DESIGN AND IMPLEMENTATION OF THE WISCONSIN STORAGE SYSTEM

by

H-T Chou, David J. DeWitt,
Randy J. Katz, and Anthony C. King

Computer Sciences Technical Report #524

November 1983

Design and Implementation of the Wisconsin Storage System

H-T Chou
David J. DeWitt
Randy H. Katz+
Anthony C. Klug

Computer Sciences Department
University of Wisconsin - Madison

This research was partially supported by the National Science Foundation under grants MCS81-05904, MCS81-02864, MCS82-01870, and MCS82-01860, the Department of Energy under contract #DE-AC02-81ER10920, and by a grant to the Computer Sciences Department from IBM.

+ Current Address: Computer Science Division, E.E.C.S. Department, University of California at Berkeley

DEDICATION

This paper is dedicated to the memory of our colleague Tony Klug who was killed in a bicycling accident in June 1983. Tony was the inspiration of the Wisconsin Storage System project. It is tragic that he never had a chance to use the fruits of his ideas.

ABSTRACT

We describe the implementation of a flexible data storage system for the UNIX environment that has been designed as an experimental vehicle for building database management systems. The storage component forms a foundation upon which a variety of database systems can be constructed including support for unconventional types of data. We describe the system architecture, the design decisions incorporated within its implementation, our experiences in developing this large piece of software, and the applications that have been built on top of it.

1. Introduction

The past decade has seen significant advances in database system technology. Relational database systems such as INGRES from Relational Technology Incorporated and SQL/DS from IBM and database machines such as the IDM 500 from Britton-Lee are now commercially available. The operating system community [REED83] has embraced the transaction concept ("atomic actions"), originally conceived in the context of database systems. Database management has become an important subsystem of many computing environments, from highly available transaction processing systems to personal computers.

New database techniques for query processing, concurrency control, and recovery have been invented, developed, and embodied in real systems. Initial performance evaluations of working systems are now becoming available [BITT83]. As the field has matured, the research community has begun to search for new applications for database technology. Commercial databases typically store data about inventories, accounts payable/receivable, banking accounts, etc. More unconventional applications include: geographic/geometric databases for cartography and solids modeling, image and picture databases for scene understanding and analysis, statistical/scientific databases for survey or experimental results, design databases for computer-aided design and programming environments, and office databases for documents. They differ from conventional databases in their data access and handling requirements. In addition, an active area of research is concerned with providing data management facilities for these applications in a network of processors or workstations.

The Wisconsin Storage System (WiSS) has been designed as a flexible data storage component upon which to build experimental database systems, either to store data for these new applications or to provide database services in a network environment. We have found that existing database systems are too difficult to modify and are not well suited for the types of applications with which we intend to experiment. While providing many of the same facilities as the Research Storage System of the System-R relational database system [GRAY78], WiSS has been implemented for the UNIX environment. It supports high performance sequential and indexed access to data stored on disk, through a relatively low level manipulation language.

This paper is organized as follows. In the next section, we describe the goals of the system.

Section 3 describes the system's architecture: physical I/O layer, buffer manager, storage structures, access methods, and concurrency control and recovery subsystems. In Section 4, we describe our experiences in designing and building the system. Section 5 compares the performance of WiSS with that of several other database systems. We then discuss how WiSS has been used to date. Section 7 gives our current status and future plans.

2. Implementation Goals

The principal objective of WiSS was to design and implement a storage system that could serve as a flexible testbed for experimental database system implementation. In particular, we wanted a vehicle for teaching students about implementing database systems and to explore ways of extending database techniques to unconventional data. For example, to handle the data found in unconventional databases, we needed to support long records that could span physical disk pages. Database systems normally restrict records to fit on physical pages, thus making it complicated to support variable length data such as text or images.

Very high performance has been a goal from the outset. The UNIX File System [THOM78], while simple and flexible, was not designed for high performance. Database systems built on top of UNIX have suffered reduced performance (see [STON81, WEIN82]) because the file system is organized around small pages and does not cluster file extensions. The file system spends significant amounts of time seeking to data rather than effectively transferring it. To avoid these problems, we have implemented an extent-based file system on top of a "raw" disk, effectively circumventing the UNIX file system. WiSS guarantees that files are allocated in physically sequential extents. The implementation on a raw device allows WiSS to manage its own buffers without the additional overhead of copying to and from UNIX buffers.

Since not every UNIX site has a raw disk available, it is fortunate that Version 4.2 of UNIX provides an extent based file system [MCKU82]. While this renders some of our own work redundant, it makes the upper levels of WiSS more transportable to other UNIX sites.

WiSS supports sequential files and B-tree indices. Indices map field values into a homogeneous collection of records with that value. A link structure was designed but never implemented. Links are

inter-file access methods, and map a record in one file into a heterogeneous collection of records in other files. They can be used to implement a CODASYL owner-coupled set.

Finally, WiSS has been designed to serve as a shared storage component for a network of workstations and other processors. In this capacity the system functions as a file server as well as a database server.

2.1. Comparison with Other Systems

The design of WiSS was influenced by the design of System R's Research Storage System (RSS) [ASTR76], FLASH [ALLC80], and PLAIN [KERS81]. In fact, except[1] for the addition of long data items, the capabilities provided by WiSS are almost identical to those of the RSS. However, the RSS is not available outside of IBM and thus was not available for our use. FLASH is a portable file system written in PASCAL. While similar in spirit to WiSS, FLASH lacks a number of capabilities provided by WiSS. These include:

(1) Support for complex scans
(2) Multiway, external merge sort operation
(3) File access a "block at a time" in addition to a record level interface.
(4) Support for long data items
(5) Support for raw disk drives.

Finally, while the structure of the PLAIN database handler is similar to WiSS, PLAIN was designed to only support relational database systems. In addition, PLAIN runs on top of and not in place of the UNIX file system.

3. Structure of the System

As shown below in Figure 1, WiSS consists of four distinct layers. The lowest level, level 0, deals with the aspects of physical I/O. It has been designed for operation with a raw disk device, although in the debugging stages of the implementation effort we interfaced it to the UNIX file system. The next level, level 1, is the buffer manager, and uses the read and write operations of level 0 to provide buffered I/O to the higher levels. Level 2 is the storage structure level. This level implements sequential

[1] In parallel with the implementation of WiSS, the RSS was extended to handle data items longer than one page in length [HASK82].

files, B^+-trees, and long data items. This level is also responsible for mapping references to records into the appropriate page references which are then buffered by level 1. Finally, level 3 implements the access methods, which provide the primitives for scanning a file via a sequential or index scan.

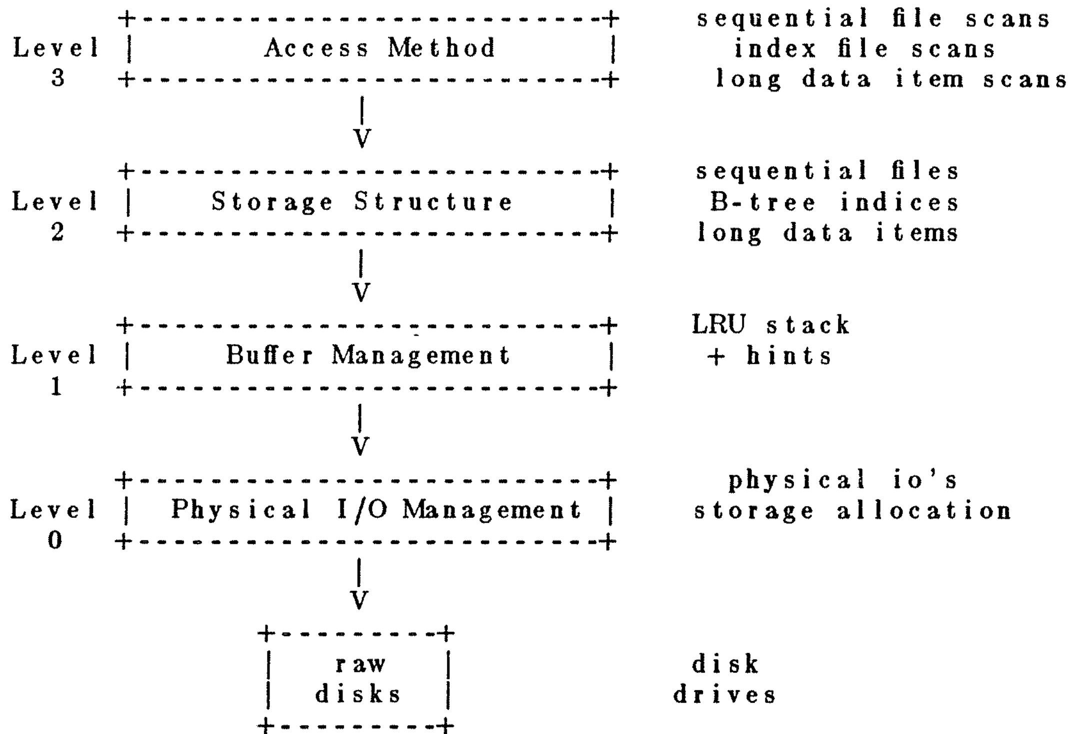


Figure 1
Architecture of WiSS

3.1. Physical I/O Layer

3.1.1. Introduction

Level 0 manages physical disk devices and schedules all I/O activities. Since files are a collection of sequential page extents, level 0 is responsible for allocating extents to files, and for managing free space on each volume.

The storage model of WiSS has volumes (disk packs, tapes, etc.) mounted on devices (disk drives, tape drives, etc.). Higher levels of WiSS are insulated from device details. Each volume has a unique volume identifier, thus allowing a volume to be moved between devices.

To simplify matters, the original level 0 was implemented on top of the UNIX file system. In this implementation a volume is assigned to a single UNIX file. The file is artificially divided into pages, which are grouped into extents.[2] A volume is identified by the higher levels of the system by a unique Volume ID, maintained at level 0, which maps the volume into the file that implements it. When a volume is first created, its UNIX file is formatted with extents. WiSS files on the same volume are allocated extents from within this UNIX file.

The implementation of level 0 on top of a raw device is remarkably similar. A raw device is treated as a "character special file". There is a 'raw' interface on UNIX that provides for direct transmission of blocks between the disk and the user's buffer. The block I/O's of a "raw file" bypass the physical address translation and block buffering of the file system, and thus avoid the overhead of the UNIX file system. To deal with a 'raw device', we treat it as a UNIX file, which is referenced by a system-defined name[3]. The system calls (e.g., read, write, lseek, etc.) are analogous to those for conventional files, with some restrictions on the number of bytes that can be transferred. The extent structure must be carefully matched to the physical properties of the real device, such as the number of blocks per track.

3.1.2. Physical Organization

Each volume is divided into a set of contiguous groups of pages termed extents. The first extent is used to contain a two page Volume Header. The first page of the Volume Header contains identifying information for the volume: the Volume ID, a number uniquely identifying this volume, and the Volume Title, a string describing the volume. In addition, the first page contains information describing the physical structure of the device: the size of an extent (in pages), the number of extents on the device, a bitmap of free/used extents, a count of files, and a remapping list for bad pages on the device. Page one of the Volume Header, also contains the address of the first page of the level 2 data structure "FileDir". FileDir is used by level 2 for mapping file names into file numbers. FileDir is always the first file (file 0) on a volume.

The second page of the Volume Header contains logical information for the device, namely

[2] Since logically contiguous pages in 4.1 UNIX are generally not physically contiguous, the pages within an extent in this case will not be physically contiguous

[3] For a disk interfaced by the MASSBUS, the name of the raw files on it is referred to as 'dev/rhp[0-7][a-h]'.

descriptors for all files on this volume. Each file descriptor specifies the list of extents currently allocated to the file and an extent file factor. When a file is created it receives an initial allocation of extents each of which consists of physically contiguous pages. When the fill factor for an extent is exceeded, the only new pages allocated in the extent will be to satisfy requests to cluster the new page near pages already in the extent (ie. store "near page X"). The fill factor is normally set to less than 100% when a file is first loaded, permitting the system to reserve space for future insertions.

3.1.3. Design Decisions

For simplicity, each extent belongs to a single file and extents are full cylinders. Using this scheme, page one of the Volume header fits comfortably on a page. If extents could be shared among files or the unit of allocation were less than a cylinder, then the first page of the volume header would no longer fit on a single page. In retrospect, trying to squeeze this information into one page was a mistake. The objective of this decision was to minimize the amount of main memory allocated for the bit map for each mounted volume. Unfortunately it generated a number of serious problems. First, since the minimum file size is an extent, small files can waste a lot of disk space. Second, records from different files cannot be located in the same area[4]. Finally, even when each extent corresponds to one cylinder it is not always possible to fit the bit map on one page. For example, the bit map for the Fujistu Eagle drive (842 cylinders, 20 tracks/ cylinder, 6 pages/track) will not fit in one 4K byte page.

We are exploring several possible solutions. The obvious approach is to expand "page 1" of the Volume header. Besides enabling us to handle large drives, this would permit us to make an extent as small as a single track. The amount of space wasted by small files would thus be reduced. Another approach would be to make extents variable in size.

Level 0 is ignorant of higher level system details. Some functions that could have been implemented here have been moved to other levels. These include linking the pages of a file together, identifying the first and last pages of the file, and managing the file directory. Level 0 performs page-level I/O operations with no knowledge of file or record structures.

---

[4] Inter-relational record colocation may be desirable for link data structures or to support database joins.

3.1.4. Data Structures

The key Level 0 data structure is named VolDev. This table keeps track of the volumes that are known to the system and have been mounted on-line. A device name is specified when a volume is first mounted. Level 0 reads the first page of the Volume Header for this device, and the following is stored into VolDev: (1) the name of the device, (2) the device's physical address, (3) the Volume ID of the device mounted on it, (4) the address of the freshly allocated buffer to hold the first page of the Volume Header.

References to a volume are by Volume ID. Thus to find internal information about a volume, level 0 must search through VolDev. The use of "mounted volume numbers," allocated when a volume is first mounted, and supplied by the application on subsequent volume accesses, could improve performance. This technique has been used for level 2's open file table and is based on the scheme used in the UNIX file system.

3.2. Buffer Manager

Level 1 is responsible for buffer management and concurrency control, and calls the level 0 interface to read pages from and write pages to disk. It maintains a buffer pool of pages, and implements a page replacement strategy.

Stonebraker [STON81] has observed that applications such as database systems have better knowledge about how pages are reused than file system buffer managers. When accessing a file by a random search method, such as hashing, a recently accessed page has the same probability of being reaccessed as any other page in the file. When searching overflow pages in certain file structures, the anchor page will very likely be reaccessed. Directory pages are frequently reaccessed, and should be locked in memory if possible.

The WiSS buffer pool manager uses a least recently used replacement strategy combined with hints from the system on which pages are important. A *low* hint means that the buffer pool page is not special and can be swapped out with little effect on performance. Low is specified for pages that are randomly accessed or have been read by sequential pages. A *mid* hint implies that the page has some importance and should not be swapped if there are less important pages available for replacement. Anchor

pages of overflow chains encountered on a sequential scan are candidates for a mid hint. A *high* page should be replaced only as a last resort. B-tree root pages and system directory pages receive high hints. When choosing a page for replacement, the buffer manager looks for the buffer with the lowest hint and the oldest timestamp.

Each user has an allocation of pages within the system buffer pool. Note that buffering is associated with users rather than files. A user's files compete for buffer resources within the user's allocation. However, a busy user can exceed his share if unused buffers are available. Since a user is only guaranteed his allotted share, the extra[5] buffers will be reclaimed if other users need their allocations. Except for the explicit notion of a "hot set" and the ability to fix pages in the buffer pool, our approach is very similar to the one proposed in [SACC82].

The buffer manager's primary data structure is BufTable, an in-memory table that maintains the state of pages that are resident in the buffer pool. BufTable is an array of records, indexed by buffer address. Each record in the table contains the following entries:

Page Identifier -	the disk page that resides in this buffer.
File Identifier -	the file to which the page belongs.
User Identifier -	the user currently using this page.
Timestamp -	Time of the last access (used to determine LRU).
Previous, Next Buffer -	pointers in a linked list of buffers for this user.
Next Free -	pointer in a linked list of buffers (when not in use).
Hint -	the buffered page's "level of importance." A hint is used to keep an important page in the buffer pool that might otherwise be swapped out due to an LRU strategy.
Dirty -	if the page has been updated, it must be written back before being swapped out.
Read/Write -	the mode in which the file was opened. A read only page does not have to be written when it gets swapped.

The following tables are used to help maintain the buffer pool:

[5] Regardless of the level of hint associated with each buffer.

ActiveFileTable -	A list of active files. This table contains a pointer to a linked list of users of a given file. If the file is opened for write access then only one user may have it open.
FileUsersTable -	This table links together the list of active users.
BufferUsageTable -	This table maintains the number of buffers each user is using. It also points to a linked list of the buffers used in BufTable.

Our initial approach for concurrency control is straightforward. The unit of lock granularity is the file. This means that files may be open to one writer or many readers.

We had hoped that the hint mechanism would allow us to avoid fixing individual buffer pool pages. We wanted to avoid the potential deadlock situation in which a user[6] (1) overruns his initial allocation, (2) fixes additional buffer slots rightly allocated to other users, and (3) still does not have enough buffer pages to complete. Deadlock will occur when no user can complete because insufficient buffer slots are available. In our implementation, no slot is fixed, so it is always possible to grant a slot request to a user. Hints seemed to provide an elegant way of keeping important pages in the buffer pool without fixing them.

However, giving a page a high hint does not guarantee that a page will stay in the buffer pool. The decision to not fix pages greatly complicated the code at higher levels. In implementing an algorithm, one could never be sure that a buffer page had not be replaced between calls to the buffer manager. To make this easier to determine, we restricted level 3 and 2 to have a single thread of execution, i.e., a call to level 3 was processed to completion before another call was permitted (this is adequate because level 3 receives messages from concurrent users and serializes their requests). The state of the buffer pool was kept consistent as long as an explicit read or write was not requested.

3.3. Storage Structures

Level 2 is responsible for implementing storage structures. Above this level, all accesses are to records. Below this level, the unit of access is a page. We have implemented sequential files, primary and secondary indices, and long data items. Indices are identified by extending the primary file name with an

[6] A "user" is actually some algorithm running at a higher level.

index number. For example, the indices on the EMP file are named EMP.1, EMP.2, etc.

Above level 2, files are known by name. Below this level, they are known by their internal file ID. The mapping of file names into file IDs is performed using the file directory associated with each volume. Level 2 is also responsible for managing a data structure, the open file table, that keeps track of all active files and their modes of access. File descriptor information from each mounted volume is cached in the open file table.

A facility for storing variable length items within a record that may grow to virtually unlimited size is also provided by level 2. The contents of such items are regarded as unstructured streams of bytes. Operations to modify, insert, and delete substrings within such items are provided.

3.3.1. Records and Sequential Data Organization

We adopted the approach used by System R ([ASTR76], [CHAM81]) and INGRES [STON76] for storing records of limited[7] size. Records are treated as black boxes. Each record is identified by a unique Record Identifier (RID). The RID associated with a record is permanent, and indicates where the data is actually stored. The format of a RID is:

```
+-------------------------------------------+
| Volume ID  |  Page Address  |  Slot in page  |
+-------------------------------------------+
```

The "Volume ID" field identifies an individual storage medium, rather than the device on which the medium is mounted. This provides for the situation in which a medium might be mounted on one of several devices; or one in which several media are mounted on a single device (such as an archive). With this mechanism, each record is given a name that is unique for its lifetime.

All data pages consist of some fixed number of bytes allocated contiguously on secondary storage. Pages in a file are doubly-linked to provide a nominal "sequential" scan. The file ID and the Page ID of the current page are stored on the page to facilitate error checking. In addition, the number of slots, or RIDs, valid on this page is stored, and space for that many records is allocated. The number of slots on a page can grow and shrink as records are added to or removed from the page. However, since

[7] By *limited* we mean less than or equal to one page in length

records can be deleted in any order, and since the RID of a record must not change, there may be unused slots in the middle of the slot array that are not returned to the free area after the corresponding record has been deleted.

The record format for the Storage Level is very simple:

```
+-----------------------------------------------+
| Record Type | Record Length | Actual Data |
+-----------------------------------------------+
```

If a record grows too big for the page it resides on, it may be moved to another location, and a marker left in its place. The marker will be a record indicating that the data has been moved; the data for the marker record will be the RID of the new location. If the record again outgrows its new home, there is no need to leave another marker behind as the only reference to its current physical location is in the original marker. It is this marker which is updated to reflect the new location of the record.

Since a record is globally referenced only by its RID, it can be placed anywhere within its data page: only its corresponding slot (which the RID actually addresses) needs to be updated when the record is moved. In particular, when another record in the page grows or shrinks, the entire page may be shifted, and the free space pointer updated. In addition, the records may appear in any order in the page, not necessarily in their RID order.

A sequential file may have random, clustered, or sorted order. Random order implies that the file is not sorted on any attribute and that insertions of records may be done anywhere. Sorted order means the file is sorted on some attribute and insertions require resorting if the record cannot be put in the required location. Clustered order is similar to sorted order except that exact order is not required: a record may be put near its correct spot. The main difference is that insertions do not require resorting. Level 2 directly supports clustered sequential files, as well as random sequential files. The interface for inserting records allows a caller to specify as a parameter the preferred location of a new record. This, however, does not guarantee the exact ordering of records. To maintain a strictly ordered file, a sort routine provided by Level 2 must be explicitly invoked.

3.3.2. Long Data Items

One of the goals of WiSS is to facilitate storage of variable length objects that may grow to virtually unlimited size (in particular, much larger than a single page). WiSS calls these objects *Long Data Items* since they can be included within a record. A Long Data Item consists of two parts: the data segments (called *slices),* and a *directory* to these data segments. A *slice* is considered to be at most one page in length. It is created by allocating all the available space in an empty data page. Therefore, when a slice is allocated, it will be the only entry on the page. If the data item needs only a small fraction of its last slice, efficient use of the space becomes a concern. We define a *crumb* to be a piece of a long data item that is much smaller than a page. A crumb is implemented as a short record that shares space with other crumbs, or with other normal records.

The *directory*[8] exists as a regular record that may grow to the size of one page. It contains the RID and the actual length of each slice within the long data item. This provides fairly efficient access to any arbitrary section within a long data item. The format of the directory is:

```
+---------------------------------------------------------------------+
| # slices | RID 1 | Length 1 | RID 2 |   ... | RID n | Length n |
+---------------------------------------------------------------------+
```

With a page size of 4K bytes, a long data item directory can accommodate approximately 400 RID/Length pairs in the directory record. At almost 4K bytes per slice, this gives an upper limit of nearly 1.6M bytes in a long data item.

To access a long data item, it is first opened, and then operated on. When it is closed, the data is shifted to remove holes caused by deletions or insertions of bytes. Compressing a long data item adjusts the amount of data on each slice to be at least as full as the page fill factor of the file the item belongs to, thereby reducing the number of pages used by the item. Compression is usually done at the time a scan on a long data item is closed. (Scans will be discussed later.)

---

[8] To include a long data item within a record, the RID of the directory is stored within the record.

3.3.3. B-tree Indices

A prefix B-tree (actually a B^+-tree) was chosen as the primary structure for storing indices. Since all the index data is stored in the leaves, they are doubly-linked to provide for sequential scans on the index. There are no real "primary indices" in WiSS in the sense that no data records are actually stored in the leaves of any B-tree. Instead only pointers to records (namely, RIDs) are stored. A primary index in WiSS is one in which the key of the index corresponds to the key on which the records in the file are clustered.

The internal structure of a B-tree contains three types of pages: root, internal nodes, and leaf pages. These pages are similar in format except for a few minor differences. Each type of page begins with a fixed amount of data space where entries[9] are stored. This data space is followed by control information. The root page also contains information needed for searching, inserting, and deleting data within the tree. This *header information* describes the attributes of the key on which the index is built. The page format of the internal nodes of the B-tree is the same as the root page except there is no header information.

The leaf page format is similar to the format of the internal pages except that instead of containing key/pointer pairs there are key/RID-list groups. The format of a key/RID-list group is:

```
+--------------------+------------------------+
| key (length, value) | RID count | RID list ... |
+--------------------+------------------------+
```

For secondary indices, an arbitrary number of records may share the same key value and hence any number of RIDs may be associated with a single key in the index. Therefore there must be a mechanism to deal with very long RID lists. The solution normally used is to chain overflow pages off the leaf page. WiSS, instead, uses a long data item to store overflow RID lists. The RID of the long item directory and the negated RID count are placed where an ordinary key/RID list group would go. The negated count acts as a flag to the software processing the index that the following RID is not the RID of a record but rather the RID of a long data item that is filled with RIDs of records possessing this key value. The format of such a long key/RID list group becomes:

---

[9] The word *entry* is used as a generic term to refer to any entity that may reside in a page

```
+--------------------+-----------------------+
|                    | negated  |   RID of   |
| key (length, value)|   RID    |  long item |
|                    |  count   |  Directory |
+--------------------+-----------------------+
```

There are two ways to construct a B-tree index. One approach is to build an empty B-tree, and then fill it up by repeatly inserting key/RID pairs. Obviously, this is inefficient both in execution speed and storage utilization (as the resulting B-tree may not be "well" balanced). The other way is to create a sorted key/RID file first, then build the B-tree from the bottom up. For a primary index, the entire data file is sorted, and then key/RID pairs are extracted into a temporary file. For a secondary index, key/RID pairs are extracted into a temporary file which is then sorted. In both cases, the temporary file is used as the basis for constructing the B-tree from the leaves upward to the root.

3.3.4. The Sort Routine

The WiSS sort routine implements a disk-based external merge-sort. Given a data file, it rearranges the order of records in that file according to some attribute. As mentioned in the previous section, it is used by the routine that constructs B-tree indices. It is also accessible to users of WiSS for those cases in which strictly ordered files are required (e.g. implementing a sort-merge join algorithm).

Since main memory has become very inexpensive, our sort routine is designed to utilize as much primary memory as possible in order to minimize the number of disk I/Os. Furthermore, since it is precisely known when a page should be read or written, there is no need to use a generalized buffer manager. Consequently, our sort routine bypasses the level 1 buffer manager of WiSS, and does the necessary page buffering itself.

The sort routine can be divided into two distinct phases: a sorting phase and a merging phase. For the purpose of discussion, we define a *partition* to be a collection of sorted records. During the sorting phase the sort-buffer is repeatedly filled with unsorted pages. The records in the sort-buffer are then sorted using a combined method of quick sort and straight insertion sort. After each sorted partition is produced, records are gathered[10] into a output page buffer, which is written to disk as soon as it is full.

After the sorting phase, partitions are merged in successive passes to produce larger and larger

[10] During the sorting phase instead of actually moving records around, pointers to the records in the partition are sorted.

partitions until there is only one partition left. To maximize performance and minimize disk space utilization, only one level 0 file is used as the external storage space. The size of this file is approximately equal to that of the source file. During each pass of the sort, the page IDs (PIDs) of the pages in the input partitions to be merged are placed in an array of PIDs. As pages are merged, the PIDs of the output pages used are recorded in another PID array. Immediately after a page is read, its PID is placed in a recycle bin. There are auxiliary data structures to describe which partition each PID in an array belongs to. In addition, since pages come from the same physical file (which is physically clustered), disk seek times are automatically minimized.

3.3.5. File Directory and the Open File Table

The file directory is used internally by the storage level to associate external file names with internal information. This internal information includes the level 0 file id, the PID of the first and last pages of the file, the page fill factor, etc. There is one directory for each volume. The address of the root page of the directory is recorded in the volume header at a fixed location.

The file directory is implemented by a hybrid method of *extendible hashing* ([FAGI79]) and *chained bucket hash* (Figure 1). In most file systems, (e.g. UNIX), the size of a file directory is too large to be kept in memory. Thus the first reference to a particular file (to open it, for example) usually requires several disk accesses. Extendible hashing guarantees that at most two disk accesses are ever needed, one for the hash table and one for the leaf page. However, to limit the amount of main memory allocated to file directory hash tables,[11] we decided to limit the maximum size of each hash table to one page[12]. This creates a problem when the page on which an insertion is to be made is full and the hash table has already expanded (through the doubling process [FAGI79]) to occupy a full page. In this case, rather than doubling the hash table to occupy two pages, overflow pages are chained to the primary page of the bucket that overflowed (see Figure 1).[12] However, since the hash table is resident in memory, generally only one disk access is required to access a file descriptor.

[11] There is one such table for each mounted volume.

[12] Assuming a uniform hashing function, we have calculated that file descriptors for approximately 64,000 files can be maintained before overflow pages will be necessary.

[13] To obtain maximum fan out (and thus maximum performance), all entries in the hash table are simplified PIDs (i.e., a page address without a volume ID).

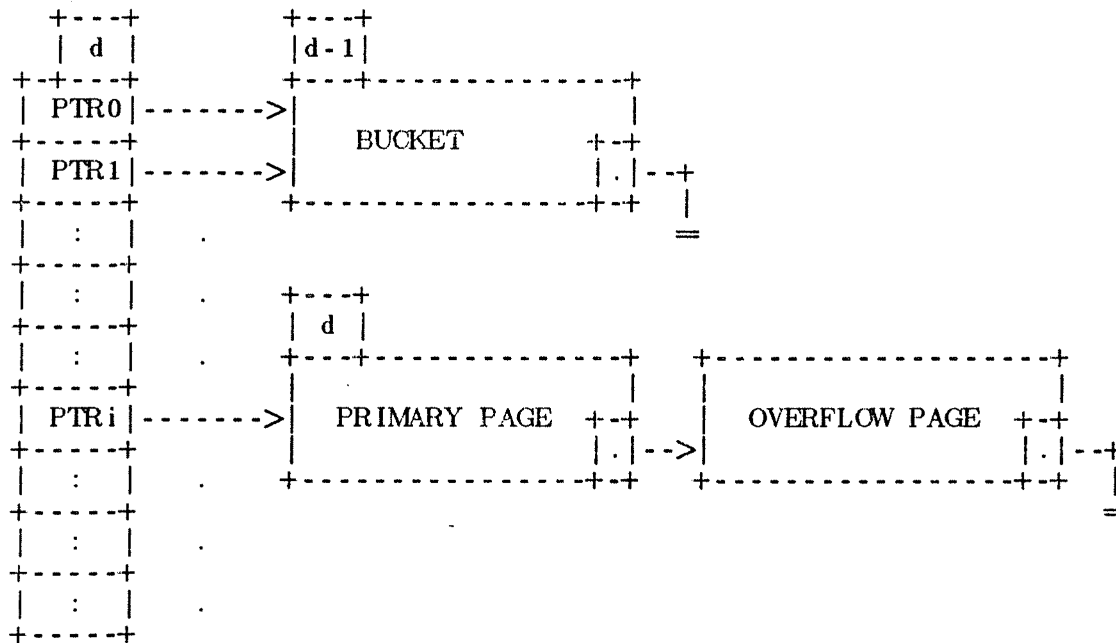


Figure 1: File Directory Structure
(Extendible Hashing + Chained Bucket Hash)

An "open file table" is used to enhance performance. It can be viewed as a cache for the file directory information from all mounted volumes. Each entry in this table contains the file descriptor as well as certain run time attributes such as access mode. The table also provides facilities for updating the internal information of an open file. Before a request to open a file is granted, the mode of access requested (shared or exclusive) is checked for conflicts with other concurrent accesses. A file may have many simultaneous readers but only one writer. If access to a file is granted, an open file number is returned. Subsequent operations on the file provide the open file number to level 2 directly, bypassing the table lookup.

3.3.6. Design Decisions

In the original design of WiSS, long data items were to be implemented at level 3 using the record structure and operators provided by level 2. While technically feasible, this approach was abandoned as too inefficient. The problem is that the level 2 interface does not permit direct access to a record in the buffer pool. Rather records must first be copied into a level 3 buffer before they can be manipulated. Thus, implementing long data items at level 3 involved a great deal of unnecessary movement of

data across the level 2 interface. The solution was to move the support for long data items into level 2. The components (slices and crumbs) of a long data item are implemented as records but are manipulated directly in the system's buffers.

In our first attempt to implement the storage level (level 2), the task was assigned to two disjoint groups of students. One group was in charge of implementing the record and sequential file structures. The other group was responsible for implementing the B-tree mechanism. The work was left in an inconsistent state when the semester ended. The following semester two different groups of students continued work on the storage level. As one would expect, each of these groups re-organized and rewrote a fair amount of the code written by their predecessors. When the sequential file structure was finished, work on long data items was begun. By the end of the second semester of coding, long data items were also operational but B-trees still only partially worked!

After the first two attempts to produce working code for the B-tree mechanism failed, it seemed desirable to simplify and integrate the data structures and operations associated with the B-tree mechanism. For example, in one early implementation, the position of the key in a non-leaf entry (ie. a key/pointer pair) was different from that in a leaf entry (ie. a key/RID list). This made the B-tree traversal algorithm, as well as the algorithm for updating B-trees, harder to implement. As we made further efforts to complete the implementation of the storage level, the following approach for organizing the code emerged.

The idea we developed was to structure the storage level as four layers: the record layer, the long data item layer, the B-tree layer, and the interface layer (which isolates the rest of level 2 from higher level users). The underlying philosophy is to use only two types of base objects throughout level 2: records and long data items. Any object whose size is smaller than that of a page is stored as a record. Objects that are too large to fit on a single page are stored as long data items. The record layer serves as the interface between records and pages. It provides routines for manipulating records in a page. The long data item layer builds various pieces of a long data item as records. Slices, crumbs, and even the directory of a long data item are all records. Thus the work of this layer is significantly simplified. Finally, the B-tree layer treats key/pointer and key/RID list pairs as records, and lets the record routines manipulate entries in B-tree pages. As mentioned earlier overflow RID lists for secondary indices are in

essence something that is too large to fit into a record. Since we already have mechanism for handling large objects, a long data items seemed to be a convenient place to hold an overflow RID list.

This turned out to be an elegant way to quickly implement the storage level while making the code significantly more manageable. Simplicity is gained, however, at the expense of both performance and space. Execution overhead arises from extra layers of procedure calls. Storage overhead results from the auxiliary control information which is associated with records or long data items. As a compromise, we re-organized most of the storage level according to this layering concept. Where performance was deemed critical we retained the original organization. For example, while B-tree entries are very similar to records in structure, the record header is not included in a B-tree entry in order to increase the node fan out.

3.4. Access Methods

Level 3 implements the access methods of sequential scan, index scan, and long item scan. The interface also provide the capabilities to create and destroy files, indices, and long data items by calling lower level primitives. Level 3 software maintains an active scan table that describes all the scans in progress. The table includes information about the scan type (sequential, index, or long data item), the open file number of the file being scanned, a cursor for describing the current scan position, and several parameters that are determined by the scan type.

In some applications, explicit control over the scan cursors is desirable. For example, in the sort-merge method for doing a join, it is necessary to back up the cursor for the inner relation if more than one tuple in the outer relation has the same join attribute value. Thus, level 3 provides routines for interrogating, setting and resetting the cursor of any scan in progress. Scans on sequential files and indices can proceed in both directions. This is handy, for instance, for listing tuples in either ascending or descending order.

3.4.1. Scan Table

For each scan, the table contains the scan type (sequential, index, or long), and the open file number of the file associated with the scan. A cursor is also maintained for each scan. For scans on long

data items the cursor is the current byte offset of the scan. For sequential scans, the cursor is the RID of the record currently being processed. For index scans, a cursor pointing into a leaf page is stored. There are, in addition, several additional parameters associated with each type of scan. For example for scans on long data items, the RID of the long data item is stored in the table. For sequential and index scans, a boolean expression can be specified as a search argument to filter out unwanted tuples. In addition, for index scans an upper and lower scan range of the key field can be specified to skip undesired key ranges.

The scan table is designed as a set of entries, each of which describes a currently open scan. Scan table entries are organized into dynamically allocated blocks. This allows virtually an unlimited number of scans to be open simultaneously.

3.4.2. SARGable Predicates in WiSS

A "SARGable predicate," is a "predicate usable as a Search ARGument." In general SARGable predicates are boolean expressions in conjunctive normal form. When a scan with a SARGable predicate is opened on a file, the only records returned by the scan will be those for which all the terms of the predicate are true.

Level 3 of WiSS supports a limited form of a SARGable predicate in which the conjuncts are simple (ie. no ORs). A WiSS SARGable predicate is implemented as a linked list of structures each of which contains one term of the complete boolean expression. Each boolean term consists of three parts: a relational operator ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$), a field value descriptor, and a pointer to the next term. Field value descriptors are a generalization of constant values in the context of records. A field value descriptor contains: the offset of the field within the record, the type of the field (integer, short integer, floating point, etc.), and the actual value to which the field is to be compared.

Any operator may be applied to any data type. There is a 256 character limit on the size of a string argument. Conceivably, it would be possible to search for a longer string by using two or more terms and adjusting the offset appropriately. The alignment of numeric fields within a WiSS record does not matter as the routine that evaluates SARGable predicates checks before performing the comparison. If a numeric field is not properly aligned, the value is copied into a temporary before the comparison is performed.

As discussed earlier, in general the level 2 interface requires that before a record can be accessed it must first be copied out of the buffer pool into a local buffer. An early version of the level 3 routines did indeed go through this procedure in order to examine the values of fields for each record. As should be quite obvious, the performance of this approach was very poor. Therefore, the level 2 interface was modified[14] to permit inspection of a record without copying it.

3.5. Recovery and Concurrency Control

The concurrency control and recovery mechanisms in WiSS are very primitive. Locking is used as the concurrency control mechanism but the locking granularity is the entire file. No deadlock detection is currently performed although adding this would be straightforward. No recovery mechanism is provided.

In retrospect it is hard to imagine[15] why we ignored incorporating some concurrency control algorithm and recovery mechanism in the design of WiSS from the beginning of the project. We recently looked at adding these functions to the present design. Adding a lock manager for page level locking is relatively straightforward (in fact such a lock manager was written but has not yet been incorporated). "Tacking on" recovery is not, however, easy. In particular, making operations on long data items and B-trees recoverable at this point appears to be a very difficult task.

4. Implementation Experience

The idea to design and implement a system such as WiSS was originally suggested by Tony Klug in September 1981. Randy Katz and David DeWitt were recruited to join the project and DeWitt's graduate course in database systems completed an initial design during that semester. Between semesters a more detailed design document was prepared. A special seminar was organized during Spring 1982 to complete the design and to begin implementation. Rick Simpkin and Toby Lehman served as chief programmers for this portion of the project. By the end of the spring term, levels 0 and 1 were implemented and debugged and level 2 was partially implemented.

Another seminar was organized for the summer term in an attempt to finish the project.

[14] The whole issue about interfaces may seem rather trivial. However, the interfaces were design to strike a balance between performance and protection.

[15] Perhaps it was because we never really thought that we would use WiSS for any "real" applications.

However, only one student (H-T Chou) was left from the spring semester. Chou assumed the responsibility as chief programmer. In addition to leaving only part of level 2 working, the first group of students left the documentation for the level 2 code that had been written in very poor shape. It was very inaccurate, describing things that had never been implemented and not describing things that had. The comments in the level 2 routines were not much better. In fact one former project member left a phone number in the B-tree code where she could be reached if her successor needed help. When tried, the phone number (like the code) was "out-of-order". Consequently, we abandoned all the level 2 code and started over on it from scratch. We had no idea how difficult it would be to implement code for the B^+-tree indices. Support for long data items was incorporated at this time. Level 3 sequential scans, index scans, and long item scans with simple restrictions were also completed.

By the end of the summer all of WiSS was partially working. At this point, we decided that all future changes should be made by only one person. The past year was spent improving the code, the documentation, and tracking down the remaining bugs. In the following section we will describe the performance of the current system. In Section 6, describe some of the projects for which WiSS has already been used.

5. Performance Evaluation

We recently have begun an effort to benchmark a number of existing relational database machines and database systems. In [BITT83a], the results of these tests on the university and commercial versions of INGRES and the Britton Lee IDM 500 database machine are presented. In this section we present the results of a selected subset of our benchmarks run using WiSS as an access method.

5.1. Description of the Systems Evaluated

5.1.1. WiSS

Since we do not yet have an operational database system constructed using WiSS, we implemented all all the major relational operators as procedures. These procedures rely entirely on WiSS for storing and manipulating relations in the test database. For each operator, algorithms that rely on sorting, indexing, and other data structures were implemented. We feel that the results presented below are

indicative of the performance obtainable by a database system implemented using WiSS and pre-compiled queries [ASTR76]. Since the other two systems were tested using an ad-hoc query interface, the times for each of the other systems could probably be lowered by at least a second if evaluated with a similar interface.

The tests were run on a VAX 11/750 with 2 megabytes of memory and four disk drives connected to the VAX with a System Industries 9900 controller using a CMI interface. The operating system was Berkeley 4.1 UNIX. The database was stored in a WiSS file system that was mounted[16] on a Fujistu Eagle disk drive (474 Megabytes).

5.1.2. University-INGRES

The version of university-INGRES tested was that delivered on the Berkeley 4.1 distribution tape. The same computer and disk drive used for testing WiSS were used for the University INGRES tests. Immediately before the database was loaded, a new UNIX file system was constructed on the drive thus maximizing the probability that two logically adjacent blocks would be physically adjacent on the disk (an atypical situation for a typically scrambled UNIX file system).

5.1.3. Commercial-INGRES

The VAX 11/750 on which Commercial-INGRES was evaluated had 6 megabytes of memory, and a Fujistu Eagle drive connected to the processor through the CMI bus with an Emulex SC750 controller. The test database was stored on the Fujistu drive. The operating system used was VMS release 3 which provides an extent based file system. For our test 800K bytes of main memory was allocated for buffer space and 200K bytes were allocated for sort space. Commercial INGRES provides a number of performance enhancements not found in the University version. These include a new query optimizer, sort-merge join strategies, 2K byte data pages (versus 1K byte pages in 4.1 Unix and 4K byte pages in WiSS), and buffer management under the control of the database system.

---

[16] The disk was mounted on UNIX as a "raw" device.

5.2. The Benchmark Database

The benchmark database consisted of five relations whose attributes were designed to facilitate writing a wide variety of queries (see [BITT83] for a detailed description of the database). The test relations contained ten thousand, 182 byte tuples. While the test relations have both string and integer attributes, we have only included the results of the tests performed using integer attributes.

5.3. Benchmark Results

In this section, we present a subset of our benchmark measurements. We have divided this section into three subsections: one each for selection, join, and update operations. Each test was run on a stand-alone machine. The times presented are the elapsed time to run each of the queries.

5.3.1. Selection Tests

For selection tests we ran a[17] query that selected 100 out of 10,000 tuples. The same query was run with and without a suitable index. Without an index, each system must scan the entire relation to apply the selection criteria. The indexing mechanism used in both versions of INGRES is based on an ISAM file organization. The B-tree mechanism was used in WiSS. For each system, the file was clustered on the index attribute. Finally, the tuples produced by each of the queries were inserted into a new relation. The results of these tests are displayed in Table 1 below.

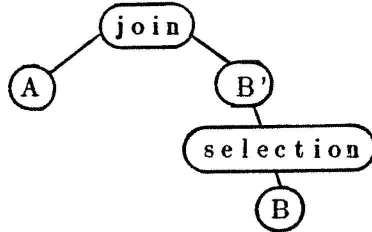
Table 1
Select 100 out of 10,000 Tuples
Total Elapsed Time in Seconds

System	No Index	Clustered Index
U-INGRES	53.2	7.7
C-INGRES	38.4	3.9
WiSS	48.1	1.7

[17] Actually, ten different queries with the same selectivity factor were run and their average execution time is presented.

5.3.2. Join Tests

To evaluate the performance of the different systems when performing more complex queries, the following query was run on each of the three systems:



In this query the B' relation is formed first by selecting 1000 tuples from the 10,000 tuple B relation. Next the A relation (10,000 tuples) is joined with B'. The relation produced by the join operation contains 1000 tuples.

Without an index, the university version of INGRES uses a modified nested loops join algorithm in which the storage structure of the inner relation (B') is first modified so that it is hashed on the joining attribute. Commercial INGRES and WiSS use a sort-merge join algorithm. When an index exists on the joining attribute, both University INGRES and WiSS use a nested loops join algorithm along with the index. In all cases the index also reduces the execution time required to produce B'. The execution times for the various systems are shown in Table 2.

Table 2
Join Queries
Total Elapsed Time in Seconds

System	No Index	Clustered Index on Join Attribute
U-INGRES	611	126
C-INGRES	109	54
WiSS	137	32

5.3.3. Update Queries

The final tests performed were to examine the time necessary to update a database in WiSS. Each update query was performed in a stand-alone mode. The relation being updated had three indices

which also had to be updated. The results are presented in Table 3.

Table 3
Update Queries With Indices
Total elapsed time in seconds

System	Append 1 Tuple	Delete 1 Tuple	Modify 1 Tuple (Key Attr)	Modify 1 Tuple (Non-Key Attr)
		Query Type		
U-INGRES	9.4	6.8	7.2	9.1
C-INGRES	2.1	0.5	1.6	1.6
WiSS	0.7	1.0	0.9	0.9

5.4. Discussion

The purpose of these performance test was to demonstrate that WiSS is a "real" system and not to make any claims that a full database system implemented using WiSS would outperform either version of INGRES. The results of these tests indicate to us that WiSS provides a satisfactory level of performance. In particular, we feel that other systems can be built on top of WiSS without worrying that WiSS will be a bottleneck. Finally, as will be illustrated below, benchmarking the system against more mature systems provided a way of uncovering a number of serious design flaws and several serious bugs in the system.

In obtaining the results for WiSS which were presented above, we observed a number of things about the performance of WiSS. First we were able to improve the performance of various routines in WiSS by a factor of between 5 and 20 with only 2 man months of effort (after spending 3-5 man years developing the system). One place where a dramatic performance improvement was made was in the buffer management routine. The initially tests of the selection operation without indices indicated that WiSS was 5 times slower than the university version of INGRES. The problem turned out to be that the buffer management routine was calling UNIX to get the value of the time of day clock for use in the LRU replacement algorithm. Avoiding that call to the operating system by using a counter as a clock had a dramatic improvement in performance.

Another significant improvement in performance was obtained by drastic modifications to the comparison routines (which are at the very inner loop of the system). Initially (except for string attributes), all attributes and the constants to which they were being compared were being converted to double precision floating point form. While this approach is simple, it is very slow. After this was corrected we obtained another 50% speedup in the compare routines by comparing numeric attributes in place when properly aligned (at the cost of a division).

The buffer management code remains the bottleneck in the system. Currently determining whether a page (identified by its PID) is present in the buffer pool is done by using a linear scan. For large buffer pools this linear scan becomes expensive. Hashing the PID would clearly provide a significant gain in performance at the expense of slightly increasing the complexity of the replacement algorithm.

Doing the benchmarks uncovered a very serious bug in the B-tree mechanism (where else!). It turned out than on index scans, the entire top of the B-tree was always searched no matter how restrictive the search criterion. While this was a simple bug to fix, it would have never been discovered without benchmarking the system as the code that handled the leaf nodes of the tree was correct and thus the scan always returned the correct result.

6. User Experiences to Date

During Fall 1982, WiSS was used extensively for special projects by students in an advanced database course taught by Katz: Among the projects implemented were:

(1) A mechanism for supporting multiple versions of records was implemented using B-tree indices.

(2) The WiSS access methods were experimentally extended to included extensible hashing.

(3) A simple entity-relationship database was implemented using WiSS as a basis.

During the Spring of 1983, WiSS was again used as the basis for projects in the advanced database course. In this course the students were given WiSS, an IDM 500 database machine reference manual, and the "front-end" software that is used to communicate with the IDM 500. Their project was to implement the software necessary to make an IDM 500 compatible database machine. Students worked in groups of 6. In general, each of the groups got a subset of the IDM system operational by the end of the semester.

7. Conclusions and Future Plans

In this paper we have described the design of a flexible data storage system for the UNIX environment. WiSS was designed and implemented as an experimental vehicle for building database management systems. The storage component forms a foundation upon which a variety of database systems can be constructed including support for unconventional types of data.

WiSS is being currently being used in a number of projects at both Berkeley and Wisconsin. At Berkeley it is being used as the basis for a design database management system for VLSI data. At Wisconsin it is being used as a file server for a multicomputer project consisting of 40 VAX 11/750 computers interconnected with an 80 Mbit/second token passing ring. In the near future we intend to use it as the storage component of a multiprocessor data base machine for statistical operations.

8. Acknowledgements

Toby Lehman and Rick Simpkin were instrumental in turning the idea for a system like WiSS into reality and deserve special recognition. Toby and Rick helped formulate the design of WiSS, were responsible for establishing coding and documentation styles, and lead the two programming teams at the beginning of the project. The workers also deserve our thanks. They include: Marianina Anania, Ron Blechman, Eric Claeys, Pat Dunkin, Wei-Laung Hu, Dawn Koffman, Bill Opdyke, Greg Raymond, Judy Rinehart, Sandy Schumacher, Ann Varda, David Ward, and Ed Wimmers.

9. References

[ALLC80] Allchin, J. E., A. M. Keller, and G. Wiederhold, "FLASH: A Language-Independent, Portable File Access System," Proc. ACM SIGMOD Conference, Los Angeles, CA, (May 1980).

[ASTR76] Astrahan, M., et. al., "System R: Relational Approach to Database Management". ACM Transactions on Data Systems, V 1, N 2, (June 1976), pp. 119-120.

[BITT83] Bitton, D., DeWitt, D. and C. Turbyfil, "Benchmarking Database Systems - A Systematic Approach," Proceedings of the 1983 Very Large Database Conference, (October 1983).

[CHAM81] Chamberlin, D., et. al., "A History and Evaluation of System R," Communications of the ACM, V 24, N 10, (October 1981).

[FAGI79] Fagin, R., et. al., "Extendible Hashing-A Fast Access Method for Dynamic Files," ACM Transactions on Database Systems, V 4, N 3, (September 1979), pp. 315-344.

[GRAY78] Gray, J., "Notes on Database Operating Systems," in *Operating Systems -- An Advanced*

Course, R. Bayer, R. M. Graham, G. Seegmuller, eds., Springer-Verlag, New York, (1978).

[HASK82] Haskin, R. and R. Lorie, "On Extending the Functions of a Relational Database System," Proc. ACM SIGMOD Conference, Orlando, Fl., (June 1982).

[KERS81] Kersten, M. L. and A. I. Wasserman, "The Architecture of the PLAIN Data Base Handler," *Software -- Practice and Experience,* V 11, pp. 175- 186, (1981).

[MCKU82] McKusick, M. K., W. N. Joy, S. J. Leffler, and R. S. Fabry, "A Fast File System for UNIX," Computer Systems Research Group Technical Report, University of California, Berkeley, (September 1982).

[REED83] Reed, D., "Implementing Atomic Actions on Decentralized Data," ACM Transactions on Computer Systems, V 1, N 1, (March 1983).

[SACC82] Sacco, G. M. and M. Schkolnick, "A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model," Proceedings of the 1982 Very Large Database Conference, (September 1982).

[STON76] Stonebraker, M., Wong, G., Kreps, P., and G. Held, "The Design and Implementation of INGRES," ACM Transactions on Database Systems, V 1, N 3, (September 1976).

[STON81] Stonebraker, M. R., "Operating System Support for Database Management," Communications of the ACM, V 24, N 7, (July 1981).

[THOM78] Thompson, K., "UNIX Implementation," *The Bell System Technical Journal,* V 57, N 6, Part 2, (July-August 1978).

[WEIN82] Weinberger, P., "Making UNIX Operating System Safe for Databases," *The Bell System Technical Journal,* V 61, N 9, Part 2, (November 1982).

[WISS82a] WiSS Implementation Team, "B-Tree Structures in the Wisconsin Storage System," Internal Documentation, University of Wisconsin-Madison, Madison, WI, 1982.

[WISS82b] WiSS Implementation Team, "Design Overview of the Wisconsin Storage System," University of Wisconsin-Madison, Madison, WI, 1982.