PROCEEDINGS

SIXTH ANNUAL

MICRO-DELCON '84

THE DELAWARE BAY COMPUTER CONFERENCE 1984

ISBN 0-8186-0554-5
IEEE CATALOG NO. 84CH2004-0
LIBRARY OF CONGRESS NO. 83-83444
IEEE COMPUTER SOCIETY ORDER NO.554

PROCEEDINGS
SEVENTH ANNUAL
MICRO-DELCON '84

THE DELAWARE BAY COMPUTER CONFERENCE 1984

Sponsored by
IEEE
Computer Society
Delaware Bay
Section

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS

MARCH 6, 1984
John M. Clayton Hall
University of Delaware
Newark, Delaware

IEEE COMPUTER SOCIETY PRESS

Published by IEEE Computer Society Press
1109 Spring Street
Suite 300
Silver Spring, MD 20910

ISBN 0-8186-0554-5 (paper)
ISBN 0-8186-4554-7 (microfiche)
ISBN 0-8186-8554-9 (casebound)
IEEE Catalog No. 84CH2004-0
Library of Congress No. 83-83444
IEEE Computer Society Order No. 554

Order from: IEEE Computer Society
Post Office Box 80452
Worldway Postal Center
Los Angeles, CA 90080

IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854

The Institute of Electrical and Electronics Engineers, Inc.

FOREWORD

MICRO-DELCON '84 is the Seventh Annual Conference sponsored by the Computer Society of the Delaware Bay Section of the IEEE. The meeting provides a one-day forum at which professionals from all areas related to computer science and computer engineering can exchange ideas.

The goal of our annual conference is to promote interaction among academic and industrial professionals in the Delaware Bay area. Traditionally, each of these sources is responsible for roughly one-half of the papers presented. In the last few years, the conference has begun to attract interest well outside of our local area, so that an area within a 200-mile radius of Newark is included.

We are fortunate to have two excellent keynote speakers. We are grateful to Dr. Henry Sowizral from the Rand Corporation (who is discussing the control of control) and to Dr. John Hutchison from the General Electric Corporation who is giving an up-to-date look at ADA.

The success of any conference is due to the authors and the referees; we gratefully acknowledge their contribution. The conference would not exist without the voluntary labor of all the committee members. We are also indebted to the staffs of the Departments of Computer and Information Sciences and of Electrical Engineering of the University of Delaware, whose help was essential to the success of the conference.

John T. Lund
Conference Chairman

A. Toni Cohen
Technical Program Chairman

MICRO-DELCON '84

SEVENTH ANNUAL CONFERENCE ON COMPUTER TECHNOLOGY

March 6, 1984

John M Clayton Hall - University of Delaware - Newark, Delaware 19711

CONFERENCE COMMITTEE

Conference Chair
John T Lund

Conference Vice Chair
Bobby F Caviness

Exhibits
Charles R Berg

Publicity
John S DeGood
Mark E Laubach

Membership
Nancy Carey

Technical Program
A Toni Cohen

Finance
Janet Lund

Registration
David M Robinson

Arrangements
W Barkley Fritz

PROGRAM COMMITTEE

Gonzalo Arce

Tim Finin

Allen Haley

Joseph E Lambert

Roger A Patterson

Donald Perlis

MICRO-DELCON '84

Table of Contents

Session III: Hardware System Implementations
Chair: Tuncay Saydem

Session IV: Tightly-coupled Parallel Architectures
Chair: Roger Patterson

Session V: Digital Signal Processing
Chair: David M Robinson

AUTHOR INDEX

A DATABASE CONSTRUCTION KIT

Mark B. Reinhold

University of Pennsylvania

Abstract

A programmer building a database is often faced with the choice between implementing the storage structures himself or using a general-purpose database management system. The former requires considerable effort. The latter may be even less desirable because DBMSs are expensive and unwieldy; moreover, they impose a data model on the programmer which may not suit the problem at hand. A Database Construction Kit has been designed and implemented that provides modular access to many techniques for the storage, indexing and retrieval of data. It is intended to be used in both small application programs only needing a random-access file, as well as in the implementation of complete personal database management systems.

Introduction

The Database Construction Kit is a file system specifically designed for use by applications which engage in any sort of database-like activity, be it simple random file access or the implementation of a database management system for a new data model. It provides a practical alternative to the usual choices of constructing one's own database mechanism or using a general-purpose database management system. The following principles were important in the design of the Database Kit.

- *Structural Extensibility.* An initial application of the kit will be to build a database management system for the functional data model. Central to the functional data model is the concept of the separate storage of data domains. To allow an efficient implementation of the functional model, such domains have been made inexpensive to create and destroy.
- *Reliability.* There is wide variation in the reliability of the file systems provided by general purpose operating systems. The kit does everything in its power to improve a database file's chances of surviving a machine crash, a power failure, or other minor disaster. Obviously, there are some events (e.g., a disk head crash) beyond the power of mere software.
- *Flexibility.* If some application requires a special-purpose storage or index mechanism, it can be integrated into the file system. If a particular application does not intend to construct B-tree indices, it does not have to carry around the module that implements B-trees.
- *Portability.* A significant amount of effort is involved in writing a file system, even a simple one which leaves the details of I/O to the host operating system. The code was written in as portable a style as possible, and no assumptions were made about esoteric capabilities of the host operating system.
- *File Access Efficiency.* Both the software and the file structure are designed to minimize the number of I/O operations required for any given task.

Concurrent database access is not provided. While the problem of synchronising concurrent database accessors so that each sees a consistent view of the database is well understood[7], the details of locking and shared file access vary so much from system to system that it is difficult to define a simple file sharing and locking mechanism that could be implemented on all systems. The kit is targeted for small but sophisticated "personal" databases and application programs for which shared access is not required. If absolutely necessary, concurrent access to a database could be provided by having all accessors interact with the file system through a common server process.

Database Abstractions

The Database Kit provides two low-level abstractions—*stores* and *indices*—from which an application builds its own higher-level structures. A store has the ability to:

- Save a datum and return an identifier marking its location;
- Retrieve a datum given its identifier;
- Update a datum in place, such that it is still marked by the same identifier;
- Delete a datum from storage.

Rather than operating on arbitrary data, an index deals with (*key*, *value*) pairs in the following ways:

- Insert a (*key*, *value*) pair into an index;
- Retrieve the *value*(s) corresponding to a given *key*;
- Delete a (*key*, *value*) pair from its index once it has been retrieved.

A database is a collection of stores and indices. A *storage method* is the program module which implements a particular kind of store; an *index method* is the module which implements an index. Collectively, the modules which implement stores and indices are referred to as the *access methods*.

Stores and indices are built within *segments*. A segment is a logical partition of the database; a different segment is used for each store or index. When a store or index is created, a *segment number* is returned which denotes the segment in which the structure resides. The segment number is used from that time onward to refer to the structure until it—or the database—is destroyed.

Stores

The simplest type of store is that for fixed length records. Such stores are easy to implement, but many types of data (e.g., character strings) come in varying sizes and should be stored so as not to waste space. Thus there are two types of stores: one for fixed-length records (record storage) and one for variable-length records (text storage).

The following operations are defined on stores.

create(*type*, ...) → *seg*
: Create a new store; *type* denotes the type of store to be created. Additional parameters may specify store-type-specific information such as the record length or the expected average character string length. *create* returns the segment number of the new store.

destroy(*seg*)
: Destroy the store residing in segment *seg*.

put(*val*, *seg*) → *id*
: Store the value *val* in segment *seg*, returning a storage identifier with which it can subsequently be retrieved.

get(*seg*, *id*) → *val*
: Retrieve the value denoted by identifier *id* in segment *seg*.

delete(*seg*, *id*)
: Delete the value denoted by identifier *id* in segment *seg*.

update(*val*, *seg*, *id*)
: Replace the value denoted by identifier *id* in segment *seg* with the new value *val*.

The storage identifiers associated with stored values are *magic*. They may be freely passed around, stored away in other stores or indices, or even converted to hexadecimal and printed to the terminal, but no interpretation whatsoever may be placed upon them. This policy ensures that no application will take advantage of knowledge about the implementation of a storage method, thereby becoming dependent upon that particular implementation of that particular storage method. The single exception to this rule is that storage identifiers from the same store may be compared with one another. The property that all storage identifiers for a given store are *unique* is often useful when composing larger structures.

Indices

There are more types of indices than stores. The present implementation offers B-trees[2] with modifications as suggested by Knuth[10], extendible hash indices[6], and associative string spaces. The type of index to use for a given application is determined by the type of the key data, the way in which the (*key*, *value*) pairs are to be retrieved, and the space versus time tradeoff. The space-time tradeoff is often the most important; generally, the faster an index can be searched, the more space it occupies and the more space it leaves unused.

In order to build an index and retrieve data from it, an index method must have some way of interpreting key data so that keys can be compared; storage methods have no need to interpret the data they handle. All data passed between the application and the file system for storage or retrieval purposes are in the form of *byte strings*. Byte strings are dynamically typed; i.e., part of their value denotes the type of the data in the string so that it may be compared with other strings of the same type. The other components of a byte string's value give the length of the string and a pointer to the string itself.

Index methods differ in how they allow (*key*, *value*) pairs to be inserted. There are three possibilities for the *insertion mode* of an index:

- Duplicate keys are not allowed;
- Duplicate keys are allowed; or
- Duplicate keys are to be overwritten.

In the last case duplicate keys are not allowed, and inserting a (*key*, *value*) pair whose *key* is already in the index will result in the automatic deletion of the old pair. Some indices support all insertion modes; some allow only one.

The insertion mode of an index is specified when the index is created, along with the type of the keys of the index and the type of the index itself.

create(*type*, *mode*, *keytype*, ...) → *seg*
Create an index. *type*, *mode*, and *keytype* denote the type of the index, its insertion mode, and the type of keys in the index, respectively. Additional parameters may specify index-type-specific information such as the maximum key length.

Unlike operations on a store, some of the operations on an index must retain state information from one call to the next. Furthermore, most of these same operations must return a value indicating the success or failure of the file system to service the request. *Cursors* solve both of these problems. A cursor has two components: a *status* and a *position*. The status component indicates the success or failure of an operation and provides some additional status information. The position component marks the position of a (*key*, *value*) pair in an index. Applications may access the status value, but—like storage identifiers—the position information in a cursor is magic. Some operations such as *insert* return cursors only to indicate a status; the position part of such a cursor is meaningless.

insert(*key*, *val*, *seg*) → *cursor*
Insert the given (*key*, *value*) pair into the index in segment *seg*. The returned *cursor* reports the status of the insertion operation. The type of *key* must agree with the key type given when the index was created. The type of *val* is irrelevant.

A second distinctive characteristic of the index methods is the manner in which keys may be retrieved. The key being searched for is referred to as the *target key*. All index methods allow a *literal match*: if a retrieve operation is performed for a target key which exactly matches a key in the index, the search is successful and the corresponding (*key*, *value*) pair is returned. The initial retrieve operation is named *first*, since it returns the first matching pair.

first(*seg*, *key*) → *key* × *val* × *cursor*
Find the first (*key*, *value*) pair in segment *seg* whose key matches the target *key*. The type of *key* must agree with the key type given when the index was created. The returned triplet contains the actual key (since the match may not have been exact—see below), the corresponding value, and a cursor which reports the status of the retrieval and marks the position of the (*key*, *value*) pair in the index for purposes of *delete* or *next*.

If there is more than one matching pair, the remaining pairs may be obtained by means of the *next* operation.

next(*cursor*, *key*) → *key* × *val* × *cursor*
Find the (*key*, *value*) pair whose key matches the target *key* and which follows the pair whose position is marked by the given *cursor*.

A table is an example of a literal-match index. A table is an array: the key of a pair is the unsigned integer index of the value in the array. Table indices do not allow duplicate keys, since there can be only one array element for a given array index.

There are two variations on the literal match: the *prefix* match and the *pattern* match. A prefix match is successful if a key in the index has the target key for a prefix. It is possible that a key is exactly equal the target key; the status value of the cursor is used to distinguish between exact and prefix matches. Any additional keys that match the prefix may be obtained with *next*. If there are a number of exact matches followed by a number of prefix matches, the exact matches are returned first. A B-tree is a prefix-match index.

A pattern match allows a search for some pattern. A pattern defines a (possibly infinite) set of strings, which set the key must be a member of in order for the match to be successful. A set of strings which denotes the set of all strings containing those strings is a simple kind of pattern; a regular expression[18] is a more powerful kind of pattern. Pattern-matching indices allow rich and complex key specifications at the expense of being much slower than other index methods. An example of a pattern-matching index is an *associative string space*, which stores (*key*, *value*) pairs in random order and retrieves them by scanning the index from beginning to end.

Together, the *first* and *next* functions can be used to retrieve the sequence of (*key*, *value*) pairs whose keys match a target key, but nothing has been said about the order of that sequence. The *natural order* of an index is defined as an ordering of the (*key*, *value*) pairs of an index which is some function of the order in which the pairs were inserted into that index. The natural order may be by key value, as in a B-tree, or it may be random, as in a hash index that uses a non-order-preserving hash function. When a sequence of (*key*, *value*) pairs is obtained from an index with *first* and *next*, the order of the sequence is the natural order of the index.

An index may be completely traversed by using a null key. By proclamation, a null key will always match any key in an index, so performing a *first* operation with a null target key should give the very first (*key*, *value*) pair in the index. Then *next* operations can be performed, using the same null key, until the end of the index is reached. A complete traversal yields all the (*key*, *value*) pairs in the index in their natural order.

A (*key*, *value*) pair may be deleted using the cursor returned by *first* or *next*.

delete(*cursor*) → *cursor*
Delete the (*key*, *value*) pair denoted by *cursor*.

In general, a *delete* operation implicitly invalidates all cursors on the index to which it is applied, including the cursor returned by *delete*. However, some index methods allow a *delete* to be followed by a *next* on the resultant cursor. All other cursors become invalid, but the cursor returned by *delete* is adjusted to reflect any changes in the internal structure of the index. The status value of a cursor returned by *delete* is irrelevant; delete operations either succeed or abort.

Not all indices fit neatly into this model. Therefore a *special* function is introduced which is used as an escape for those operations not yet described.

special(*seg*, *request*, ...) → *cursor*
Perform some special operation, denoted by *request*, on the index in segment *seg*. The value of the returned *cursor* may or may not be meaningful.

One use of the *special* function is with a table index. In some applications the actual position of an element in the array is unimportant, but all the used positions must be packed down at the bottom of the array, i.e., sequentially numbered from 0 to n with no gaps. When inserting a new element, the key of the first unused position is found with the *special* function and then used as the key for the insertion operation.

Databases

Finally, there are functions to manipulate the entire database. Initially a database must be created or opened.

create(*name*) → *fullname*
Create a new database with the given *name*, returning its full name.

open(*name*, *mode*) → *fullname*
Open an existing database of the given *name*, returning its full name. *mode* indicates whether the open is for read-only or for read/write operations.

When the various storage and index methods are called to update the database, the database file is not actually modified until the current transaction is committed. A transaction is a set of database operations terminated by *commit*. The file system does the work of guaranteeing that transactions are *atomic*; an atomic transaction is one that either runs to completion or has no effect on the database at all[11]. Until a transaction is committed, there is no visible record in the database file that it was ever started, so if the system suddenly crashes, the database is left in a consistent state.

commit
Declare the end of a transaction. When *commit* returns, any changes to the database have been irrevocably committed.

A transaction may be a*borted* at any time before the commit point, either by the application (e.g., in case of operator error), by the database file system itself (e.g., in case of a shortage of some resource such as disk pages), or inadvertently (e.g., in case of a machine crash). Of course in the latter case, nothing will be around to call the *abort* function. All of the functions introduced in this section will trigger an internal abort if something goes wrong unexpectedly.

abort
Abort the current transaction. Any pending database updates are thrown away and forgotten.

When the application is finished with the database, the database file should be closed.

close
Close the currently open database.

Using the Kit

In order to provide a more intuitive feel for how the Database Kit is used, this section discusses how one could implement database management systems for the the functional, relational, and network data models using the database abstractions just presented.

The Functional Data Model

The functional data model describes information in terms of entities and functions that define mappings between entities[16]. For example, in a university database a particular student and the courses he is taking may be modelled as a function mapping a student entity to a set of course entities.

Functions exist in several varieties. Functions may be defined with zero or more parameters, and functions may return a single value or a (possibly empty) set of values. A function that returns a singleton is expressed with a single arrow (→), while a function that returns a set is expressed with a double arrow (⇒).

A special function is defined for each type of entity: the *generator function* for an entity type has no parameters and returns a set containing all the entities of that type in the database.

Here are some sample functions on the university database:

person() ⇒ *entity*
name(*person*) → *string*
course(*student*) ⇒ *course*
grade(*course*, *student*) → *integer*

The database must contain some metadata—data describing the structure of the rest of the data—so that the functional DBMS can figure out how to evaluate functions. For each function, a *function descriptor* contains the following items: how the function is stored, what segment(s) it is stored in, the number of parameters it takes and their types, the result type of the function, and whether the function returns a single value or a set. Two segments are used for the metadata. A record store contains the function descriptors, and an extendible hash index maps function name strings into the storage identifier of the corresponding function descriptor.

The type of an entity is denoted by the storage identifier of the function descriptor for its generator function. Each entity within a type is assigned an instance number unique to that entity type. Generator functions are stored in a table index whose value size is zero; the table is used only to uniquely identify entities—by their instance numbers—and to traverse that set of instance numbers. A table with a zero value size is just a bitmap, which is a very efficient way of storing a generator function. When a new entity is created, the key of an empty table position is obtained with the *special* function.

A 1-to-1 function is stored in a table whose key is the domain and whose value is the range. A 1-to-n function is stored in two segments. A text store holds the range set of the function; a table maps the domain entity instance number to the storage identifier of the corresponding range set. An n-to-1 function is stored in an associative string index with the domain tuple as the key; a table would be inappropriate here as such functions tend to be very sparse. If fast access were absolutely necessary, a B-tree or extendible hash index could be used instead. n-to-n functions are implemented in a manner similar to 1-to-n functions, except that an associative string index is used instead of a table to map domain tuples into range set storage identifiers.

Only small values such as entity numbers and integers can be directly passed around as function values. Character strings are treated in the following way. A global string space is maintained in a text segment and in a B-tree; the B-tree maps strings into string numbers, and the text segment maps string numbers back into strings. Functions dealing with strings refer to strings in these segments. This approach would be duplicated for any other data types with large values.

The Relational Data Model

See Date[4] for an introduction to the relational data model. Assume for simplicity that tuples in a relation are of fixed length. A single relation may be stored in a table index, one tuple per record, where the key of each record is its row number in the relation. An index is used here instead of a store because it allows an efficient traversal of the rows of the relation. An index on a relation can be constructed in a B-tree or extendible hash segment. Key uniqueness may be enforced by creating the index to not allow duplicate keys.

The metadata required for a relation includes the number of the table segment containing the relation, the number of fields in each tuple and the type and length of each field, and any information required to link a relation with its associated indices. A descriptor record for each relation is kept in a record store, while an extendible hash index provides a mapping from relation name to relation descriptor record storage identifier. The descriptor record may be extended to describe any integrity constraints applicable to the relation.

The Network Data Model

Again, Date[4] provides a readable presentation of the network (CODASYL) data model. A simple implementation of the network data model would use a record store for each record type and then construct the traditional pointer chains from record to record within each set. However, a design which takes more advantage of the file system's capabilities is possible.

A record store is used for each record type. A table index is used to store descriptor records for each record type containing the number of the record segment, the type and length of each field, and any constraints that apply to the record (e.g., a field must be unique within a set).

A set is implemented by means of two segments. A table index maps an owner record number to a member set number, and a text segment maps the member set number to a set, which contains zero or more member record storage identifiers. Since record numbers only identify records within a segment, a schema table must be kept describing each set in terms of its owner record type, member record type, and the numbers of the segments containing the owner-set table and the sets themselves. Indices that provide fast access from key to record may be constructed on B-trees or extendible hash segments as necessary.

A Real Application

The first application of the Database Kit was to construct a program to maintain a database of bibliographic refer-

ences in the style of REFER[13]; that program was used to prepare the references for this paper.

The structure of the bibliographic database is as follows. Each entry consists of one or more fields of text (author, title, date, etc.). The keys for an entry are concocted from all words in that entry; keys which are numbers outside of the range 1800-2000 (i.e., not viable year numbers) are ignored, as are keys which are on a list of the 100 most common English words. All keys are stored in lower case and are truncated to twenty characters. A database contains the following segments:

entid → *entry*	text store; contains actual entries
entsetid → {*entid*}	text store; contains sets of entry ids
key → *entsetid*	B-tree; maps key to entry set id
∅ → *entid*	table of all entries

Entries are kept in a text store. Since a key may belong to more than one entry, the value of a key in the key index is the storage identifier of a text string containing one or more storage identifiers for the entry store. A table is used to keep track of all entry identifiers so that all the entries in the database can be produced, e.g., in order to unload the database into a text file.

A bibliographic citation is specified by giving a set of keys for the entry. For example, volume one of Knuth's *The Art of Computer Programming* series could be cited with the keys "*Knuth Art Computer 1973.*" Each key is retrieved from the *key* → *entsetid* index; each of these *entsetids* produces a set of *entids*, the intersection of which produces the set of storage identifiers corresponding to the desired references. The file system performs quite well in this application; it takes approximately 0.15 CPU seconds and only 9 disk reads for the program look up all the references in this paper from a database of about 100 entries.

Implementation Overview

This section briefly describes how the implementation of the Database Kit meets its design goals. A soon-to-appear technical report[15] gives extensive details on the implementation.

Software Structure

The initial implementation of the database kit was done on a VAX under the VMS operating system and the Eunice UNIX* emulator[8]. It is written entirely in the C programming language[9] according to the UNIX version 7 standard. The implementation depends upon no special features of the VAX, so it could be ported to a PDP-11 with minimal effort. High quality compilers are now becoming available for larger microcomputers (e.g. 68000-based systems), extending the set of machines to which this implementation could be transferred. Of course, it will run on any VAX with DEC's VMS operating system.

*UNIX is a trademark of Bell Laboratories.

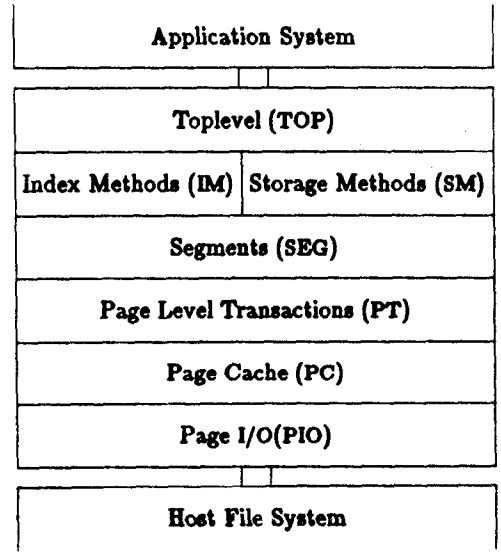

Figure 1. Software Layer Structure

The file system is organised as a set of layers, one on top of the other. Each layer provides some abstraction in the form of a set of functions and procedures to the layer above it, and uses the abstraction provided by the layer below it to construct its own abstraction. Externally, a layer is defined only by the services it implements and the services it uses; the implementation of the layer is completely invisible to the outside world. The independence of interface and implementation allows different versions of a layer to be used in different versions of the file system. Figure 1 depicts the overall structure of the file system.

The PIO layer uses the host's file system to provide the abstraction of a file of pages numbered from zero. Under VMS, this is a fairly direct mapping to the file I/O services of the RMS record management services component[5]. To maximise portability, all that is assumed of the host operating system is that it can write disk pages atomically, i.e., a write operation either completes successfully or does nothing at all.

The purpose of the page cache layer is to reduce file activity by keeping the most frequently referenced database pages in memory. The abstraction provided by the PC layer is that of a set of page *frames*, each of which contains a single page of the database file. The number of page frames may be varied by means of a compile-time constant. The PC layer uses a modified least-recently-used (LRU) algorithm to select which pages are to be kept in the cache. The LRU algorithm takes advantage of additional knowledge about how a page frame is intended to be used, as suggested by Stonebraker[17]. When a page is requested, information such as the manner in which the caller intends to access the page in the near future and

whether it will be written or not is used to determine the length of time for which the page will remain in the cache.

The page-level transaction layer handles the database update process that takes place during a transaction and is finished up at transaction commit. It is also responsible for maintaining the freelist—the list of unused pages in the database file—during a transaction. Further discussion of this layer requires knowledge of the file structure, and will be postponed until the next subsection.

The segment layer uses the pages provided by the PT layer to build database segments. A segment is a set of pages arranged in a linear address space that is not necessarily contiguous; i.e., some page numbers may not map to valid disk pages. Each page contains a fixed number of bytes. The storage identifier for a particular byte is composed of the number of the page within the segment and the offset of the byte in that page. Each segment has a *type* attribute which describes the type of structure it contains. This model is much like that of the Relational Storage System used in the System R relational database management system[1].

The access method layers, IM and SM, use the segments provided by the SEG layer to build stores and indices. Unlike most layers of the system, the IM/SM layer consists of several modules. One module is present for each access method in use; two additional modules contain supporting code for segment space management and other utilities such as byte string comparison and hashing.

The outside world only calls functions in the toplevel. The functions available to the application program correspond directly to those outlined in the section on database abstractions. The functions for manipulating indices and stores are *overloaded*: there is one set of functions for all operations upon stores and one set of functions for all operations upon indices. For example, there is a single function to delete a (*key*, *value*) pair from an index given a cursor obtained from *first* or *next*, which function is used on each and every index when that operation is required. The toplevel layer determines which access method module to call by examining the type attribute of the segment in question.

The toplevel layer is also responsible for handling and reporting transaction aborts. If the host operating system provides the required services, the toplevel layer protects against internal errors in the file system code by catching system error traps and using them to initiate internal aborts. Information about a transaction abort is made available to the application through two global variables and a user error trap function.

File Structure

In order to implement efficient and independent segments, simple atomic database updates, and to take advantage of locality of reference within a segment, the database file is structured as a tree (see figure 2). At the lowest level (the PIO layer) the file appears as a collection of sequentially numbered pages. The next two layers of the file system (PC and PT) transform this view into a tree of pages, with the seroth page as the root. The root page points to a set of subtrees, each of which constitutes a segment. The root of each segment subtree is the *segment header*, which contains the number of pages allocated to the segment, the type of the structure stored in the segment, and direct and indirect mapping pointers to the data pages in the segment.

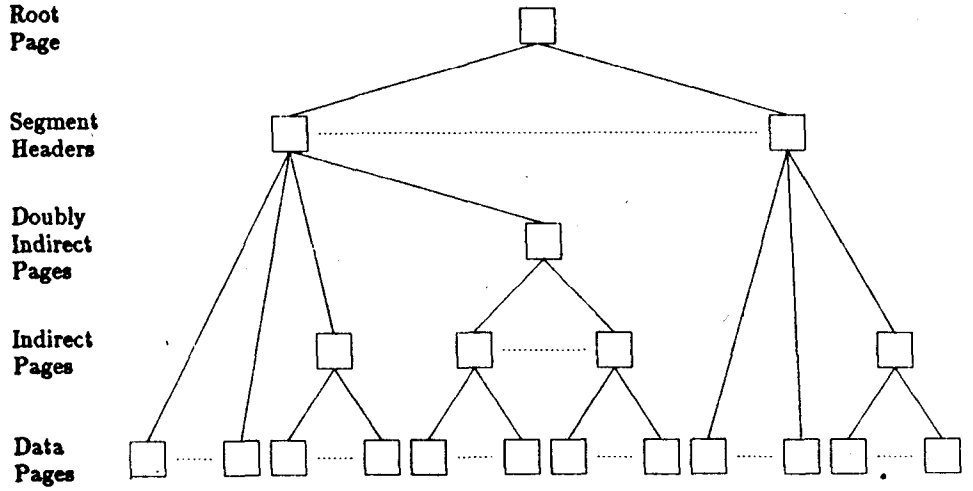

Figure 2. Database File Structure

Several attributes of the file structure are variable and may be changed by means of compile-time constants. This allows a certain amount of "fiddling" to tune the system for particular applications. The most important attribute is the file page size, which affects all other file structure constants. Pages are typically between 2^9 and 2^{12} bytes long and are required to be integer multiples of the disk page size of the host operating system. No part of the file system depends upon a particular page size; the current implementation has been verified to work with page sizes of 512, 1024, and 2048 bytes. Pages within a segment are denoted by *segment page numbers*, which are typically two or three bytes long, allowing from 2^{16} to 2^{24} pages in a segment. The number of pages denotable by a segment page number may be smaller or larger than the structural limit of the segment dictated by the page size constant.

Let s be the file page size measured in units equal to the size of a segment page number. If i units are required to store other information, then the segment header consists of $s - i$ mapping pointers. All but two of the mapping pointers point directly to data pages; thus the first $s-i-2$ data pages may be accessed directly from the segment header page. The segment header points to one indirect mapping page, which points to s data pages; the segment header also points to a doubly indirect mapping page, which points to s indirect mapping pages, each of which in turn points to s data pages. In terms of the page size s, a given segment may contain $s^2 + 2s - i - 2$ data pages. For a page size of 1024 bytes and a segment page number size of two bytes, $s = 512$, so if two units are taken up for other information ($i = 2$), the segment capacity is 263164 pages (269 megabytes). In this case, the structural limit is greater than the limit imposed by the size of the segment page number, so each segment in fact would be limited to 65535 data pages, or 67 megabytes.

Since segments are naturally subtrees of the tree structure of a database file, they incur little extra overhead of themselves. It is not unreasonable to consider a database of hundreds of small segments, each containing a single data domain for some data model. Segments can be created and destroyed by the SEG layer independently of the data structures built within them. This separates segments from the structures they contain and isolates each segment from all others in the database, reinforcing one's faith in the file system. A bug in the module that implements B-tree indices may cause some random page to be overwritten, but only in the segment containing the faulty B-tree; other segments will be left alone.

The tree-based file structure helps reduce file activity by decreasing the access cost for a page as locality of reference within a segment increases. n references to a small area within a segment are cheaper than n widely spaced references in a segment, which in turn are much cheaper than n references to entirely different segments. There is also a bias in favor of segments whose data is clustered at the bottom end: references to the first $s - i - 2$ pages in a segment will be cheaper than references to the next s pages, which in turn are cheaper than references to the remaining s^2 pages. This is due to the intentionally slightly lopsided mapping scheme described above.

The index and storage methods can improve locality of reference through the use of *near-mode allocation*. When a new page is created in a segment, the caller can specify the page number of an existing page. The SEG layer tries—but is not guaranteed—to allocate the new page as near as possible to the old page. Near-mode allocation only works in terms of the internal segment structure; it does not attempt to provide physical contiguity in the disk file.

Another advantage of the tree-based file structure is that it admits a simple algorithm for performing completely atomic updates. When a segment page numbered d is to be written, a function in the SEG layer recursively descends the tree, beginning at the root. When page d is reached, a call to the PT layer finds an unused file page for it and writes it to that page through the cache. The PT layer returns the number p of the file page assigned to the segment page. The pointer to the old copy of d in the page immediately above is modified to point to its new location in the file, namely the file page numbered p.

The recursion unravels until the root page is reached. The appropriate pointer in the root page is modified, but the root page is not written until the transaction is committed. Since disk write operations are assumed to be atomic, the final action of writing the root page—always back to file page zero—updates the entire file in a single atomic operation. Until the modified root page is written, all newly written pages are still in the freelist. If a transaction aborts before the root page is written, the database is left in a consistent state. This scheme was inspired by Paxton's transaction mechanism for distributed file systems[14].

Just before the root page is written to commit the transaction, the freelist is updated. As pages are allocated and freed during a transaction, a list of *recently freed* pages is built using pages still in the freelist. A separate list of recently freed pages is required because a page freed during a transaction cannot be re-used in the same transaction. If it were rewritten and the transaction were aborted then the database would not be left in a consistent state. At commit time, the recently freed page list is merged with the old free page list to form a single new freelist. Because the freelist update takes place immediately after the rest

Table 1. Index Method Performance

Index Method	A-string	B-tree	ISAM
Construction			
disk writes	1512	5023	5082
CPU time	67.95	113.17	32.80
page faults	597	578	343
file size	228	322	396
space utilization	96.9%	70.1%	46%
Retrieval			
disk reads	109	118	180
CPU time	70.71	2.61	4.63
page faults	38	45	5
cache hit ratio	99.7%	95.7%	—

of the file pages are updated and just before the root page is written to commit the transaction, aborting the transaction will cause the file pages containing the recently freed page list to remain in the old freelist.

Performance

Each storage method was tested by randomly executing put, get, update, and delete operations on a large fixed quantity of data. Both the record and text storage methods averaged between 80 and 100 operations per second.

Each index method was tested by building an index on the first 5000 words in the system spelling dictionary. The fifteen character key is the word itself; the four character value is the word's index in the dictionary. 250 keys known to be in the index were retrieved in random order. The results of this test are presented in table 1.

For comparison, the same tests were performed using the indexed sequential (ISAM) file facility provided by the record management services component of VMS[5]. While VMS ISAM consumed less CPU time in constructing the index, it produced a file that was noticeably larger and more than half empty. The B-tree index method performed significantly better than VMS ISAM in the retrieval test in terms of both CPU time and disk reads. The lower page fault statistic for retrieval from the ISAM file is explained by the fact that the retrieval was done in the VMS command language rather than by an explicit program. The command language interpreter is permanently resident in memory, while a program must be paged in from disk as it runs.

The extendible hash index method was not evaluated in this test. The implementation is not as efficient as was hoped for; it produced a file of over 1000 pages for the test data. Extendible hashing works best with large hash buckets; i.e., on the order of 2^{13} bytes. In this implementation a bucket is a single page (2^{10} bytes), and the directory is split many more times than it ought to be just in order to make more space in the index. The retrieval performance of the extendible hash index method is adequate, but the insertion performance and space utilisation are unacceptable.

Conclusion

Aside from the poor performance of the extendible hash index method, several other problems require attention. Currently, the module in the IM/SM layer which allocates variable-length byte strings in a text store cannot handle strings longer than the length of a database page. This is not too serious a limitation, but some facility should be provided for storing large contiguous chunks of data.

Two minor enhancements would significantly improve the performance of the SEG layer. When new pages are created in a segment, free page positions are found by recursively traversing the map pages of the segment. Typical segments will have only a few map pages anyway, so the search is not overly expensive. However, several page accesses could be saved by using a bitmap to keep track of which pages in the segment are defined and which are undefined. This strategy would only cost 8 pages (of 2^{10} bytes each) for a completely full segment and would make the page allocation process much more efficient, especially in large segments.

The second enhancement is to provide some space in the segment header structure for use by the access methods. Each access method must store some information about how it structures data in a segment; typically this is done by reserving the zeroth page of the segment for that purpose. But pages are large and usually only ten or twenty bytes of data are involved. It would be more efficient to allow access methods to store and retrieve an *access method header* which is maintained in part of the segment header. This would reduce the number of direct mapping pointers that can be stored in a segment header, but the loss would be repaid by not having to access an additional page for necessary structural information.

At the moment, alphabetic case is significant in all the index methods which support character string keys. It is not clear whether ignoring case for character string keys in indices is a desirable thing or not; for many this is a religious issue. This problem should be thought out and a way to enable and disable case folding should be implemented.

The single pattern-matching index method, the associative string space, supports only a limited sort of pattern. Full regular-expression pattern matching[18] would significantly extend the power of associative string indices. One problem lies in implementing a regular-expression search