keyType;
        BOOLE         duplicates;
        /*
         * Integrity Constraints.- It is not a must for the user to supply
         * this information during index creation. However, if these are not
         * specified at creation time, index will not enforce key limits.
         */
        KEY           low_key[],
                      high_key[];
        BOOLE         enforce_limits;
} INDEX_PARAM;

// Contains statistical information about an index.
typedef struct INDEX_STATS
{
        REC_NO        key_cardinality,
                      record_cardinality,
                      num_data_pages,
                      num_index_pages;
        KEY           min,              // Min key values in each dimension
                      max;              // Max key values in each dimension
} INDEX_STATS;
```

```
// Holds access information.
typedef struct INDEX_ACCESS
{
        PAGE_ID             root_id;
        int                 height;
} INDEX_ACCESS;

// Index header record.
typedef struct INDEX_HDR
{
        INDEX_PARAM         index_param;
        INDEX_STATS         index_stats;
        INDEX_ACCESS        index_access;
}INDEX_HDR;

// FILE_PARAM
// This is not specified. The information content of the structure
// depends on the system and its file structure. The design of this
// structure would depend on what details the system expects from the
// user to create a file in the system.
typedef struct FILE_PARAM
{
} FILE_PARAM;

//CREATE_PARAM-Parameters for creating an index.
typedef struct CREATE_PARAM
{
        INDEX_PARAM         index_param;
        FILE_PARAM          file_param;
}CREATE_PARAM;

// Status : Error Codes
enum Status {
          okay,                     // Call Succeeded in all respects
          noOP,                     // Not a valid operation
```

```
        argErr,                 // Wrong type or Unexpected value
        dimErr,                 // Dimension out of range
        duplicate,              // Duplicate key.
        endOfFetch,             // All reqd records fetched
        endOfFile,              // No more records in the index
        errorCreate,            // Error during creation
        internalErr,            //  Some internal error.
        indexDoesNotExist,      // Wrong file_id or open_id
        invalidInterval,        // Interval value out of range.
        invalidPredicate,       // Predicate references multiple-
                                // columns or a non-existent
                                // column.
        noFreeSpace,            // Page is full.
        notAllowed,             // Operation not permitted
        notSupported,           // Feature not supported
        pageDoesNotExist,       // Invalid page_id
        recordDoesNotExist,     // Invalid rec_id
}Status; // Error Codes
```

REFERENCES

Astrahan 1976 : M.M. Astrahan, et al. System R: A relational approach to database management.(ACM Tarns. Database Syst. 1, 2 (June 1976) 97-137.

Batory 1986 : D. S. Batory, GENESIS: A project to Develop an Extensible Database Management System, Proc. Int'l Workshop on Object-Oriented Database Sys., Pacific Grove, CA, September 1986, 207.

Bayer 1972 : R. Bayer and E. McCrieghton, Organization and Maintenance of Large Ordered Indices, Acta Informatica 1, 3(1972), 173.

Baeza-Yates 1989 : R. A. Baeza-Yates and P. A. Larson, Performance of B+-Trees with Partial Expansions, IEEE Trans. on Knowledge and Data Eng. 1, 2(June 1989), 248.

Becker 1991 : B. Becker, H. W. Six, and P. Widmayer, Spatial Priority Search: An Access Technique for Scaleless Maps, Proc, ACM SIGMOD Conf, Denver, CO, May 1991, 128.

Beckmann et al. 1990 : N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles, Proc, ACM SIGMOD conf, Atlantic City, NJ, May 1990, 322.

Bentley 1975 : J. L. Bentley, Multi-dimensional Binary Search Trees Used for Associative Searching, Comm. of the ACM 18, 9(September 1975), 509.

Bentley 1977 : J. L. Bentley and D. F. Stanat and E. H. Williams, Jr., The Complexity of Fixed-Radius near neighbor searching, Inf. Proc. Lett. 6, 6(December 1977), 209-212.

Bentley 1979 : J. L. Bentley and J. H. Friedman, Data Structures for Range Searching, Computing Surveys 11, 4 (December 1979), 397.

Carey 1986 : M. J. Carey, D. J. DeWitt, J. E. Richardson and E. J. Shekita, Object and File Management in the EXODUS Extensible Database System, Proc. Int'l Conf. on Very Large Data Bases, Kyoto, Japan, August 1986, 91.

Comer 1979 : D. Comer, The Ubiquitous B-tree, ACM Computing Surveys 11, 2(June 1979), 121.

Deux 1990 : O. Deux et al., The Story of O2, Transactions on knowledge and data engineering 2(1): 91-108.

Finkel 1974 : R. A. Finkel and J. L. Bentley, Quad Trees: A Data Structure for Retrieval on Composite Keys, Acta Informatica 4, 1(1974), 1.

Graefe 1993 : G.Graefe, Options in Physical Database Design, ACM SIGMOD record, 22(3), September 1993.

Graefe 1993a : G. Graefe, Volcano, An Extensible and Parallel Dataflow Query Processing System, IEEE Trans. on Knowledge and Data Eng. 5, 6 (December 1993).

Guenther 1991 : O. Guenther and J. Bilmes, Tree-Based Access Methods for Spatial Databases: Implementation and Performance Evaluation, IEEE Trans. on knowledge and Data Eng. 3, 3(September 1991), 342.

Guibas 1978 : L. Guibas and R. Sedgewick, A Dichromatic Framework for Balanced Trees, Proc. 19th Symp. on the Found. of Comp. Sci., 1978.

Gunther 1987 : O. Gunther and E. Wong, A Dual Space Representation for Geometric Data, Proc, Int'l Conf. on Very Large Data Bases, Brighton, England, August 1987, 501.

Gunther 1989 : O. Gunther, The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Data, Proc, Int'l Conf. on Data Eng., Kobe, Japan, April 1991, 23.

Guttman 1982 : A. Guttman and M. Stonebraker, Using a Relational Database Management System for Computer Aided Design Data, IEEE Database Engineering 5, 2(June 1982).

Guttman 1984 : A. Guttman, R-Trees: A Dynamic Index Structure for Spatial Searching, Proc, ACM SIGMOD Conf., Boston, MA, June 1984, 47.

Henrich 1989 : A. Henrich, H. W. Six, and P. Widmayer, The LSD Tree: Spatial Access to Multi-Dimensional Point and Non-point Objects, Proc. Int'l Conf. on Very Large Data Bases, Amsterdam, The Netherlands, August 1989, 45.

Held 1975 : G.D. Held, M. Stonebraker, and E. Wong, INGRES - A relational data base management system. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., 1975, pp.409-416.

Hinrichs 1983 : K. Hinrichs and J. Nievergelt, The Grid File: a Data Structure Designed to Support Proximity Queries on Spatial Objects, Nr. 54, Institut fur Informatik, Eidgenossiche technische Hochschule, Zurich, July 1983.

Hoel 1992 : E. G. Hoel, and H. Samet, A Qualitative Comparison Study of Data Structures for Large Linear Segment Databases, Proc. ACM SIGMOD Conf., San Diego, CA, June 1992, 205.

Hutflesz 1988a : A. Hutflesz, H. W. Six, and P. Widmayer, Twin Grid Files: Space Optimizing Access Schemes, Proc. ACM SIGMOD Conf., Chicago, IL, June 1988, 183.

Hutflesz 1988b : A. Hutflesz, H. W. Six, and P. Widmayer, The Twin Grid File: A Nearly Space Optimal Index Structure, Lecture Notes in Comp. Sci. 303(April 1988), 352, Springer Verlag.

Hutflesz 1990 : A. Hutflesz, H. W. Six, and P. Widmayer, The Rfile: An Efficient Access Structure for Proximity Queries, Proc. IEEE Int'l Conf. on Data Eng., Los Angles, CA, February 1990, 372.

Jagadish 1991 : H. V. Jagadish, A Retrieval Technique for Similar Shapes, Proc. ACM SIGMOD conf., Denver, CO, May 1991, 208.

Kemper 1987 : A. Kemper and M. Wallrath, An analysis of Geometric Modelling in Database Systems, ACM Computing Surveys 19, 1(March 1987), 148.

Kolovoson 1991 : C. P. Kolovoson and M. Stonebraker, Segment Indexes: Dynamic Indexing Techniques for Multi-dimensional Interval Data, Proc, ACM SIGMOD Conf. Denver, CO, May 1991, 138.

Kriegel 1987 : H. P. Kriegel and B. Seeger, Multidimensional Dynamic Hashing is Very Efficient for Non-Uniform Record Distributions, Proc. IEEE Int'l Conf. on Data Eng., Los Angles, CA, February 1987, 10.

Lomet 1990 : D. Lomet and B. Salzberg, The hb-tree: A Multi-attribute Indexing Method with Good Guaranteed Performance, ACM Trans. on Database Sys. 15, 4(December 1990), 625.

Lomet 1992 : D. Lomet, A Review of Recent Work on Multi-attribute Access Methods, ACM SIGMOD Record 21, 3(September 1992), 56.

Neugebauer 1991 : L. Neugebauer, Optimization and Evaluation of Database Queries Including Embedded Interpolation Procedures, Proc. ACM SIGMOD Conf. Denver, CO, May 1991, 118.

Robinson 1981 : J. T. Robinson, The K-D-B Tree: A Search Structure for Large Multi-Dimensional Indices, Proc. ACM SIGMOD Conf., Ann Arbor, MI, April-May 1981, 10.

Samet 1984 : H. Samet, The Quadtree and Related Hierarchial Data Structures, ACM Computing Surveys 16, 2(June 1984), 187.

Seeger 1991 : B. Seeger and P. A. Larson, Multi-Disk B-trees, Proc. ACM SIGMOD Conf., Denver, CO, May 1991, 436.

Stonebraker 1983 : M. Stonebraker, et. al., Application of Abstract Data Types and Abstract Indices to CAD Data, Proc. Engineering Applications Stream of Database Week/83, San Jose, Ca., May 1983.

Stonebraker 1986 : M. Stonebraker, Inclusion of New Types in Relational Database Systems, Proc. 1986 IEEE Data Engineering Conference, 262-269.

Stonebraker 1988 : M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout, The Design of XPRS, Proc. Int'l Conf. on Very Large Data Bases, Los Angeles, CA, August 1988, 318.

Stonebraker 1990 : M. Stonebraker, L. A. Rowe, and M. Horohama, The implementation of Postgres, IEEE Trans. on Knowledge and Data Eng. 2, 1(March 1990), 125.

Srinivasan 1991 : V. Srinivasan and M. J. Carey, Performance of B-tree Concurrency Control algorithms, Proc. ACM SIGMOD Conf., Denver, CO, May 1991, 416.

Yuval 1975 : G. Yuval, Finding Near Neighbors in k-dimensional Space, Inf. Proc. Lett. 3, 4(March 1975), 113-114.

Wong 1977 : K. C. Wong and M. Edelberg, Interval Hierarchies and Their Application to Predicate Files, ACM Transaction on Database Systems 2, 3(September 1977), 223-232.