TIME PERSISTENT DATA ALLOCATION

STORAGE ENGINE

By

RAY HARVICK

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

1996

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December 1997

OKLAHOMA STATE UNIVERSITY

TIME PERSISTENT DATA ALLOCATION

STORAGE ENGINE

Thesis Approved:

Jacques E. LaFrance
Thesis Advisor

W. Nick Street

J Chandler

Blayne E Mayfield

Wayne B Powell
Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to my advisor, Dr. LaFrance. When I was facing the red tape the university places before a student, you helped cut through it, and helped me accomplish my goal. I also wish to express my thanks to Dr. Street, who helped me with his advice and his constructive remarks. In addition I would also like to express a special thank you to Dr. Chandler, who is always a friend to all graduate students in the OSU computer science department.

However, without the support, suggestions, and backing of my Wife, Lee, I would not have my bachelor, let alone complete my Masters. Without Lee, I would be a lost soul. Not only did she support me, but was my fellow student in many classes as well as my best friend. If only I would have followed her advice closer, I would avoid many struggles I have encountered. I also want to thank my daughter Kristie, who although only 2 ½, has brought such joy into my life, and a reason to work hard. I also want to thank my wifes parents who can inmeasurable amount of support. I would also like to thank my own parents, Joe and Arleta Harvick who have always been supportive of me.

Finally, I wish to thank the Department of Computer Science for their support during these last two years.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter Page

1. Introduction

The Data Allocation Structure Engine is a software solution that is capable of storing and retrieving data (records, documents, etc.) of variable sizes (lengths), in an efficient and timely manner.

As with many technologies, database engines can be presented in dependency categories or layers. The first layer might consist of service routines to manage [hard] disk files. This type of categorizing is also referred to as the 'layered approach' [CHO89].

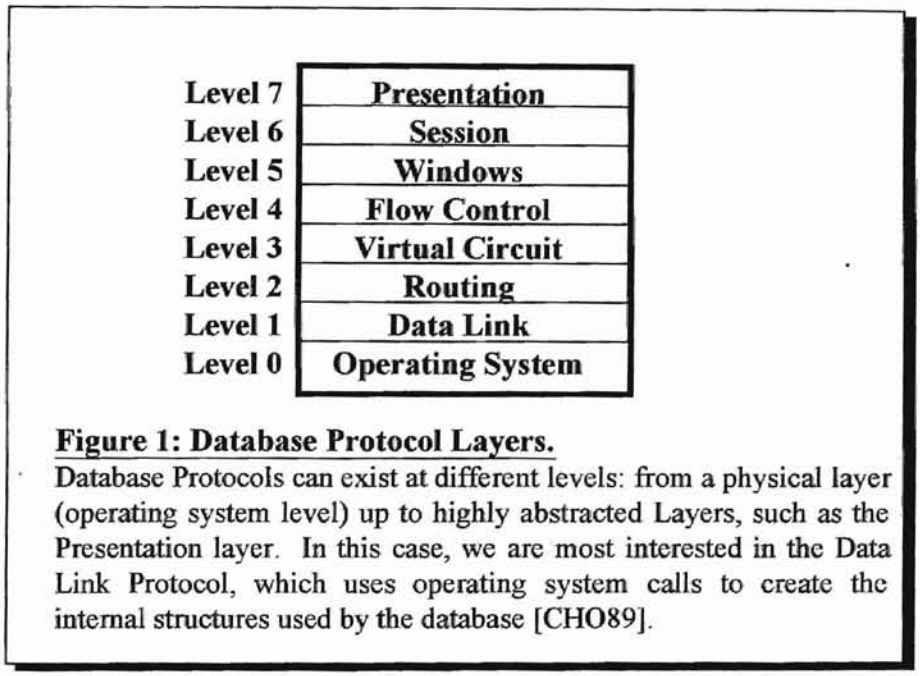


Figure 1: Database Protocol Layers.
Database Protocols can exist at different levels: from a physical layer (operating system level) up to highly abstracted Layers, such as the Presentation layer. In this case, we are most interested in the Data Link Protocol, which uses operating system calls to create the internal structures used by the database [CHO89].

Layers can be further divided into sub layers.

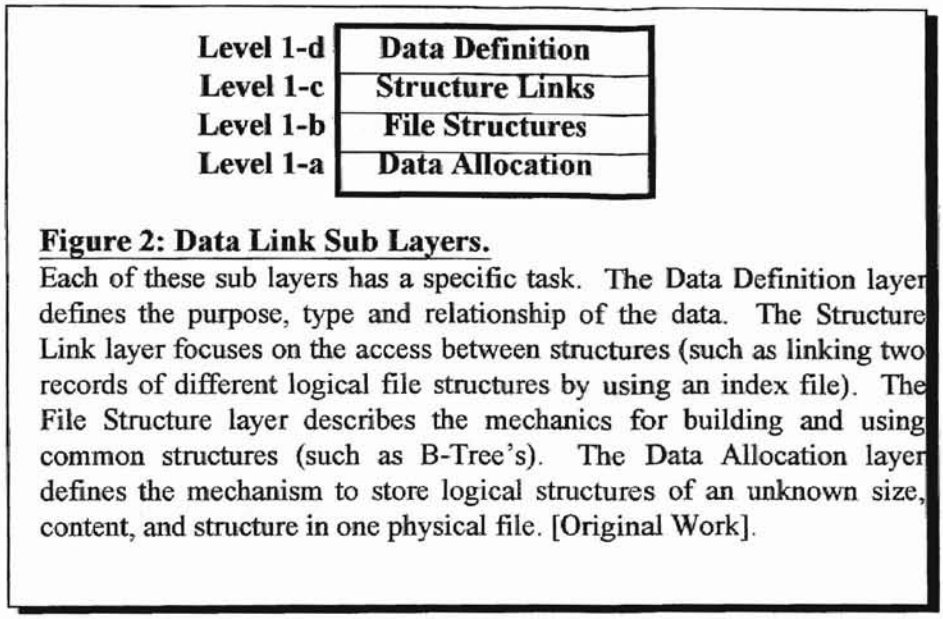


Figure 2: Data Link Sub Layers.
Each of these sub layers has a specific task. The Data Definition layer defines the purpose, type and relationship of the data. The Structure Link layer focuses on the access between structures (such as linking two records of different logical file structures by using an index file). The File Structure layer describes the mechanics for building and using common structures (such as B-Tree's). The Data Allocation layer defines the mechanism to store logical structures of an unknown size, content, and structure in one physical file. [Original Work].

The Data Link Layer shown in Figure1 provides an abstraction layer between the operating system and the higher protocol layers. The Data Link layer can be thought of as the foundation of the database (since it is the lowest layer before the operating system).

In Figure 2, the Data Link layer has been divided into sub-layers, with the lowest being the Data Allocation layer. The Data Allocation Layer allows the database to store one or more logical types of structures into one physical file (from an operating system perspective). This layer is similar to the file management systems found in many operating systems, in that it organizes data on a physical storage medium, such as a disk [SIL95]. The Data Allocation layer performs much the same operation, but organizes

data within a physical file rather than to a media. The file accessed by the Data Allocation layer can be thought of as a 'virtual disk'.

The need to discover a more robust data storage foundation for database engines is paramount. The speed and accuracy of any database engine is a direct result of the speed and accuracy of its foundation. If this layer is poorly constructed, it can negatively affect the overall performance of the database. In this thesis, we will discuss the cost and time considerations for producing a data allocation engine for databases.

Today, computers and other related devices can generate vast amounts of information, of which the vast majority is never utilized. The format in which this information is stored can vary from computer to computer, depending upon the machine architecture and software source of the file. For example, information can be stored in non-formatted files such as text files or ASCII files, or in indexed files, relational databases, hypertext databases, object-oriented databases, or as word processing documents that incorporate many special characters. Files are often stored on disk media and grouped in structures often referred to as "directories." These directories have little organization information that is useable by users of the system. They also have little to no cross-referencing and little or no association to other files. For example, a user of word processing on a personal computer may write several documents on the same subject. If that same user wanted to look up previous work, he/she would have to open each document and search manual for the required information. If this information was stored in a database, then a database search engine could be used to find the required information.

In a book named *Intelligent Databases* [PAR89], by Kamran Parsaye, Mark Chignell, Setrag Khoshafian, and Harry Wong, the idea of combining several database technologies is explored. It will not take much more thought to expand this thinking to include other file formats. While this book discussed many important topics, it did not discuss the low-level architecture necessary to develop such a database. It did, however, suggest that an object-oriented database would be an appropriate format as an underlying architecture. In further research, I discovered little discussion of the physical organization of an object-oriented database that I was able to locate at the University Library and local bookstores. Further search of the Internet, proved just as frustrating. Several of the books I read simply stated that the architecture used by most object-oriented databases were proprietary and gave no other details. The best discussion on the underling architecture came not from books on databases, but instead came from operating system books [SIL95] [TAN92] [MIL87]. These books discussed how operating systems organized files using file management systems. The most import of the operating systems discussed, was that of UNIX. UNIX has structures known as inodes that facilitate the processing of large data files. Although the idea of inodes is important, it is incomplete, in my opinion, for use in Data Allocation Storage Engines.

2. Background and Related Work

2.1 File Systems

Hard disks by their very nature are slow compared to computer memory (RAM). They are capable of storing hundreds of thousands of megabytes of data at a rather substantially financial cost saving. To illustrate, how slow they are, the time it takes for relatively slow RAM to read a byte of data is approximately 120 nanoseconds (120 billionth of a second). To access the same amount of data from a typical hard disk it would take approximately 30 milliseconds (30 thousands of a second) or a ratio of 1/120,000,000,000 of a second to 1/30,000. In other words, a relatively slow RAM is about 4,000,000 times faster than a hard disk drive. On the other hand, hard disks provide a relatively cheap device to store large amounts of information [FOL92]. The goal of file structures is to find creative ways to access data faster.

Some of the earliest works of file structures were based on the assumption that the files were on magnetic tapes. This meant that the access had to be sequential and the larger the file the greater the cost of access [FOL92].

In time hard disks became available which allowed for random access. This allowed computer scientist to invent new ways to access data. These included many indexing schemes. The indexes made it possible to store small amount of data with pointers to larger data sets. This made access quicker, since a program could load more of the

smaller data into memory, search it for the appropriate key, and use its pointer to access the larger data sets [FOL92]. As the need for more and more data grew, so did the size of the files containing the indexes. More and more clever access methods were needed to speed the search for the appropriate data. Over time, many important data structures were invented, including AVL trees, Hashing functions, and B-Trees [FOL92]. As data became more portable and distributed, the need grew to organize two or more file types into one physical file. One of the firsts of these file types was B+ Trees. These trees could store both the index and the data [FOL92]. As the need for heterogeneous data grew, databases where developed that could store indexes, their data, and the definition of their data. Many of these databases stored their data in separate physical files. However, having data spread between different files also presented problems. When you wanted to copy a database, you where required to copy several files. And if you wanted to have more than one database, you needed to organize your files either internally or externally so that the databases were separate. If the data was separated internally, then it was difficult to split the databases if they need to be moved to a new location. If they were separated externally, then they would have to either have different names for similar database files (a management nightmare) or they would have to be organized in some other fashion (such as storing them in different directories). However, if all the data for one database could be stored in a single file, then each file would keep all of its data abstractions in one file. This meant to move a database, only one file was required to be moved.

> Files are an abstraction mechanism. They provide a way to store information on the disk and read it back later. This must be done in such a way as to shield the user from the details of how and where the information is stored... [TAN92]

One of the most closely guarded secrets in the database community is the underlying storage of data in a database. While there are many books and articles written on storage structures for databases (such as b-trees), there is little written concerning how to store two or more different structures in one file. This is largely because companies that produce commercial quality databases consider such information as a key technology, and guard it jealously.

The earliest work on data storage came not from work with databases, but from work on operating systems [FOL92]. Since operating systems must be able to store and retrieve large amounts of information, such structure techniques are valuable in constructing a storage system for databases.

Three common ways operating systems store data on a disk are by, (a) Byte sequence, (b) Record sequence, or (c) Tree structure [TAN92] [SIL95].

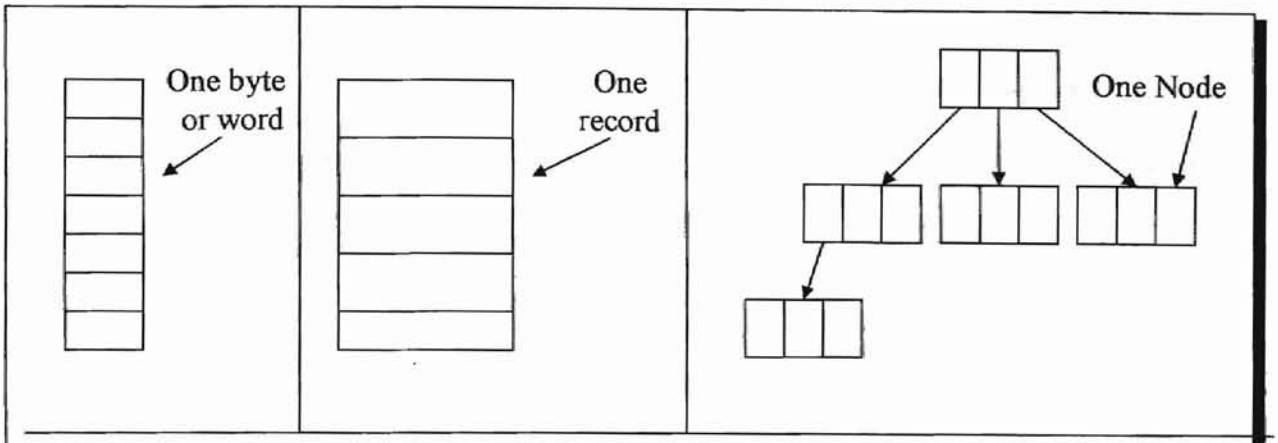


Figure 3: Operating System storage structures.
The first structure above (Left) shows the structure of byte allocation. In this structure, the entire storage structure must be allocated before any storage can be used. The second structure (center) stores data in records. This scheme allows for quick access to any datum, but is difficult to insert records. The third structure (right) provides variable length blocks and allows for quick insertions and deletions. [TAN92]

While storing information in a sequence of bytes is efficient if you have small files or files that have no requirement for structure (such as text files). However, storing information in a series of sequential bytes makes it difficult to locate information that may be buried in the middle of the file. To find any information in the file, the system must do a sequential search. To insert information a search of the file was needed to find the exact amount of space required for storing the information [TAN92]. Dividing the information into a sequence of records works well, if you have well-defined record sizes, or if your records are large enough to store the largest possible size. It then becomes a simple task to access the information by record number. The difficulty with record format is that the system must pre-allocate enough space for all records. Insert is also difficult since each record must be moved down one at a time until a slot is available at

the proper location [TAN92]. Tree structures give little improvement over records, in that they are capable of storing information in a related format. Tree structures work well when you need to insert information. The system need only make a link from the previous node. The major draw back to tree structures is when the system needs to locate a particular record. In order to locate a particular record, the system may have to chain down several nodes. [TAN92]

Another interesting structure for databases is the organization of file pointers. Most operating systems (such as MS DOS and UNIX) store data files in byte format, leaving the application program to determine the internal format [TAN92] [SAL95]. However, these same systems must store and maintain access to each separate file, otherwise the disk would become one large binary file. To do this, operating system maintains a list of files and directories. This division of disk into files is very useful in understanding how to organize a disk file for use by a database [TAN92].

> Three major methods of allocating disk space are in wide use: contiguous, linked, and indexed. Each method has its advantages and disadvantages. [SAL95]

The contiguous method requires each file to occupy a set of contiguous blocks on a hard disk. This allows quick and easy access to any part of the disk. An application program (operating system) need only know the starting address of the first block and the offset of any datum that is required. This also speeds up the time required to read files, since the

head of the disk need not jump around to find the next piece of information, since all data is contiguous. This is the method used by IBM VM/CMS operating system. This provides good performance, but is may cause numerous block fragmentations. While this solution provides for quick access time, it also produces problems, including external fragmentation and a requirement that the amount of required space be determined ahead of time. [SAL95]

The problem created by contiguous allocation is solved by linked allocation. With linked allocation, each file is a linked list of disk blocks. These blocks may be anywhere on the hard disk. The application (operating system) need only know where the first block resides in order to find a location on the disk file. To create a file, only one block is needed. As the file grows, more blocks are allocated and linked to the last block in the file. With this method there is no external fragmentation nor is there a need to know the size of the file ahead of time. However, link allocation is not very efficient for any purpose besides sequential processing [SAL95]. Another disadvantage of link allocation is the need for space for the pointer. Link allocation also suffers from another problem, reliability. If one of the block pointers is bad, then any part of the file beyond that block is lost. One way to reduce the overhead for block pointers is to gather the blocks into a set called clusters, and allocate clusters rather than blocks. However, the larger the cluster, the greater the internal fragmentation, since no other file could use that cluster [SAL95].

Another variation on the link allocation is the use of file allocation tables (FAT). This is the method used by MS DOS and OS/2 [SAL95]. FAT systems preserve a section at the start of each partition for the FAT table. Each block has one entry in the table and is index by block number. FAT have much in common with link allocation. Each directory contains a table with the block number of the first block of a file. Each partition contains a set of pointers for that partition [SAL95]. This has the advantage of separating the links from the data. In this scheme, only the links need be read to find a particular location on in a disk file. In cases where a file resides only in one partition, random access can be accomplished by first reading the FAT table for the partition, and then following the links until the proper position can be found [SAL95].

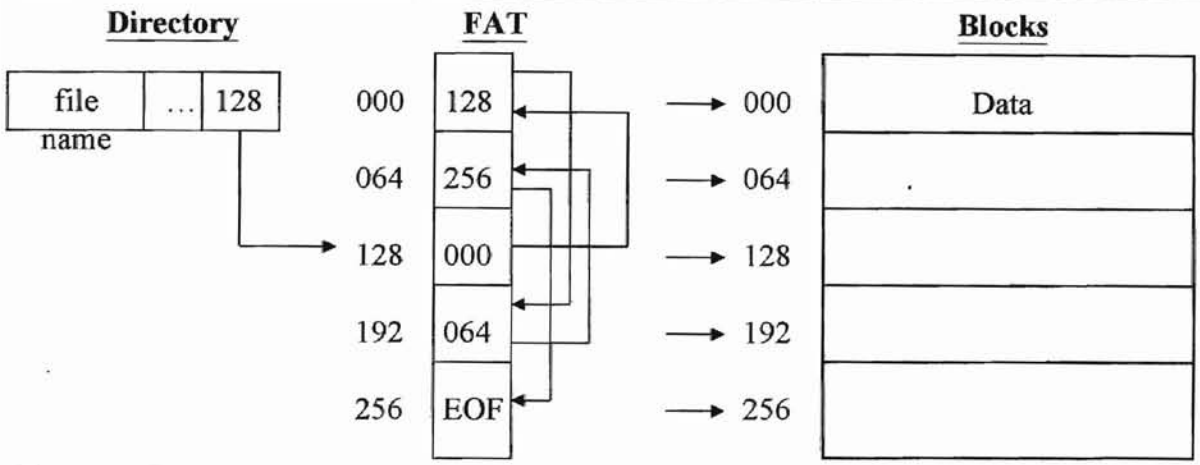


Figure 4: FAT tables

In the diagram above, the directory structure points to the starting address in the FAT table (128). For every entry in the FAT table, there is a corresponding block of the file. This is similar to a node in a tree except that the pointers to the next node are stored in a separate file from the data. So when the directory list point to the first node in the FAT table, it is also pointer to the data block at that address. If each entry in the FAT table is 4 bytes long and each entry in the data blocks is 512 bytes, then the first entry in the FAT table would be at address 128 * 4, while the first entry in the data block would be 128 * 512 [SAL95].

The final operating system storage method to consider is indexed. This is the scheme normally used by UNIX. UNIX uses a structure called an i-node (index-node). This structure store the first few block addresses in the node itself, which is stored in main memory while the file is open. For smaller files, all the block address will be stored in main memory. For larger files, one of the block addresses contains additional block addresses (address to a node instead of a block) called a single indirect block. If even more blocks are required, then another address in the node (called a double indirect block) is used. The double indirect block is a node that contains the address of a block that contains a list of single indirect blocks. If even more space is required, then a triple indirect block is used. [SAL95]

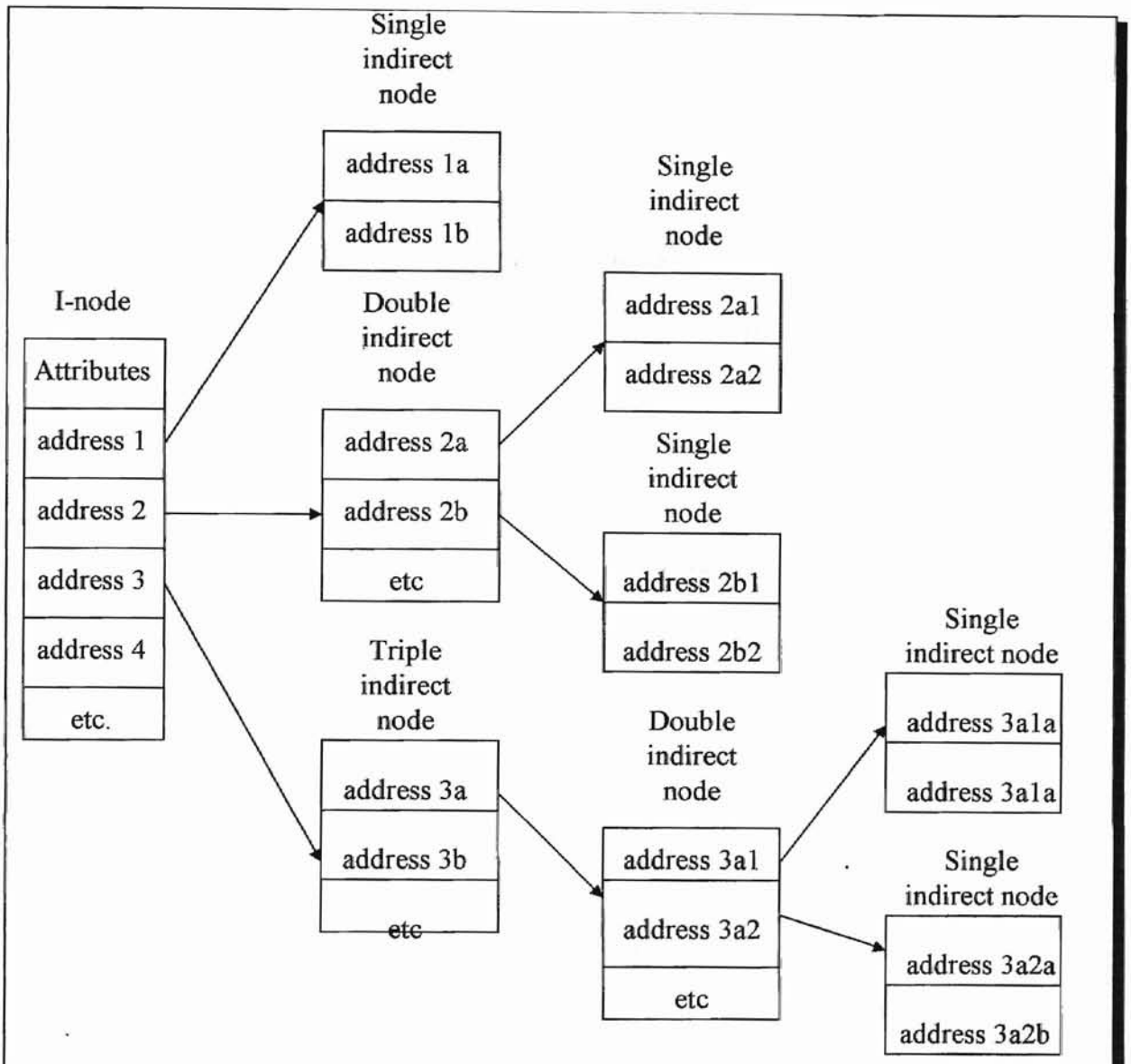


Figure 5: UNIX i-node disk file structure

In the diagram above, the attributes (such as ownership) and the initial address blocks are stored in top node (left most block). Each address is either an address to another node or to a block of data. Inodes form a tree of address with the leaves pointing to blocks of data. This allows for quick location of any data location [SAL95].

2.2 Time Persistent

> A persistent search tree differs from an ordinary search tree in that after an insertion or deletion, the old version of the tree can still be accessed. [SAR86]

In 1986, Neil Sarnak and Robert E. Tarjan published a research paper entitled "Planar Point Location Using Persistent Search Trees." In this report, a portion of the article is written concerning the "persistent sorted set problem."

> "We wish to maintain a set of items that changes over time. The items have distinct keys, with the property that any collection of keys of items that are in the set simultaneously can be totally ordered. (The keys of two items that are not in the set at the same time need not be comparable.) [SAR86]

This article presents three operations on the set of data:

access(Key K, Set S, Time T)	Given a Set S, Time T, and a Key K, return the Item I who's key is $\geq$ K.
insert(Item I, Set S)	Given Item I, Set S, and current Time T, Insert Item I into Set S at current Time T.
delete(Item I, Set S)	Given Item I, Set S, and current Time T, Delete Item I from Set S at Time T. *Note: Item I always have a predefined key. Any update occurs no earlier than any previous operation in the sequence than Time T. In other words, updates are only allowed in the present time.

The article goes on to present two methods of time persistence in search trees. The first method is to copy paths and the second method is to extend the edges so that for any time T that an Item I in the Set S has been changed, there is exactly one set of Edges E.

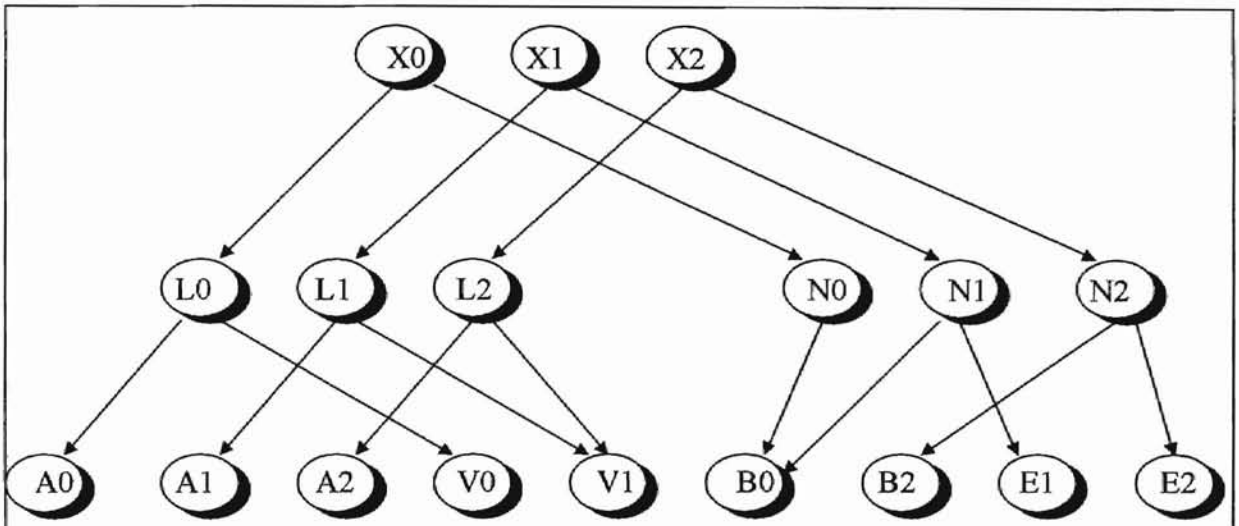


Figure 6: Persistent nodes with path copying.
In the above diagram, all letters represent datums and all numerics represent time. At time 0, there are 6 Datums (X0,L0,A0,V0,N0,B0). At time 1, E1 is added and A, V are modified. At time 2; B, A, and E are modified. [SAR86]

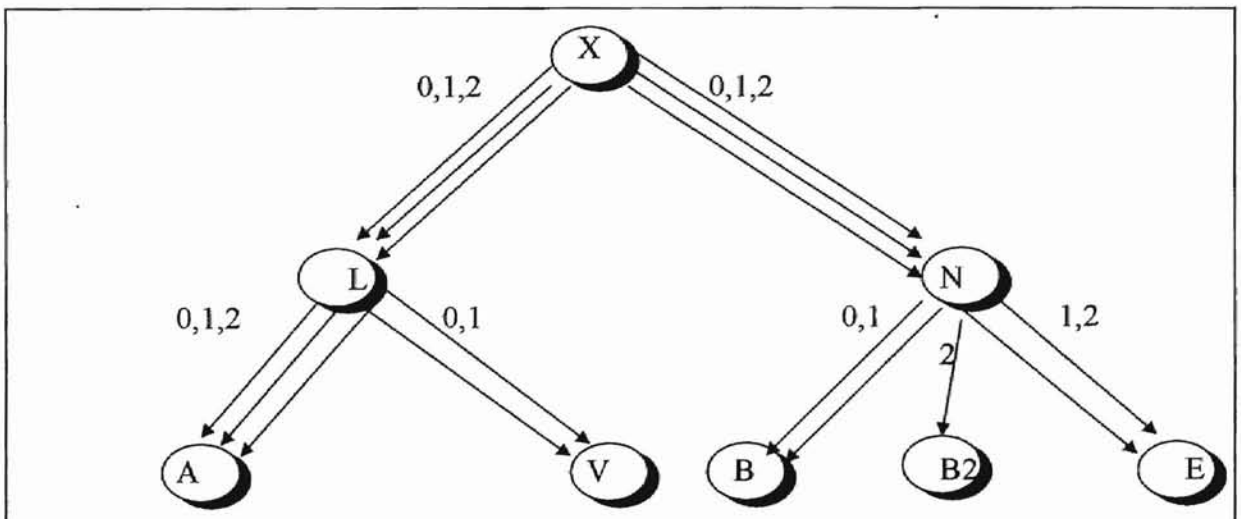


Figure 7: Persistent nodes with edge copying.
In the above diagram, all letters represent datums and all numerics represent time. At time 0, there are 6 Datums (X,L,A,V,N,B). At time 1, E is added, and at time 2, B2 is modified. [SAR86]

A major disadvantage of path copying is that it uses non-linear space. Using edge copying reduces this problem but introduces the problem of fat nodes. For each potential node, there must either be enough space for all edges, or addition nodes will be required if more edges are required. The article goes on to introduce a third alternative: edge copying with limited node copying.

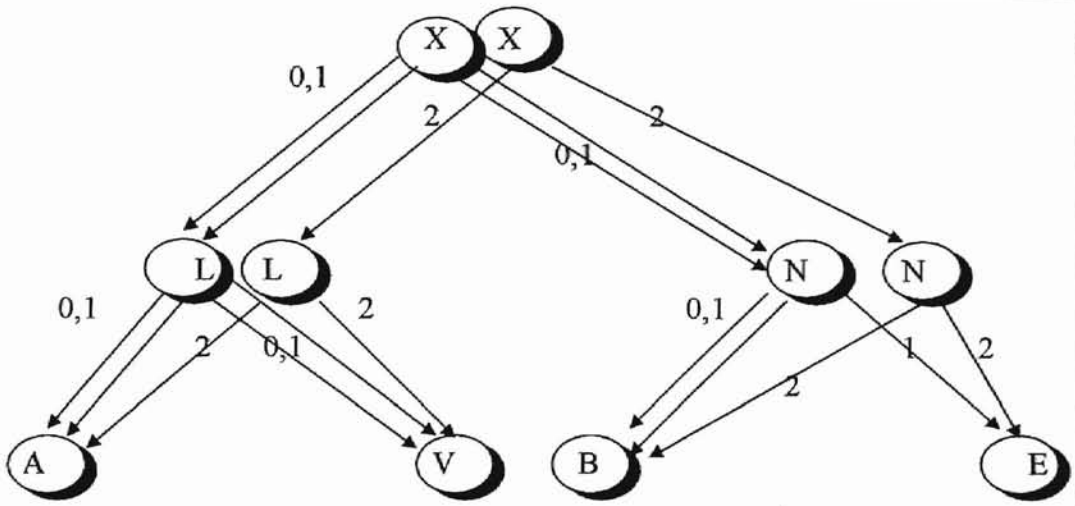


Figure 8: Persistent nodes with limited path copying.
This diagram, all letters represent datums and all numerics represent time. At time 0, there are 6 datums (X,L,A,V,N,B). At time 1, E is added, and at time 2, N and L are modified and E is added. [SAR86]

This last method, Persistent Tree with limited node copying, reduces the overhead of path copying at the same time limiting the size of a node (reducing the size of a fat nodes). (See [SAR86])

3. Problem Domain

3.1 Challenges facing Data Allocation Engines

In this document, we will be referring to records. A record for sake of discussion is any set of data that is organized and stored as a whole. This data may be relational, object, hierarchical, textual, etc.

Two of the greatest challenges facing data allocation engines for databases are fragmentation and seek time. Fragmentation can be either internal or external. Fragmentation is any disk space location that is not currently being used. This fragmentation can either be useable space or unusable. Internal fragmentation is any free space that is associated with a record such that no other record may use that space. External fragmentation is any free space that is available for any purpose. External fragmentation is normally usable unless it is too small. Internal fragmentation is almost never useable. [FOL92]

For purposes of this report, seek time is defined, as the number of reads a disk must perform to retrieve a record. The more seeks that are performed, the slower the database will run. To reduce seek times, it is important to group related information together as much as possible.

3.2 Design issues of Data Allocation Engines

A data allocation storage engine is a software layer, program, sub-program, or function(s) whose purpose is to store data structures in any storage medium (such as a hard disk). The data allocation storage engine provides the logical storage mechanism to perform the process to manage physical storage. This storage could be the file/directory structure of an operating system or a database. Data allocation storage engines free calling programs from having to construct the physical organization of a file on a hard disk. For instance, imagine if every program had to be concerned with allocation of blocks and sectors on a hard disk, and linking them together. This would require redundant work for each program. Instead, the operating system provides storage protocols for programs to use. An example of this is the storage structures used by UNIX. From the operating system point of view, data is broken into blocks and are then linked together [SAL95]. From an application point of view, data consist of one large chunk of physical address space. The application is oblivious to the fact that blocks of data may be physically stored in various parts of the hard disk. To the application, the data is contiguous. To the operating system, the data (file) is a series of linked blocks of data. Data allocation storage engines have three primary designs: fixed formats, buckets, and variable length. Fixed formats means that all data is grouped in a format with structures of the same length. Variable length formats means that data is stored in variable length formats. Buckets means that a record is sub divided into chunks (blocks) and divided between several buckets [FOL92]. These blocks of data are then linked

together to form one record. With fixed length format and buckets, there is internal fragmentation but no external fragmentation, since all data is organized in a fixed format of the same length [FOL92]. In variable length formats, there is external fragmentation but no internal fragmentation, since a group of data will only take up the amount of space that they require. Variable length formats tend to have more over head dedicated to tracking and reducing fragmentation. Fixed formats tend to have a greater loss of space, since all records must be stored in the same amount of space regardless if they are the largest possible record or the smallest. Buckets formats have more overhead in order to track pieces of the data and some internal fragmentation. With buckets, there is a trade off between loss of space due to internal fragmentation and loss of space in order to save links between buckets [FOL92].

In examining the advantages and disadvantages of variable length format, fixed length, and buckets, it was decided to go with buckets.

With buckets, the larger the structure for storing the data, the more internal fragmentation and the less overhead involved. On the reverse side of the coin, the smaller the storage structure for data, the larger the overhead.

3.3 Determining the organization for a Data Allocation Engine

In determining an appropriate organization structure for any data allocation storage engine, it is important to take into account performance criteria. While speed is a primary concern, reduction in fragmentation is also critical.

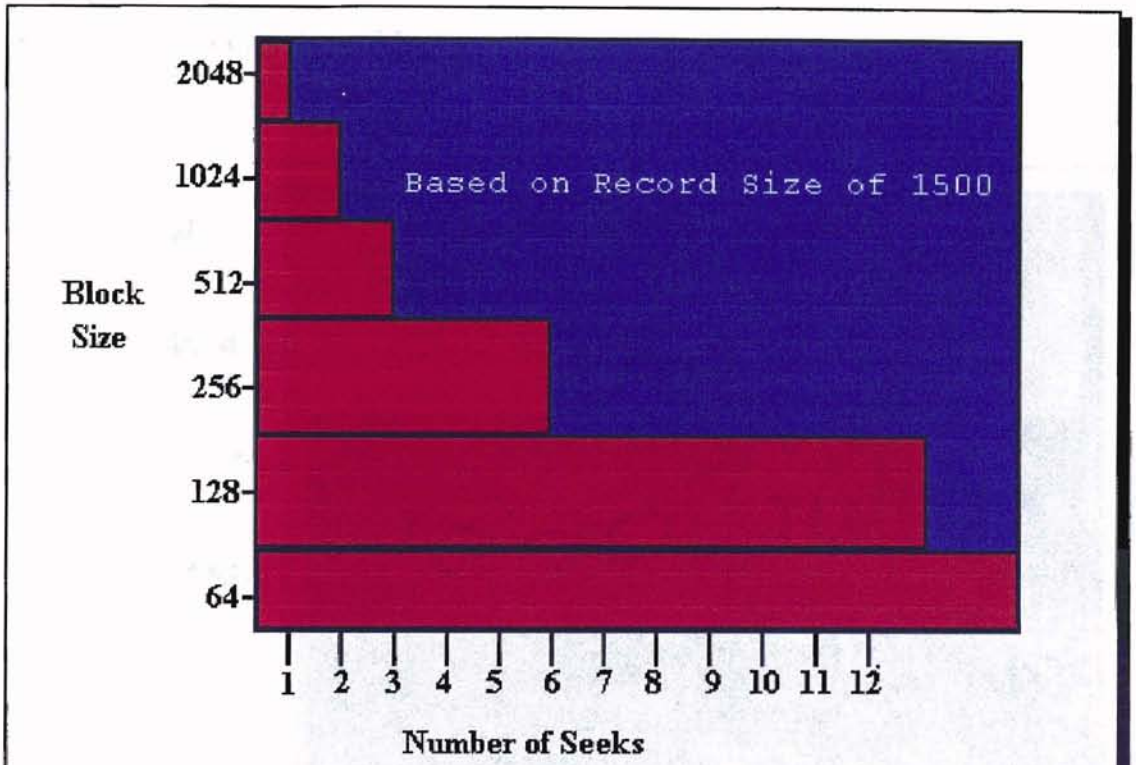


Figure 9: Seek Time graph

The above graph illustrates the time required to read a record of 1500 bytes based on the data divided into blocks. This chart only looks at usable space in each block and does not include the overhead required linking the blocks together. If a block size is 2048, the 1500 bytes will fit into the block with 548 bytes of unused space, and will require only one seek (read). With block size of 1024, 2 blocks would be required to provide enough space for 1500 bytes (2 seeks). Using the following formula, the number of seeks can be calculated: Integer [((Record Size-1)/Block Size)+1]. [Original work]

As can be seen in figure 9 above, the larger the block size, the smaller number of seeks necessary to read a record. By reducing the number of seeks on a file, the access time is also reduced, producing desirable access time. If access time were the only criteria, then large block sizes would be highly desirable. This assumption ignores space utilization. If we take a look at space utilization, or in other words, fragmentation, we get a very different picture [Original work].

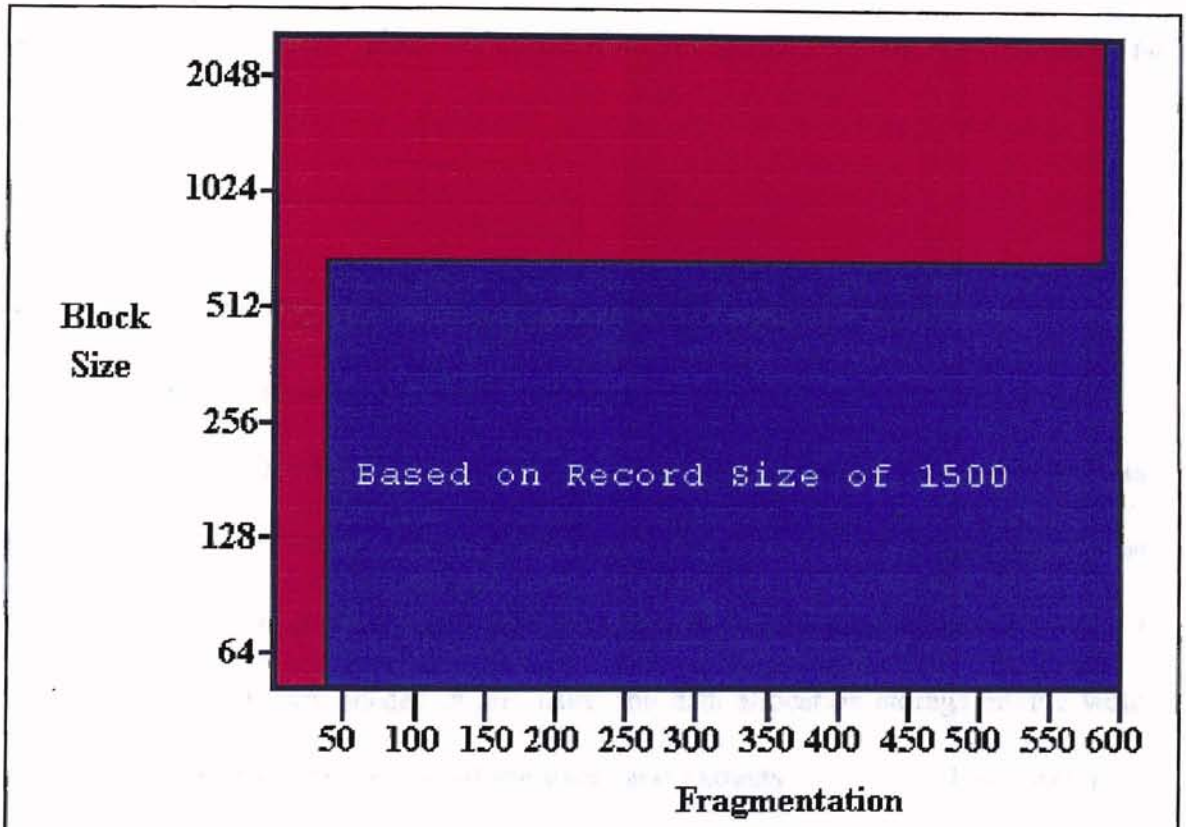


Figure 10: Disk Fragmentation graph
This diagram demonstrates the fragmentation based on block size for a record of 1500 bytes. Until the block sizes approach 1024 bytes, the fragmentation remains low. However, as the block sizes passes 512 bytes, the fragmentation increases dramatically. ([Block Size] – Remainder {[Record Size] divided by [Block Size]}). [Original work]

As can be seen in the above chart, larger block sizes produce greater fragmentation. If we compare the graph in Figure 10 with Figure 9, we can see that the optimal block size for a record size of 1500 is 512 bytes (based on multiples of 64 bytes). This block size requires only 3 seeks, while only having 32 bytes of fragmentation. If we knew that all records would be 1500 bytes, we would, of course, make all block sizes 1500 bytes.

What about when record sizes range from very small (less than 64 bytes) to very large (greater than 2048 bytes)? Given the information in Figures 9 and 10, there would be no optimal block size.

3.4 Issues relating to organization of Data Allocation Structures

[*Note: The following is Original work].

Suppose we group blocks together, we can then have the advantage of large and small blocks. This group of blocks we will call a page. Then each time the database needs one or more blocks in a page, it can read the whole page and extract the necessary blocks. In this way, if n blocks are needed in any page, the data allocation storage engine would only have to perform one seek, read the page, and extracts n blocks. If we now use a small block size combined with large page size, we can now have the advantage of having large and small block sizes. Let us consider the case of a page that is 2048 bytes long, structured with blocks 128 bytes long. We would then have less seek times and less fragmentation (assuming that the blocks that contain the record are stored in the same pages).

Can it be that simple? In a word, No! Consider a record that is 1500 bytes long. This would require 12 blocks with 60 bytes of fragmentation (each block consisting of 120 bytes of data space+8 bytes of overhead for linking. 12 * 120 = 1440. Then the last block would hold 60 bytes of data space, 8 bytes overhead and an internal fragmentation of 60 bytes.) If each page contained 16 blocks (2048 / 128), we would have 3 blocks left over. If we use these 3 blocks plus 9 others for the next record, we have two seeks instead of one. While not ideal, still acceptable. So far, there is no real problem. But suppose we had records of different sizes, for example 1500 bytes and 900 bytes. Let us call the first record, record A (1500 bytes), and let us call the second record, record B (900 bytes). Record A would require 12 blocks while record B would require 8 blocks with 124 bytes of fragmentation. If we now add to the database structure, one record type A and one record type B, then the first page would require 12 blocks for record type A and 3 blocks for record type B. Then in the second page, record type B would utilize 5 more blocks. Now let us add another record type A. This would require the 11 remaining blocks in page 2, plus 1 block in page 3. So far, every thing is okay. But, suppose we delete the record type B and add another record type A. This next record type A could take up the blocks left by the deleted record, this would mean that this 4th record would take up 3 blocks in page 1, 5 blocks in page 2, and 4 blocks in page 3. As you can see, the more diverse the record sizes, the more seeks that would be required. In the end, we have gained little by organizing blocks into pages.

Instead of allowing record type A to take the blocks freed when record type B was deleted, suppose we only allowed record type A to reuse blocks that were freed when other records of type A were deleted. In this case, we are assured that for records less than 2048 bytes, there can be no more than two seeks and no more than 127 bytes of fragmentation per record. Let use now take this a step further, suppose that we kept track of all pages that had no used blocks. In this case, a page whose blocks were all available could be used for any record types.

Let us now consider another scenario. Suppose we a have a set of pages that was utilized by record type A above. Let us further assume that we have 12 more records. The first record would be in page 1 and would take up 12 blocks. The second record would start in page 1 and take up 4 blocks, the remaining blocks would be located in page 2 (4 blocks). The third record would take up 12 blocks in page 2, and so forth. Now suppose that ever-other record was deleted. That would result in a significant number of free blocks in the database.

Suppose that instead of tracking open blocks by data type, we track the number of open blocks per page. Let us further gather a set of pages into a group and call it a cluster. The first page in each cluster can contain a set of fields that track the used and Free blocks in each page of the cluster. When a record needs a set of blocks, it can read the first page in the cluster, determine a suitable set of blocks, place the records in the appropriate blocks in the page(s), and mark the blocks as used. If there is no suitable sets of blocks in this cluster, the record can then expand its search to other clusters, and if

necessary, allocate a new page. If we further implement a rule that says that a record of 1024 bytes or less can not span more than two pages, we would reduce the seek time to a maximum of two seeks per record.

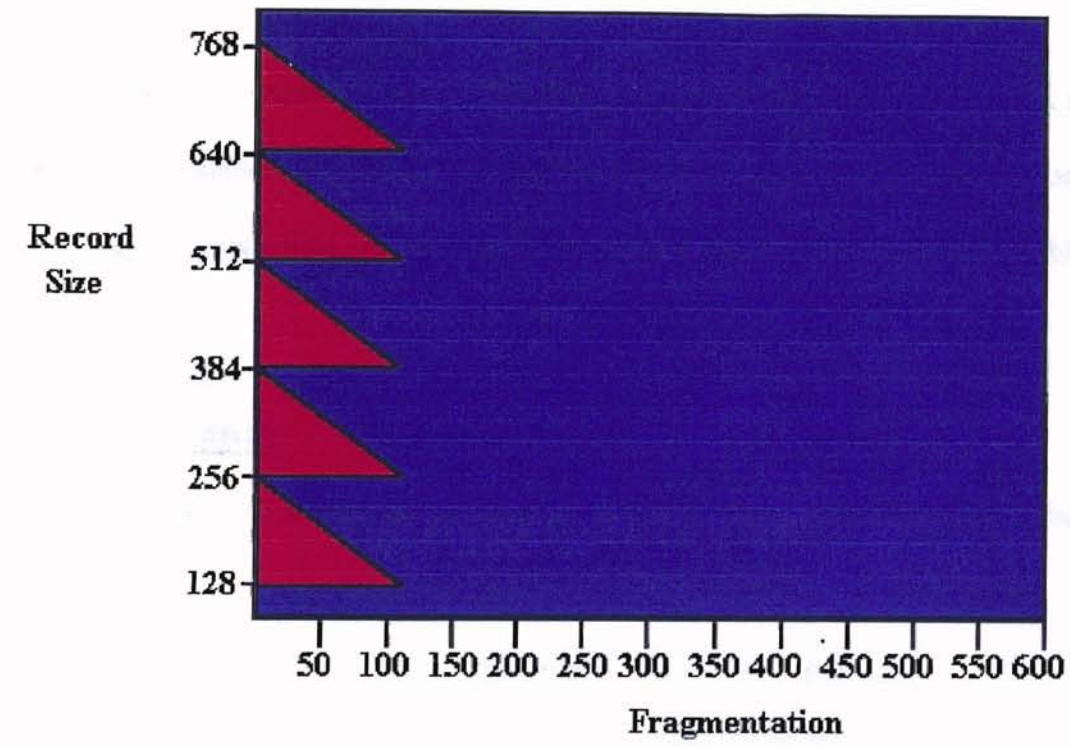


Figure 11. Disk Fragmentation using Blocks, Pages, and Clusters.
The above graph is based on block sizes of 128 bytes with 120 reserved for data and 8 bytes for block pointers to link blocks together. [original work]

As can be clearly seen from Figure 11 above, by using clusters, pages, and blocks, the amount of fragmentation can be greatly reduced. In this schema, the range of fragmentation ranges from 0 to 127 bytes. This of course is a desirable outcome, but does it come without a cost. Once again, the answer is no. In order for 2 or more blocks to maintain a relationship, they must have some way of tracking each other. This can be

accomplished by using block pointers. Block pointers would require an overhead of 4 to 8 bytes per block. This means that the system will require between 0 to 6.25% overhead (0 to 3.125% for 4 byte pointers) not counting overhead for cluster which can bring overhead up to as much as 8%. Reducing the amount of potential fragmentation (as shown in Figure 11 above) will offset the loss of usable space due to overhead.

Reducing the potential fragmentation is still only part of the picture, we still must look at seek times. If we maintain the rule that no record of 1024 bytes or less, can span more than two pages, then seek time will always be between 1 and 2 seeks, this is a highly desirable outcome.

3.5 Time Persistence

One of the issues that arise in databases is time recovery. In many cases, the users either wants to look at some past historical information or produce reports, or back erroneous data out. Or in other cases, users may wish to go back to a certain point in time, use the database, and then return to the present. For whatever reason, one useful tool a database can have is time persistence. One of the goals of this project will be able to roll back the database to any point in time, and begin processing at that time. At some point the user may decide to continue processing at the new time line, return to the old time line, or update the new time line with data that was previously entered. In order to accomplish these tasks, time persistence will be used. [SAR86]

3.6 Conclusion to the Problem Domain

In looking at possible structures for data allocation engines, it appears that an organization that consist of clusters, pages, and blocks will meet the previous stated performance criteria.

4. Solution Domain

4.1 Using blocks in a Data Allocation Storage Engine

A block is used to store a sub-set of a database record. Each block consists of 'B' bytes of user space and eight bytes of system space. For discussion sake, let 'B' bytes equal 120 bytes. These blocks are logically chained together in order to provide enough storage to save the record. In each block, the first four bytes are reserved in order to provide a pointer to the previous block. The next four bytes are reserved in order to provide a pointer to the next block. If either block points to the zero address, the pointer indicates the beginning or end of the record.

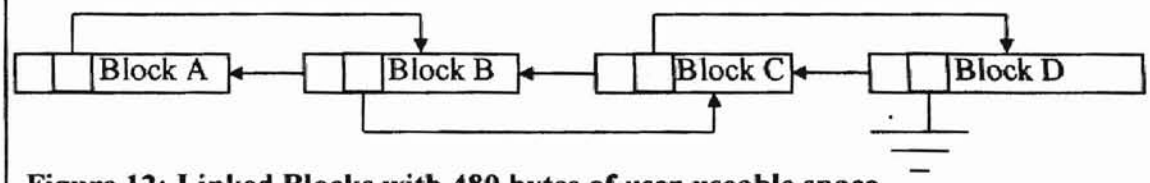


Figure 12: Linked Blocks with 480 bytes of user useable space
The above figure illustrates how blocks linked together provide the necessary storage room for a record. (128 bytes – 8 bytes for links) * 4 blocks = 480. [Original Work]

4.2 Collecting blocks into pages

[Original Work]

A page is a collection of blocks. For discussion purposes only, let each page contain 16 blocks. Each page therefore is 2048 bytes long consisting of 16 separate blocks (16 * 128 bytes).

Page N

Block 01	Block 02	Block 03	Block 04
Block 05	Block 06	Block 07	Block 08
Block 09	Block 10	Block 11	Block 12
Block 13	Block 14	Block 15	Block 16

Figure 13: Page structure consists of 16 blocks of 128 bytes each

Each page is a physical representation of data storage allocation. In figure 3 above, the blocks were logically linked together. In figure 4 above, the blocks are physically positioned together. So that if we look at the blocks in figure 12, "Block A" may be the 10th block in the 2nd page, while "Block B" may be the 3rd block in page 5.

4.3 Using clusters to organize pages

[Original work]

A Cluster consists of a header page that relays availability of blocks in pages. Each cluster consists of 1048 frames. Each frame is 2 bytes long consisting of 16 bit flags. Each bit flag indicates, if true, the block is available.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	0	1	1	0	0	1	0	0	1	1	1	0	0	0

Figure 14: Bit map frame of a Cluster

Blocks 1,4,5,8,11,12,13 are in use, and blocks 2,3,6,7,9,10,14,15,16 are available for use. Each Cluster has a head cluster with 1024 frames. Each frame represents the availability in blocks in a page. [Original Work]

As you can see in Figure 8 above, Frames 1,4,5,8,11,12, and 13 are available for use while frames 2,3,6,7,9,10,14,15, and 16 are used. This indicates that there are 7 blocks available in the page and 9 blocks that are used. This means there is 840 bytes of user available space (7*(128-8)). Each Cluster consists of 1048 frames as described in Figure 9 below.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
Etc.	.....						
1017	1018	1019	1020	1021	1022	1023	1024

Figure 15: Cluster Header pages consisting 1024 frames. Each frame represents the availability of blocks in each page in the cluster. [Original Work]

By using the frame bit map of any page in the Cluster, the system can quickly indicate which pages have enough empty blocks to fulfill the needs of a record in need of storage space.

The cluster, page, and block format appear from all analysis too significant reduce fragmentation in the file. However, it does have a cost of 8 bytes of system allocation for every 120 bytes of user space requirements. This has been deemed acceptable.

4.4 Reduction of Seek Time

[Original Work]

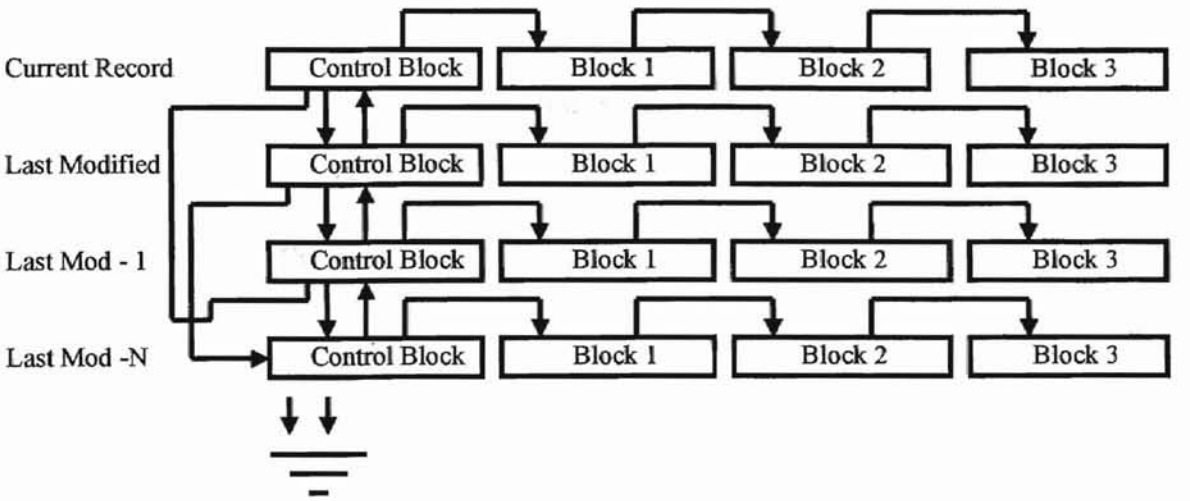
While this appears to resolve the fragmentation issues, it does not address the seek time issues. During the Analysis phase, it has been determined that no more than two seeks

per record of 1024 bytes or less is acceptable. Based on this determination, it has been concluded that a record of 1024 bytes or less may not span more than two pages. In the event that the current cluster is unable to provide the required block sized pages, the next cluster will be queried until an appropriate size combination is produced. In general, the number of pages a record will be allowed to span can be computed with the following equation. Let R = record size, B= usable bytes per block, N = number of blocks per page, and P = the number of pages that a record is allowed to span. Then let P = Maximum of (Minimum of R/((B*N)/2) and R/2) and 1. Test results indicate that this solution for random set of records will produce a very high utilization of blocks per page while keeping the seek time down. Test conducted with all records of equal size, results showed as R approached B*N, block utilization dropped. In a worse case scenario, up to 25% of blocks per page were never utilized. Once R = B*N, utilization become 100%.

4.5 Time Persistent

[The following is based on the work of Neil Sarnak and Robert E. Targen --SAR86].

Let each Data Set (Record) start with a control set (1st block) that consists of a pointer to the start of the current data source, first/last data source before modification, pointer to the descriptive format (class definition), last modified by, modified date, etc...



Control Block Fields

Field	Bytes	Description
Control Block	1 Bit	1st bit indicates that it is a control block
Dependency	31 Bits	Count of Data Sets dependent on this one
Data Set Type	2	Type of Data Set (Class, Object, etc.)
Definition	4	Pointer to definition of Data Set
Data Source	4	Pointer to first block in Data Set
Next Modified	4	Pointer to the Next (newer) modification
Prev Modified	4	Pointer to the Previous (older) modification
Last Modified	4	Oldest Modified Data Set
Date Modified	4	Date stored numerical (12101996)
Time Modified	2	Data stored numerical (2359)
Name Modified	N	Identification of Person who made modification

Figure 16: Time Persistent organization.

The Control Block requires that all blocks that are not control block set their first bit to 0. This will cause no problem since the first 8 bytes of each block are used to link blocks together. The first 4 bytes are used to link the previous block and the next 4 bytes are used to link the next block. Since each block is 128 bytes long, the block pointer represents the block location rather than byte location. This means that the block pointer has 7 bits that are not required in a 32 bit system (2^7 = 128). [Original Work, based on the work of Neil Sarnak and Robert E. Targen [SAR86]]

4 . 6 Data Allocation Storage Engine Function Pseudo code

The following is a list of function calls that the DASE will use as interfaces. These are the calls that the interfacing databases will use.

Function **Pseudo Code**

Create
```
if file does not exist
        Create file
        build control file
                build Control Record
                        # of Clusters  = 1;
                        Indicator for DASE
                        # of Pages in last cluster  = 0;
        Write control file
        build Cluster 0
                Set all bit flags to 0 (all pages unused)
        Write Cluster
        Close Data File
else
 return error
```

Open
```
if file exists
        Read Control Record
        if Control Record DASE
                Return no error
        else
                 return error
else
        return error
```

Insert
```
if file is open
        Compress data
        Split data into blocks
        find a page with (# of blocks)/2 free
        find another page for the rest of the blocks
        Insert the blocks into both pages
                adding internal links to each block as we go along.
        Return the address of the first block.

Else
        return error
```

Function	Pseudo Code
Read	if file is open using the address supplied, read the first page. If address points to a valid block Extract all blocks from page and build compress data set. Continue on to the next page until all blocks are read and the compressed data set is built. Decompress data set return a pointer to the decompressed data set else return error else return error
Update	Destroy previous data set if successful destroy Compress data set split data set into blocks write the first block at the same location as the old first block Fill the rest of the in the current page with as many blocks as possible Find another page with appropriate number of free blocks write remainder of blocks to second page else return error
Close	Close the data file
Destroy	if file is open Open first page pointed to by address If it a valid record set mark all blocks in this page and connected pages as free else return error else return error

4 . 7 Class/Object Relationships

In developing this system, an Object-Oreinted approach was utilized. The software was first broken into two sub-parts, the first part (left) dealt with the clusters, pages, and blocks (and disk access), and the second (right) dealt with assemble/disassemble of the source data (document/record, etc.) The following chart is a diagram of the relationships of classes/objects in this project.

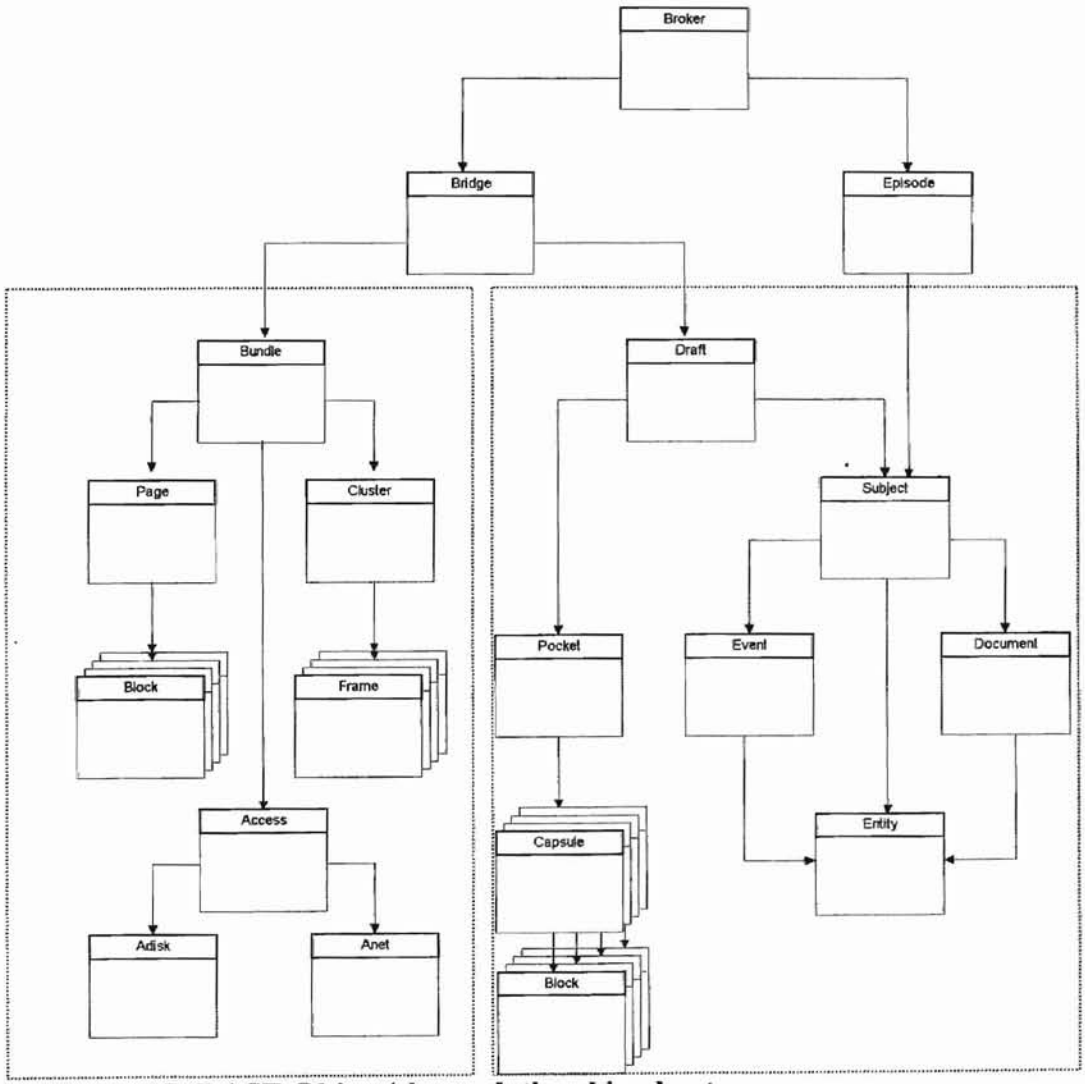


Figure 17: DASE Object/class relationship chart
[Original Work]

In the above chart (Figure 17), The Broker is the main interface and provides and external application program interfaces (API). The Bridge provides a linkage between the Draft and the Bundle. The Bundle coordinates DASE disk structures and storage (clusters, pages, blocks, etc.). The Draft takes as input a series of bytes, (such as a record), breaks them down to blocks, or takes blocks and assembles them into the original record. The Page is a representation of a page in the DASE and stores a collection of blocks. The Cluster class is representation of a cluster header page and stores a collection of frames. The Block class represents a block of data (128 bytes). The Frame class represents a frame structure. Access class provides operating system level access to the hard disk or network. Adisk class is used to physically store information on the local computer, and the Anet class is used to store information on a foreign computer. The Subject class is an instance of the current datum (record) and information concerning the event instance (time space information). The Document is the actual data to be stored or accumulated. The Pocket is a collection class of the Capsules. The Capsules are wrappers containing a single block.

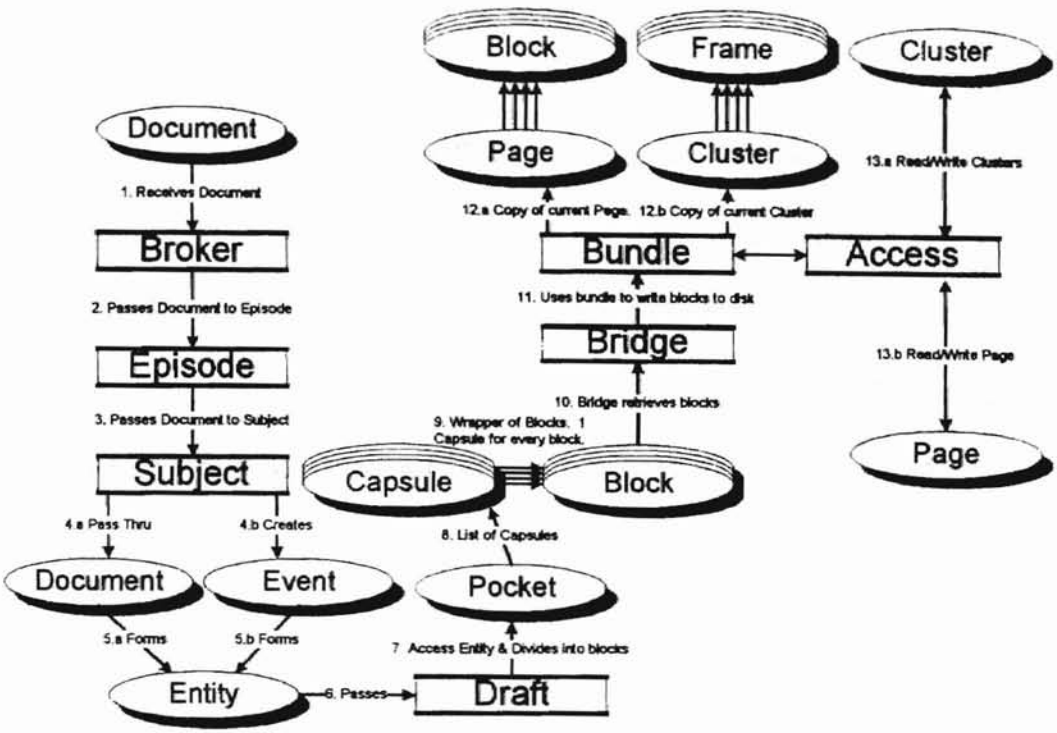


Figure 18: DASE Object flow chart (Insertion or updating a record).
[Original Work]

In figure 18 above, the flow of a document being added (or updated) is diagramed. A document is a series of contiguous bytes (such as a record). The Broker receives the document from an external source and sends it to the episode, which in turn sends it to the Subject. The document is then temporarily stored and an event is created. An event is a structure that tracks time dependent information of a document. The event and the document are then merged to form an Entity. The Bridge is then called by the Broker and told to process. The Broker calls the draft and requests that the Entity be disassembled into blocks. The blocks are then wrapped by a Capsule to form a link list. The Pocket is then

created to store the root pointers to the Capsules. The Bridge then queries the Draft for the total number of blocks. The Bridge then calls the bundle (who calls the page) and request the number of blocks per page. The Bridge then calculates the minimum number of Blocks that must be stored per page. The Bridge then request that the Bundle locate a Page that meets the minimum requirement of available Blocks. Once an appropriate Page is located, the Bundle returns the count of the number of available Blocks to the Bridge. The Bridge then continuously queries the Draft for the first/next block until either all of the available blocks in the page are used, or there are no more blocks to be stored. As each block is stored in the page, the associated Frame in the Cluster is updated to reflect the fact that the block is now in use. Whenever a new page is read from the disk (or created), the old page is first queried to determine if it is dirty (has been altered in some way), and if it has been, it is rewritten to the appropriate location. Whenever a new page is retrieved, the bundle determines which cluster it belongs to. If it belongs to a cluster that is not currently residing in the Bundle, then the Bundle will query the Cluster to determine if it is dirty. If it is, it is rewritten back to the disk file and the new appropriate cluster is read and placed in the Bundle.

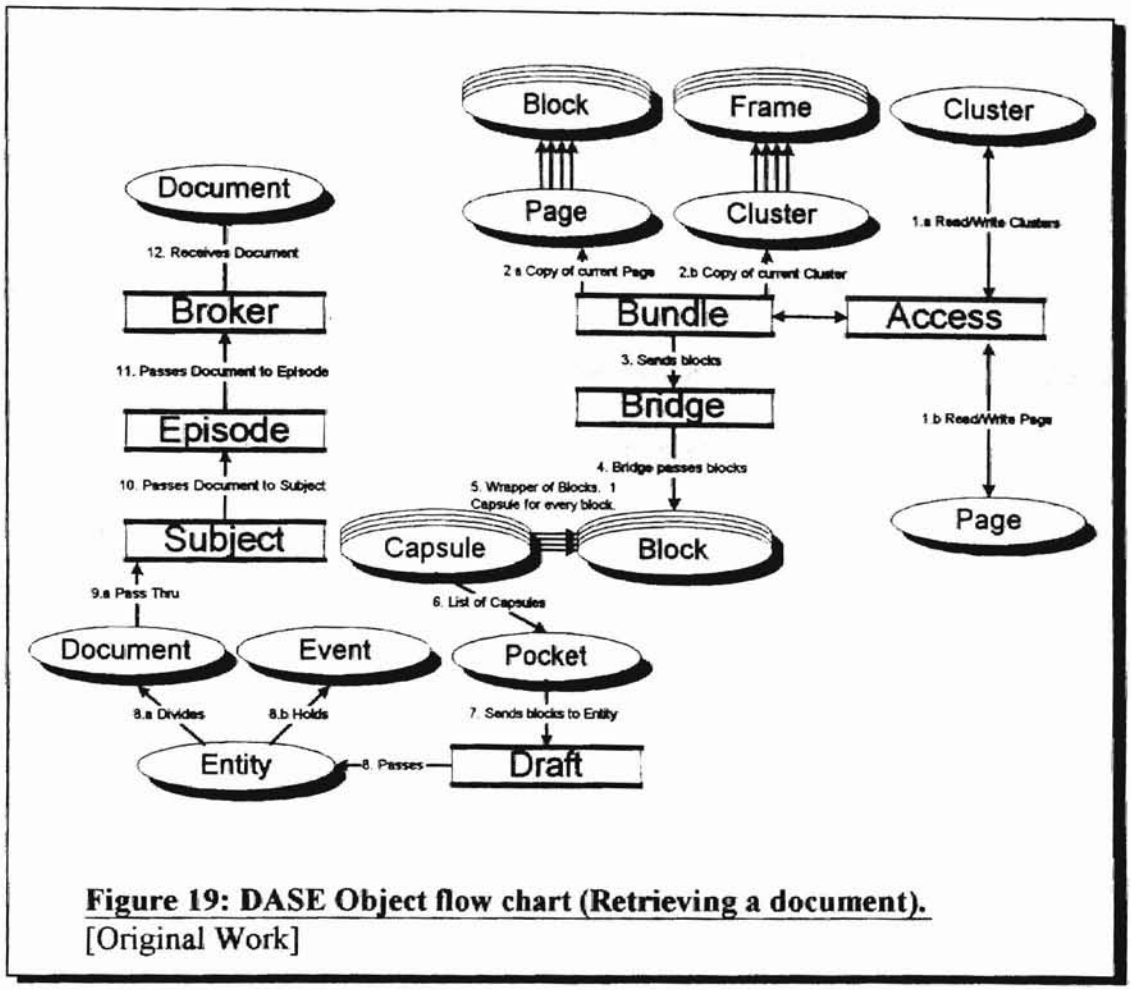


Figure 19: DASE Object flow chart (Retrieving a document).
[Original Work]

In figure 19 above, the flow of a document being retrieved is diagramed. A document is a series of contiguous bytes (such as a record). The Broker first receives a request for the document at a particular location from an external source. The request is then sent to the Bridge. The Bridge then calls the Draft and sends a message to prepare for a retrieval process. The Draft cleams out the packet and all of the Capsules and associate Blocks are deleted. The Bridge then request that bundle retrieve a block at the location received from the external source. The bundle examines its current page and determines if the block is located in that page. If it is, it returns a pointer to the block. If it is not at that

location, the page is queried to determine if it is dirty. If it is, it is rewritten to the disk file and the appropriate page with the requested block is read. (note: If the requested page is in a different cluster other than the one currently in the bundle, the cluster is queried to determine if it is dirty. If it is dirty, it is written to disk and the appropriate cluster for that page is read into memory). The appropriate block pointer is then passed to the Bridge. The Bridge extracts the next block pointer from the block and then sends the block pointer to the draft. The draft then copies the block, wraps it in a Capsule and appends it to the end of the pocket. The Bridge then examines the next block pointer. If the next block pointer is > 0, the Bridge repeats the process of requesting a block from the Bundle and passing it to the Draft until all blocks have been read. Once all blocks have been read, the Bridge request that the Draft assemble the document. The Draft will extract the data sub record from the blocks and will build the Entity. The Entity is then split into an Event and a Document. The Bridge then releases control to the Broker. The Broker then calls the Episode and requests the document. The Episode in turn calls the Subject, who returns a pointer to the document. This document is then passed up to the Episode who in turn passes it to the Broker. The Broker then makes a copy of the Document and passes it to the calling process.

4.8 Class/Object Definitions.

The following diagrams depict the internal functionality and storage of each of the main classes in DASE.

Broker	
Methods:	**Objects**
Public: Create Open Destroy Close Insert Read Update	Private: oBridge oEpisode

This Class is the main entry point into the DASE system. It provides the Application Program Interface (API).

Bridge	
Methods:	**Objects:**
Public: Read Update Insert Access	Private: oBundle oDraft

The Bridge coordinates the efforts between the Draft and the bundle. Its purpose is to coordinate the process of either receiving a data stream, breaking into blocks and storing it, or retrieving blocks, assembling them and return the results to the calling process.

Bundle	
Methods:	**Objects**
Public: NewBlock GetBlock SetBlock NextBlock FreeBlock	Private: oPage oCluster oAccess

Access	
Methods:	**Objects:**
Public: Path Type Open Write Read Create Delete	Private: oAdisk oAnet

The Bundle coordinates the work of the Pages, Clusters, and the Access objects. The current working Clusters and Pages are stored in the Bundle, and the Access Object is used to read or write the Cluster or Page to the hard disk.

Access is used to read or write data to the disk file. This class removes the need for the Bundle to read or write directly.

Page	
Methods:	**Objects**
Public: Get Save IsClean( ) IsDirty( ) Clean( ) Static: Size Blocks Blocks(Count)	Private: oPage oBlock Static: cBlkCount

Block	
Methods:	**Objects:**
Public: Build SetNext GetNext SetData GetData CpyData CpyBlock Operator = IsClean IsDirty Clean Static: BlockSize DataSize Overhead	Private: oBlock oNext oBlkID oData oStatus Static: cBlkSize cBlkID

The page holds the current page data, the status of the current page (Dirty or Clean) and the number of blocks in a page.

Note: A Dirty page is a page which has been modified since it was either created or read from the hard disk, but which has not yet been written to the hard disk.

Blocks contain the sub records of a document. The are also capable of extracting the address of the next block in the link. Like Pages, they also know if they are dirty.

They also contain the Configured block size and the next block ID.

Cluster	
Methods:	**Objects**
Public: GetMin GetFrame Build Static: Size Count MetaBuild	Private: oFrame oData Static: cFrameCount

Frame	
Methods:	**Objects:**
Public: Check UnCheck InUse IsFree Clean IsClean IsDirty NoOfFree Static: MetaBuild Size Private: SetBit GetBit	Private: oFrame oStatus Static: cFrameSize

The Clusters contain the frame data as well as a pointer to the individual frames. They also hold the number of frames per cluster.

The Frames are used to determine if the block is in use or not. For each block in the Cluster (the cluster here is a set of cluster header and Pages, not the cluster structure), there is one bit in the frame. If the block is in use, and then bit is set to one, otherwise, it is set to zero.

Adisk	
Methods:	**Objects**
Public: Path Type Open Write Read Create Delete	Private: oFILE

Anet	
Methods:	**Objects:**
Public: Path Type Open Write Read Create Delete	Private: oSocket

The Adisk is used for accessing data from a disk file. Access must use Adisk to access data.

This class is currently not in use in DASE. It was left in the design for future reference. When implemented, it will supply the ability to read web pages off of the Internet.

Episode	
Methods:	**Objects**
Public:	Private:
Receive Retrieve Backout CheckPoint	oSubject

Draft	
Methods:	**Objects:**
Public:	Private:
Assemble Disassemble GetNext AppendBlk	oPocket oSubject

The Episode is used as a process for passing the document to the subject and requesting a checkpoint or a back out. If a checkpoint or a back out is requested, then either the previous data is read from the disk for the current document (or the previous versions of the document are deleted).

The Draft either takes an Entity and breaks into blocks, or takes a series of blocks, and assembles an Entity.

Pocket	
Methods:	**Objects**
Public:	Private:
Append Next Head Clear	oHead oCurrent oTail

The Pocket is a link list of Capsules.

Capsule	
Methods:	**Objects:**
Public:	Private:
GetBlock SetBlock NextBlk	oNext oBlock

This object is a node in a link list that contains a pointer to a block.

Subject	
Methods:	**Objects**
Public: SetDocument GetDocuemnt GetPrevious	Private: oEvent oDocument oEntity

Event	
Methods:	**Objects:**
Public: DataType Source Next Prev Last Date Time Name	Private: Type Bit Dependencies Data Type Data Source Next Modified Prev Modified Last Modified Date Modified Time Modified Name Modified

This object provides the process of either merging an event and a document to form an Entity, or the process of taking an Entity and breaking it into an Event and a Document.

The object controls the time sensitive information of an Entity. By using the Next and Previous links, this class allows the system to go forward (Next) or backwards in time (Prev).

5. Practical Applications of Data Allocation Storage Engines

5.1 Interfacing with the Data Allocation Storage Engine

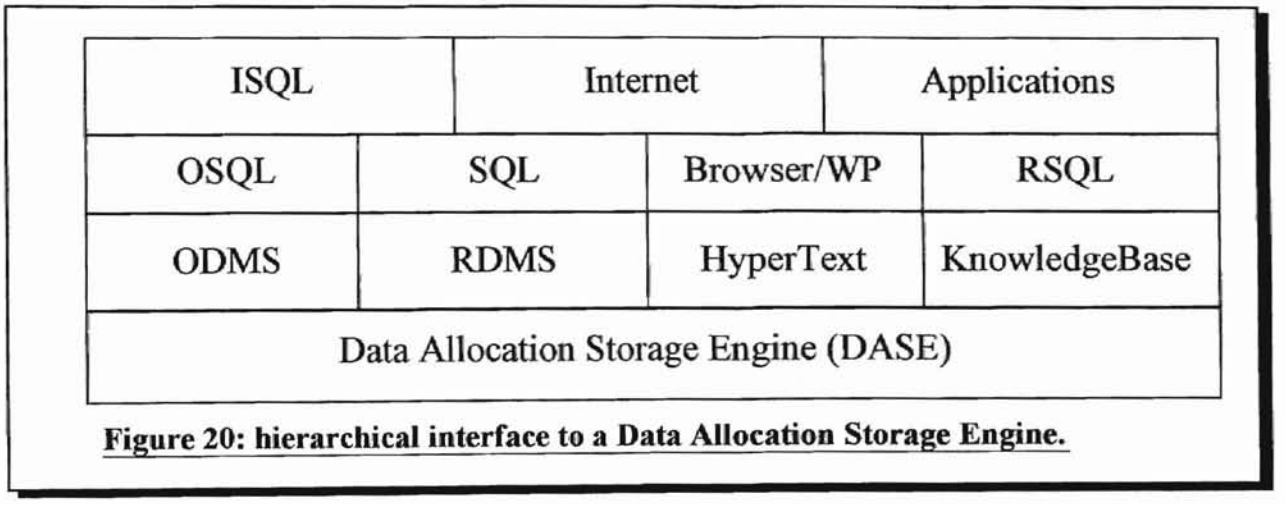


Figure 20: hierarchical interface to a Data Allocation Storage Engine.

In figure 20 above, we show a possible hierarchical view of layers of an intelligent database. Here we can see that 4 types of databases use the Data Allocation Storage Engine (DASE). Each of these databases has their own unique formats for data. While this is a small collection of possible usage of a DASE, they are by no means exhaustive. The DASE provides several general functions that all databases use. Some of these functions are:

Function	Purpose
Create	Create a disk file for the DASE (if already exists, return error)
Open	Open an existing DASE data file. (if it does not exist or is not a DASE data file, return error)
Insert	Insert a data set into the DASE data file. Return errors if unable to insert, otherwise, insert the data set into the DASE data file and return an internal address of the first block.

Read	Read a data set from the DASE data file. If not a legal location, return an error, otherwise, return a pointer to the uncompressed data set.
Update	Locate the indicated data set, and re-write the information. If not a legal location, return an error.
Close	Close the DASE data set.
Destroy	Remove data set from DASE.

5.2 Disk File of a Data Structure Allocation Engine

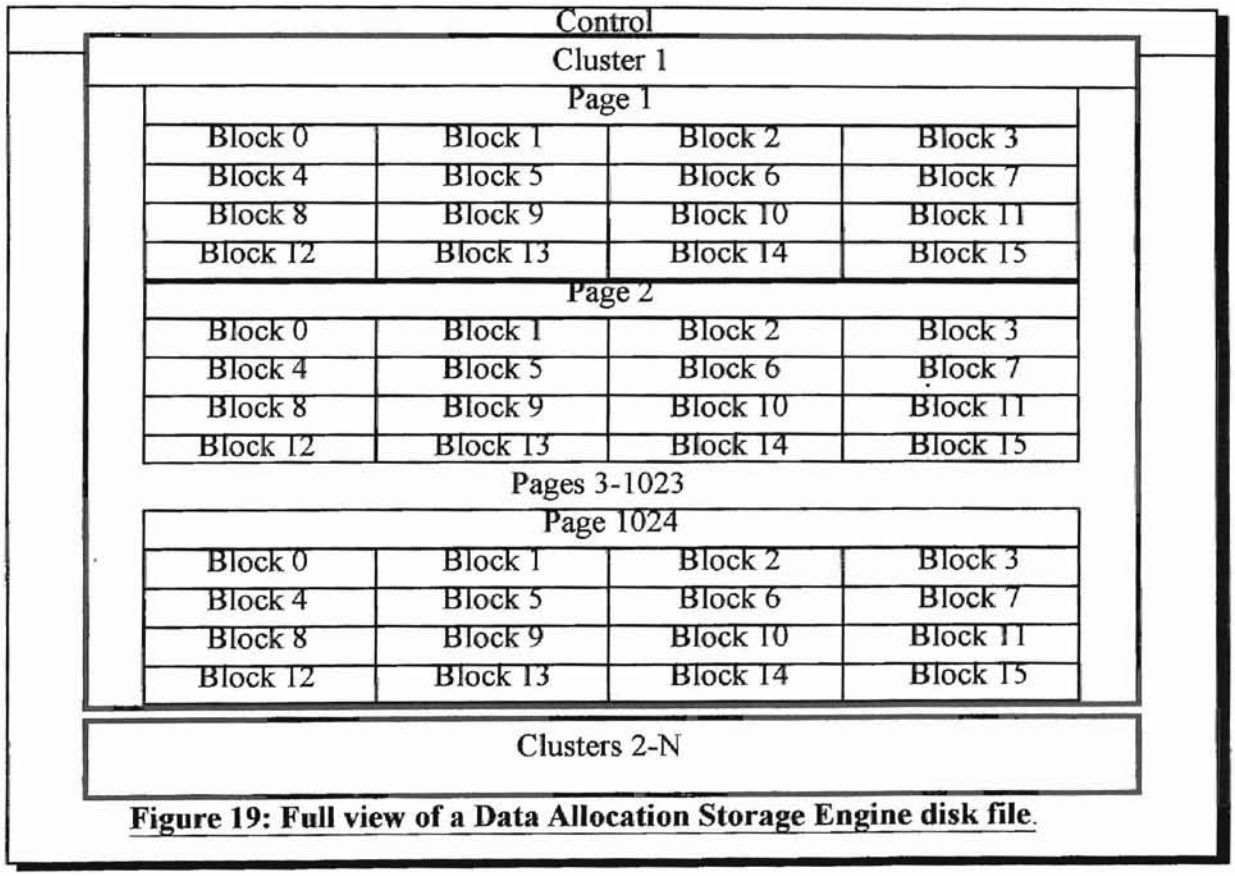


Figure 19: Full view of a Data Allocation Storage Engine disk file.

In Figure 19 above, we see the logical/physical layout of a DASE data file. Here we can see that Clusters are both a logic set of pages, and a physical set of data. The Cluster

data set itself contains several bit flags to track the free & used blocks in pages that belong to the logical pages of a cluster. The next 1024 pages belong to the logical cluster set. In other words, each logical cluster contains a cluster head data set, followed by a set of pages.

Proceeding all Clusters is a DASE control record (data set). There is only one control record in any physical DASA disk file. The control record stores the number of clusters and the number of pages in the last cluster. The control record also stores a unique character sequence to indicate that the data set is a DASE and what version it is. This will aid Open call to determine if the file that is being open, is indeed a DASE data file.

5.3 Storing Data in the Data Allocation Storage Engine

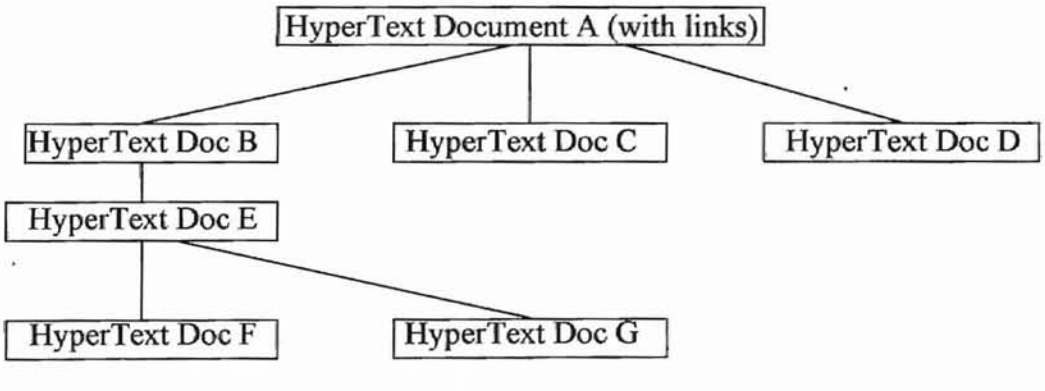


Figure 20: Hierarchical HyperText Documents

Consider the above set of HyperText Documents interconnected together. Let us assume that each document is variable length. Let us therefore give each document the following lengths after data compression is completed:

Document Name	Length	Blocks
HyperText Doc A	100 bytes	1
HyperText Doc B	300 bytes	3
HyperText Doc C	800 bytes	7
HyperText Doc D	525 bytes	5
HyperText Doc E	303 bytes	3
HyperText Doc F	200 bytes	2
HyperText Doc G	50 bytes	1

If we look at a DASE data file that has stored only these documents, it might look as follows: Let HyperText Doc A be represented by H_A, HyperText Doc B be represented by H_B, etc.

In the next few pages, we will show how this data is stored in a DASE disk file. Each set of data (document) is first broken into a set of blocks. Each block consists of 120 bytes of data plus 8 bytes of overhead (for illustration purposes only). In the first example, we will show the stored blocks without showing how they are linked together. In the second example, we show how the blocks and data are linked together. A hyperlink is a link that is not controlled by the DASE, but is a logical linked set up by the calling process. All "Internal Block Links" are fully controlled by the DASE. All blocks that indicate free are unused blocks that are available for future use.

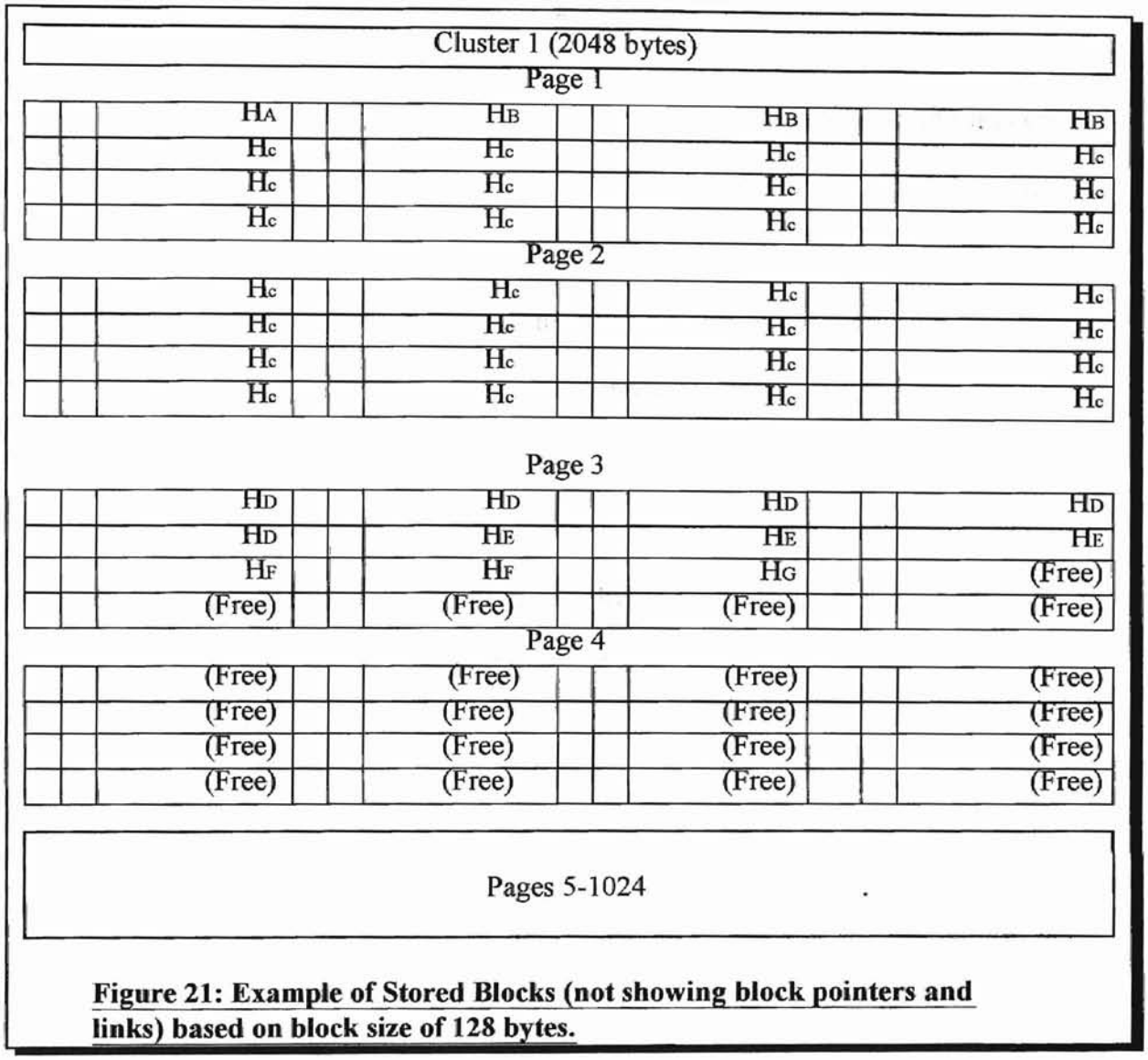


Figure 21: Example of Stored Blocks (not showing block pointers and links) based on block size of 128 bytes.

6. Conclusions and Future Work

This has been an enjoyable project in which I have learned so much. In the process of building the DASE, several issues arose. One of these issues was how big to make the blocks and how many blocks per page. In the end, I decided to make the software configurable. I did this by building static methods and variables into key classes. This required the software to call these static methods before any objects were instantiated.

There are several next steps I intend to implement using the concepts and software developed for this project. It is my hope and dream to use this software to develop an Intelligent database capable of storing any type of data format, be that object, hypermedia, knowledge, or any other data format.

It is my hope that the Data Allocation Structure Engine will be useful to other developers attempting to create a single file system for multiple systems. Those systems may be word processing, Databases, spread sheets and much more. These applications can run on single machines, networks, internet or intranets. The possibilities are endless.

With a strong foundation such as Data structure allocation engines, software packages can have multiple uses but have direct access to each others data.

Future enhancements to the data structure allocation engine will include security checks, with logins. Partitioning of the data file for greater flexibility. Other features may include automatic backup processing.

BIBLIOGRAPHY

[CHO89] Chorafas, D., (1989). Handbook of Database Management and Distributed Relational Databases. Blue Ridge Summit, PA.

[DIE89] Diehr, G., (1989). Database Management. University of Washington. Scott, Foreman and Company. Glenview, Illinois.

[FOL92] Folk, M., and Zoellick, B., (1992). File Structures, Second Edition. Addison-Wesley Publishing Company, Inc.

[GAR89] Gardarin, G., Valduriez, P., (1989). Relational Databases and Knowledge Bases. Addison-Wesley Publishing Company, Inc.

[KEM94] Kemper, A., Moerkotte, G., (1994). Object-Oriented Database Management. Prentice Hall, Englewood Cliffs, New Jersey 07632

[KIM92] Kim, Won, (1989). Introduction to Object-Oriented Databases, The MIT Press.

[KIM90] Kim, Won, Garza, Jorge F., Ballou, Nathaniel., and Woelk Darrell, (1990). Architecture of the ORION Next-Generation Database System. IEEE Trans. Knowledge and Data Engineering, Vol.2, No.1, Mar 1990, pp. 102-124.

[KRO83] Kroenke, D., (1983). Database Processing, Second Edition. Science Research Associates, Inc. A Subsidiary of IBM.

[NUT92] Nutt, Gary, j., (1992). Centralized and Distributed Operating Systems. Prentice Hall, Englewood Cliffs, ;New Jersey 07632.

[MIL87] Milenkovic, M., (1987). Operating Systems, Concepts and Design. McGraw-Hill Publishing Company.

[MUL89] Mullender, Sape., (1989). Distributed Systems, ACM Press, New York, New York.

[PAR93] Parsay, K., Chignell, M., (1993). Intelligent Database Tools and Applications, Hyperinformation Access, Data Quality, Visualization, Automatic Discovery. John Wiley and Sons Inc.

[PAR89] Parsay, K., Chignell, M., Khoshafian, S., Wong, H., (1989). Intelligent Databases, Object-Oriented, Deductive, Hypermedia Technologies. John Wiley & Son, Inc.

[SAR86] Sarnak, Neil., Tarjan, Robert, E., (1986) Planar Point Location using persistent search trees. Communications of the ACM July 1986, Vol 29, No. 7.

[SIL95] Silberschatz, A., Galvin, P., (1994). Operating System Concepts, Fourth Edition. Addision-Wesley Publishing Company.

[TAN92] Tanenbaum, Andrew S., (1992). Modern Operating Systems. Prentice Hall, N.J.

[TUR90] Turban, Efraim., (1990)., Decision Support and Expert Systems: Management Support Systems. Macmillan Publishing Company.

VITA

Ray Harvick

Candidate for the Degree of

Master of Science

Thesis: Time Persistent Data Allocation Storage Engine

Major Field: Computer Science

Biographical:

Personal Data: Born Fresno, California, On December 10, 1956, the son of Joe and Betty Harvick.

Education: Graduate from Silver Creek High School, San Jose, California in May 1983; received Bachelor of Science degree in Technical Education from Oklahoma State University, Stillwater, Oklahoma in December 1996. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in December 1997.

Experience: Raised in San Jose, California. Joined the U.S. Coast Guard in January 1976. Left the Coast Guard in January 1980 and started working as a computer programmer. Continued to work as a computer programmer until January 1993 where I enrolled at Oklahoma State University as an undergraduate student in Technical Education.