

Hashed Files:
Internals, Structure and Performance

by

Jeff A. Fitzgerald and Dr. Peggy A. Long

Published in 1991 uniVerse Symposium Proceedings
October 1991

Hashed Files: Internals, Structure and Performance

by

Jeff A. Fitzgerald and Peggy A. Long, Ph.D.

INTRODUCTION

Computers using the Pick Operating System and its relatives, Prime INFORMATION and uniVerse, are noted for the power and flexibility which they offer. Much of the power of Pick is in the realm of database management. Pick is not the first choice for high-precision scientific calculating, but when it comes to manipulating large quantities of business data it is unsurpassed.

We believe that the success of Pick in database management is due, at least primarily, to the methodology used to store and retrieve data. Pick utilizes a file structure known as a "hashed" file. As is typically the case, hashed files have associated costs and benefits. Most of the costs are paid by the operating system, a situation that we usually take for granted. When the costs sometimes levied by hashed files become extreme we may become aware of it as a result of user comments about system performance.

Taking care of hashed files is the single most effective method of improving and maintaining high system performance. We all want our systems to be fast -- this reason is sufficient to suggest that we can benefit by learning the structure of hashed files. There are other reasons for acquiring this knowledge, however. One is that a knowledge of file structure will aid us in software design. Most analysts design software from a functional perspective, paying little attention to the file access efficiency of their design. Armed with a knowledge of how the database is physically stored on disk, it is possible to realize tremendous gains in data retrieval speed and avoid potentially disastrous problems later on.

A knowledge of hashed file structure will also serve us in diagnosing, correcting and preventing system problems. Any of us who have had the experience of suffering a GFE (Group Format Error) can appreciate how beneficial an understanding of file internals can be. This understanding will also assist us in making system management decisions.

In this article, we will explain the hashed file structure used by uniVerse. In understanding the "how and why" of hashed files you will be able to understand how data are stored, how data are accessed and how a file should be configured to insure optimum performance.

AN ELEGANT DESIGN - HASHED FILES

The use of a database management system such as uniVerse generally implies that there is a lot of data to be stored. It also implies that a user may frequently choose to randomly access data records or to select a subset of the data. A database manager must provide a great deal of flexibility in data retrieval capabilities. Implementation of a retrieval system for database software is dependent upon the sophistication of the data file structure. Let's look at three common file structures and consider how data retrieval could be implemented.

Probably the most common file structure is one which we call a SAM file. This is an abbreviation for Sequential Access Method. Picture this file as one continuous string, as though the data were written on a roll of bathroom tissue. Each little "sheet" represents one physical disk unit of storage which is allocated to the file. To find a specific data record, the retrieval process must start at the beginning of the file (or "roll") reading each "sheet" until the desired record is found. If this data record is near the end of the "roll", it may take more time than we care to wait while the search is performed. This is not an ideal structure for a database but it is a very space efficient way to store data. There is no wasted space.

Because of the lengthy search time in reading a database implemented as a SAM file, the concept of indexing the data became desirable. Functionally this meant that the location of each data record would be available in a table. To find a specific data record, the retrieval process looked up the address in the table then accessed that address, reading the data record. This technique provides much faster access than the SAM structure. However, the cost of this method is in the disk space required and the processing overhead to maintain the index table. The table needs to be stored in some order so that it may be quickly searched. Some implementations store the data records in the same order as the table and allocate the same amount of space for each data record.

The concept of hashed files incorporates the best features of both of these file structures. It is a compromise to provide faster access to specific data records and to reduce the disk space usage and the size of the data location tables. It combines more efficient disk utilization than the indexed file and faster access to data records than the sequential file - a very elegant design.

The hashed file design uses the concept of subdividing the data into groups and indexing the group instead of the individual data record. This greatly reduces the space requirements for the location table. Each group is actually a small sequential file. If the length of the group is kept short, the search time remains satisfactory.

HASHED FILE STRUCTURE

When a file is created in uniVerse three parameters (in addition to the name of the file) may be passed to the CREATE-FILE verb: (1) the file TYPE, (2) the file MODULO, and (3) the file SEPARATION. For example, the following command will create a file named "TEST" with a type of 18, modulo of 101 and separation of 4:

CREATE-FILE TEST 18 101 4

We will discuss these three parameters (TYPE, MODULO and SEPARATION) individually, beginning with MODULO.

MODULO - A hashed file is divided into a number of "groups". Each group is a storage location where data may be placed. By dividing the file into groups, we can achieve faster data access by limiting the portion of the file which must be searched in order to retrieve a given data item. By analogy, consider a three-drawer filing cabinet with drawers labeled "A through I", "J through Q", and "R through Z" respectively. As we approach the cabinet, file folder in hand, we glance at the file folder label (e.g. "FITZGERALD & LONG") and match it to the appropriate drawer (the first drawer, because "F" is in the range "A through I"). We then place the folder in the appropriate drawer; we can completely ignore the other two file drawers. Retrieval of a folder works in a similar way, allowing us to find a folder much more quickly than if we had to search the entire file cabinet.

The modulo which is chosen for the file determines how many groups are provided for data storage. Each group is allocated a fixed amount of disk space to hold data. We will call the initial disk space assigned to a group a "data buffer" -- the exact size of a data buffer depends upon the "separation".

Suppose that we had chosen a modulo of 3 for the TEST file we just created. Further suppose that the separation chosen for the file was 1, which will result in data buffers of 512 bytes. The file is created with three groups, each of which can contain 512 bytes of data. The groups use disk space ("frames" on native Pick) which is logically contiguous. The address of each group can be calculated as an offset from the beginning of the file. The file can be represented by the drawing in Figure 1.

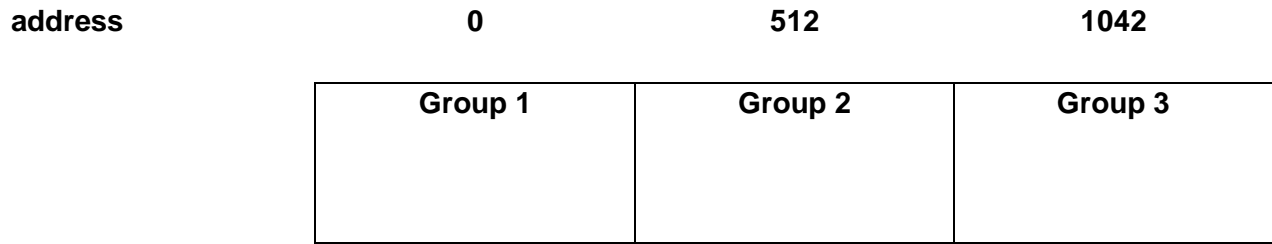


Figure 1 - A hashed file with modulo 3

As data items are written to the file a "hashing algorithm" determines into which of the three groups the items are placed. The hashing algorithm is the computer equivalent of looking at the file folder label to see which drawer of the filing cabinet the folder should go into.

OVERFLOW - Since each group can contain only 512 bytes of data, it is very easy to fill the primary data buffer of a group. For example, suppose that six records of one-hundred bytes each hash into group 1. We need at least 600 bytes of space and the data buffer used by group 1 can only hold 512 bytes (actually even less since there is internal overhead in the group as well). We all know that the data goes somewhere, since our users never get messages saying, "Sorry, but your file's full!". The additional data is placed into an "overflow buffer" which is appended to the group. In our example, the file will now look like Figure 2.

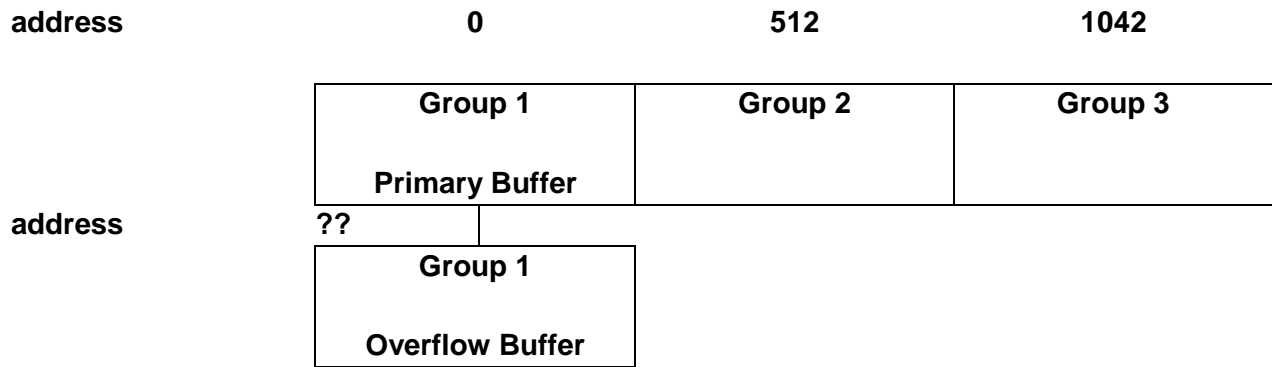


Figure 2 - A hashed file in overflow

Our file is now in "overflow" -- group 1 consists of two 512 byte data buffers. We are calling one of the data buffers the "primary" buffer and the other the "overflow" buffer. The address of the overflow buffer is assigned at the time the disk space is allocated and will vary

seldom be contiguous or close to the primary data buffer. As more data is added to the group it will append as many overflow buffers to the group as are needed to store the data. We have seen groups with literally hundreds of overflow buffers.

Overflow is one of the costs of the hashed file structure. The cost of overflow is measured in terms of the extra work required to access data and the performance degradation resulting from this extra work.

In our file cabinet, folders are alphabetized within each drawer so that locating a folder is easy. Hashed files are not as "intelligent" however; the data within a group is not sorted. When a new item is placed into a group it simply goes to the end of that group. When a data item is retrieved from a group, the group is searched sequentially from the beginning until the item is found or the end of the group is reached. The longer the group and the more items in the group the longer it will take to add or retrieve data in that group.

The real cost of overflow, however, is in the additional disk I/O required to read and write the overflow buffers. In our example it will require at least two reads and two writes to update a data item in the overflow buffer for group 1. If more overflow buffers are present in the group, the overhead required to access data will increase dramatically.

RESIZING - The modulo chosen at the time of creation of a file is a compromise. If the modulo is too large for the data, disk space is wasted and performance degrades for processes which must read the entire file, like query language reports. But because new data is being added to the file and existing items are being updated, a modulo which was once appropriate may not be large enough to last for very long without overflow being added to the file. The solution is one of the costs of the hashed file structure: **In order to maintain acceptable performance, hashed files must be restructured systematically and frequently to new modulos which will reduce the amount of overflow in the files.**

The proper modulo may be calculated by counting the number of bytes of data and dividing by the average number of bytes we wish to place in a group. In addition, the modulo should always be a prime number other than 2 or 5 (a prime number is one that can only be divided by itself and 1). As an example, suppose that our TEST file, which is a modulo 3, has had lots of data added to it. The groups have gone into overflow, so that the file looks somewhat like Figure 3.

Group one of the file has three overflow buffers, group two has two overflow buffers and group three has four overflow buffers. Reading the extended groups will require more disk I/O and more CPU time. In addition, the groups are more vulnerable to damage (group format errors) and group lock problems.

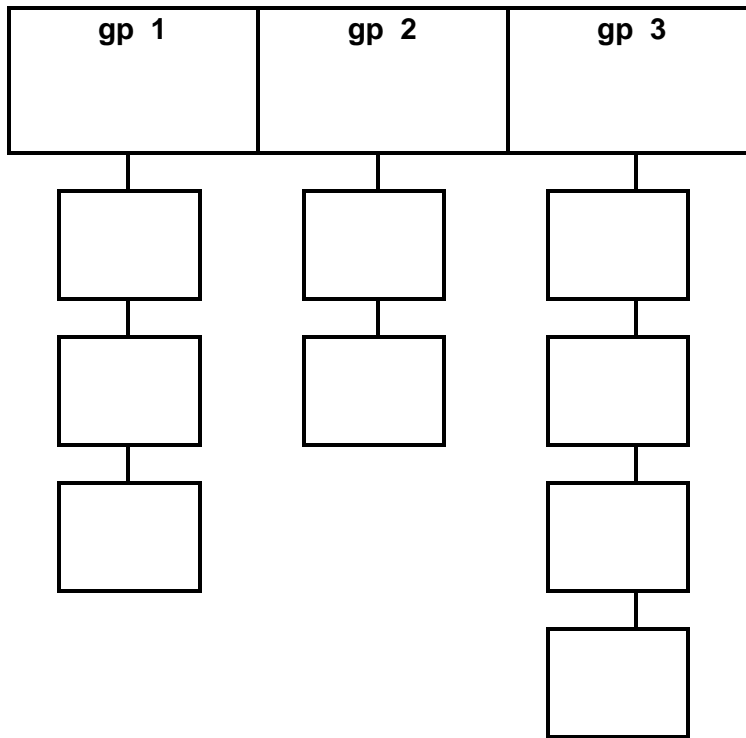


Figure 3 - A file with heavy overflow

Suppose that the file contains 5,400 bytes of data. Our calculation for the new modulo is $5400 / 400$ which yields 13.5. We are dividing by 400 because we would like to place an average of 400 bytes in each group, filling a 512 byte data buffer about 80% full. We next round 13.5 up to the next higher prime number which is 17. To change the file to the new modulo we use the uniVerse RESIZE command. The new file would look like Figure 4.

The file will perform much more efficiently now because every data item in the file can be accessed with a single disk I/O. Of course, as more data is added to the file it may go back into an overflow condition. Because data is added, removed and updated very frequently, file maintenance is one of the on-going responsibilities of the system administrator.

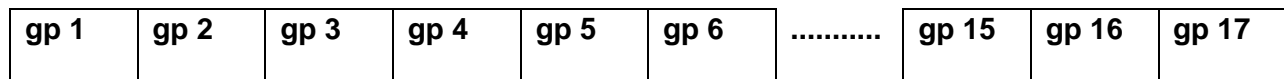


Figure 4 - A file with no overflow, modulo 17

SEPARATION -The separation determines how many multiples of 512 bytes to allocate for each data buffer or overflow buffer. For example, a separation of 2 would yield 1,024 byte buffers; 4 generates 2,048 byte buffers; 8 defines 4,096 byte buffers.

Deciding on the best separation for a particular file can be very complex because there are a number of issues which should be considered. We are aware of no formula which has been experimentally proven to produce the best solution. The issues one should keep in mind while choosing the separation for a file are:

1. Average size of data records
2. Number of large records
3. Physical size of disk transfer block
4. Disk space utilization
5. Unix system calls
6. UniVerse sequential search time

This is quite a list to keep in mind when choosing the separation. Although there are exceptions, a very strong case can be made for using a separation of four for most files. In the following paragraphs we will present our logic for selecting a separation of four.

On most older Pick machines, the physical disk block size was 512 bytes. It made sense to choose a logical buffer size which matched the physical block size. On Unix systems, the block size varies depending upon the way the disk drive was formatted and the parameters used when the filesystem was created. Unix systems we have seen vary greatly and include a 512 byte block, 1k, 2k, 4k, 8k and 16k. Making separation match the Unix configuration may have some merit, but being certain that one knows what the configuration is not always practical. On some Unix systems it is possible to configure each filesystem with a different block size.

VMARK published a technical note stating that separations should be powers of two, not to exceed 16 for best performance. What does this mean? It means that the only separations we should consider are 1,2,4,8, and 16. This narrows our choices considerably! Why only powers of two? These separations will insure that the Unix physical block boundaries and the uniVerse data buffer boundaries are compatible. Consider a separation of three. This would create buffers of around 1500 bytes. If the Unix buffer is 2k, the data buffers would not line up with the physical buffer boundaries - this would cause two Unix disk reads to take place in order to read one uniVerse group. It's much more efficient when the Unix disk blocks and the uniVerse groups do not overlap.

Let's consider using a separation of 1 (512 bytes). It is logical to conclude that the smaller the data buffer the shorter the sequential search for data in that buffer. This is true but there are other considerations. If the data records are very small (under 50 bytes) then this may be a good choice. However, many uniVerse data files have records which average around 300 bytes. UniVerse will not break a record across a buffer boundary unless the record is larger than the buffer. For example, if we store a 300 byte record in a 512 byte buffer there will be approximately 212 bytes of free space available. For the sake of simplicity we will ignore the record headers. The only way uniVerse can use this 212 bytes of free space is in storing a record which hashes to that group and is less than 212 bytes. If the next record to be stored in that group is 225 bytes, uniVerse will append an overflow buffer and store the new record in the new buffer. The 212 bytes in the primary buffer will be flagged as available but will never be used unless a small data record is added to the group. In this scenario the wasted space in the file will be approximately 40%. This means that to avoid overflow, the file will need to be 40% larger than necessary. It would take 6 buffers of 512 each to store 6 data records or 3072 bytes of disk space. By simply changing the separation to four, 2048 bytes, we can store those 6 records in one buffer ($6 \times 300 = 1800$ bytes which will fit into a 2048 byte buffer).

But, what about sequential search time? Doesn't it take longer to search 2048 bytes than it does to search 512? It probably does. However, there is another issue which we should consider. When uniVerse needs to read data, Unix performs the task. Part of this task is a "system call". Each time data are passed to uniVerse they are passed in chunks which are the size of the buffer defined by the separation. This means that our example of a file with a separation of 1 containing 6 data records would require at least 6 system calls to read all of the data. With a separation of 4, the same data could be accessed with one system call. By increasing the size of the buffer (increasing the separation), we can reduce system calls and thereby save system overhead. The end result is faster access to the data.

Does this mean that we could achieve even faster access if we made the separation even larger? Probably not! In our experimentation, we saw a decrease in access time for individual records when the separation was 8 and larger. Our belief is that the time saved in reducing system calls is not as significant as the time it takes to read a larger block of data from disk and for the sequential search through the larger buffer.

What about large records? It is not unusual for an analyst to increase the separation to get rid of overflow caused by large records. Let's say that in a particular file with about 1000 records, most of the records are around 300 bytes, but there are 10 records that have more than 3000 bytes each. UniVerse has an elegant way of storing large records, records larger than a buffer. In a separation 4 file, there would be at least 10 buffers of overflow because of these large records. UniVerse places the last 2000 bytes of each large record in an overflow buffer and the first 1000 bytes in the primary group buffer. This is a great way to get the "lumps" out of the data. The overflow buffers which contain the large records are read ONLY when the large records are needed. Otherwise, this overflow from the large records is not read. This saves time. If we change the separation to 8, a 4k buffer to get rid of the overflow from the large records, we cause uniVerse to have to read and search through 4000 bytes for each record, not just the large ones. The larger buffer would increase search time except for the 10 large records!

Our conclusions are that a separation 4 is generally the best solution for most files. Using a separation larger than one or two usually saves disk space and system calls. Separations larger than four frequently increase sequential search time which degrades performance. We would consider using only separations which are a power of two to avoid any mismatch between the logical and physical disk buffer configuration.

TYPE - Hashing is probably the most obscure, sparsely documented and least understood aspect of uniVerse database files. Although hashing is the basis for the file system designs for uniVerse, PICK and INFORMATION files, you may not even associate the word "hashing" with these software systems. It is certainly not prominent in the documentation. INFORMATION and uniVerse users are more apt to be familiar with the term "Type" which is the label used for the parameter which specifies the hashing algorithm. Choosing the "Type" is jargon for selecting a hashing algorithm. Because PICK systems only support one "Type" (one hashing algorithm), there is minimal awareness of the concept of hashing among PICK users.

Why belabor the point so here? Because this file parameter, Type, can make the difference between a file which performs efficiently and a file which requires excessive I/O operations to retrieve a data record. The hashing algorithm selected for the file may be inappropriate for the set of keys and data in a particular file.

PICK users may be unaware that there are 17 hashing algorithms from which to choose. Although you may be using "PICK flavor" accounts, you may choose from all of the available Types, 2 through 18. In choosing the most effective Type for each data file, we can expect to improve system throughput time in our uniVerse database system.

In a paper filing system, the quantity of data to be stored is usually subdivided into the drawers of a filing cabinet. These drawers are the logical equivalent of the three groups in our modulo 3 file. Let's imagine that we have a filing cabinet which consists of three drawers. Each drawer is large enough to hold 2048 sheets (bytes) or Separation 4. Each drawer is labeled with a range of alphabetical characters which define the range of beginning characters on the labels (record keys) in our file drawers. This is a very verbose explanation to describe the process of placing the document which is labeled "FITZGERALD" into the drawer which is designated as the "A - J" drawer.

Alphabetization is such a commonly known and accepted hashing algorithm that it needs no analysis or explanation of how it works. We have all used it many times. If you have ever worked in an office which used filing cabinets you have probably encountered the problem with the drawer which contained the "S" files and the "T" files frequently becoming overloaded before any of the other drawers. When a drawer becomes full, it's necessary to reallocate the filing space by redistributing the documents over the domain of file drawers. This process requires that the documents be "re-hashed" into new locations. Because certain letters occur more frequently in our language than others, the distribution of the number of documents per alphabetic character is not even. This means that more words in the English language begin with "S" than with "Z". Thus, the need to redistribute the documents in the filing system occurs as more documents are added to the system.

Hashing is simply the process of subdividing the data into groups such as drawers in a file cabinet. The hashing algorithms or formulae used by the uniVerse file system use the Key or ID of each data record to determine where to place that data record as well as where to locate it when retrieval is requested. Although the actual algorithms are mathematically complex we can simplify the explanation of how hashing works through a demonstration of a hashing algorithm we designed which functionally mimics the processes performed by the "real" hashing algorithms.

The first step in a hashing algorithm is to convert the key into a "hashed value". This means we want to convert the key into some integer number which we may then divide by the modulo. For example, if we were to use the rule of adding the digits of the ID to derive the "hashed value" then the ID '123' would sum to 6 ($1 + 2 + 3 = 6$). Although the "real" hashing algorithms are much more complex than this, you can see from this example that we are converting the ID into some other number by way of a mathematical formula or rule.

The second step is to divide the "hashed value" (6) by the modulo. Why? Because, we want the result of the division to produce a remainder. Adding one to the remainder defines the group number where this particular data record will be stored.

Think about how division works. When we divide, the result is expressed as a quotient and a remainder. The remainder will always be between 0 and the denominator minus 1. Any number divided by three will result in a remainder of 0, 1, or 2. For example, 6 divided by 3 is 2 with a remainder of 0. Seven divided by 3 is 2 with a remainder of 1. Eight divided by 3 is 2 with a remainder of 2. Nine divided by 3 is 3 with a remainder of 0. There are only three possible remainders: 0, 1, and 2. We have defined our file as a modulo 3. The denominator of the hashing algorithm is equal to the modulo. Therefore, a modulo of three defines three groups. There are only three possible remainders when the denominator is three. If we add one to each possible remainder then the resulting range of numbers is 1, 2, and 3. These three numbers define the range of groups in a modulo three file. This technique of creating an algorithm to convert the ID to a "hashed value" and then dividing by the modulo of the file is classically called "hashing by division". The point? "Hashing by division" is dependent upon the even distribution of remainders to provide good hashing throughout a datafile.

Let's examine a simplistic set of data and several hashing algorithms which we have invented for demonstration purposes. Using the hashing algorithm of summing the digits of the keys let's look at how the following set of keys would hash.

KEY	Sum	Reminder	Group No.
123	6	0	1
124	7	1	2
125	8	2	3
126	9	0	1
127	10	1	2
128	11	2	3
129	12	0	1
130	4	1	2

Let's step through our hashing algorithm. Note the first two lines of the preceding matrix as you read the remainder of this paragraph. Add the digits of the key "123". The sum is "6". Now divide by 3, the modulo. The remainder is 0. Add one to the remainder. The result is "1". This is the number of the group where the data record with the ID of "123" will be stored. Both the ID and the data are stored together. The next key is "124". Add the digits. The sum is 7. Divide by the modulo 3. The remainder is 1. Add one to the remainder. The group number where the ID and record will be stored is "2". Note that we are using sequential numeric IDs (123 - 130). Note that the data are evenly distributed among the three groups (see the Group No. column). In this example the data will be evenly distributed as long as the IDs are sequential and numeric.

Most files do not use sequential numeric IDs, because there is value in assigning IDs which have meaning. Frequently a pattern of some sort is used in the IDs of a typical database. There is good reason for this. Users want to organize the data in a meaningful way with IDs which are also meaningful. Knowing that a part number for "pipe" begins with "P" assists the user in remembering part numbers and also in checking for inappropriate entries in purchase orders or invoices as well as serving other accounting needs.

Let's imagine that we have an accounting file with accounting codes which are sequenced like this:

123 126 129 132 135

Note that we are using every third number. Using the same hashing algorithm as in the previous example, let's examine how this set of keys will hash. Remember the rule: add the digits of the IDs, divide by the modulo and add one to the remainder.

KEY	SUM	REMAINDER	GROUP No.
123	6	0	1
126	9	0	1
129	12	0	1
132	6	0	1
135	9	0	1

This is an extreme case where hashing is exceptionally poor. Every record will hash into GROUP 1. There will be no data in groups 2 and 3. This is the equivalent of having all of our file folders in the same drawer and not using the rest of the cabinet.

Another type might work better to evenly distribute the data throughout the three groups defined by the modulo. Let's modify our formula (create another hashing algorithm) to see if we can improve the distribution of data such that all of the groups are used.

Our second hashing algorithm adds the digits in the IDs plus it adds 1 for every odd digit in the key.

KEY	SUM	REMAINDER	GROUP NO.
123	6+2=8	2	3
126	9+1=10	1	2
129	12+2=14	2	3
132	6+2=8	2	3
135	9+3=12	0	1

Does this offer a better distribution? It is a little better but not perfect. There are 3 records in group 3, 1 in groups 1 and 2. As you can imagine, this is the type of process which led to multiple hashing algorithms.

The uniVerse documentation defines Type 2 as: "The keys in this file are wholly numeric and significant in the right most eight characters." The Type 2 hashing algorithm is constructed such that it favors all numeric RANDOM keys which are most unique in the last eight characters. Does this mean that it will work well for any file which contains all numeric keys? Probably (we haven't checked ALL possibilities) but the operative word is "RANDOM". Hashing is dependent upon the remainder (dividing by the modulo) being evenly distributed.

Look back at our first example of hashing. We used sequential, numeric IDs and the hashing (distribution) was perfect. In our second example, we used IDs which used every third number. Both of these sets of keys fit the description "wholly numeric and significant in the last eight characters". But there is a very large difference in the way the two sets of IDs hash. We are not suggesting that our hashing algorithms are equivalent to the Type 2 hashing algorithm. They are not. What we are demonstrating with our examples is that a hashing algorithm may distribute the data beautifully for one file and very poorly in another file even when it appears as though the set of IDs in both files are similar enough to hash the same. Because we very seldom use random keys but instead we use IDs which have meaning (which generally implies a pattern) the IDs cannot be random and therefore we cannot depend upon the descriptions of the keys to guarantee us that the hashing will be the best. How then can we choose the best Type for each file? Experimentation. There are two utilities in uniVerse HASH-TEST and HASH.AID, which allow us to play "what if" the file were changed to a different type, for example a Type 3, Modulo 7. HASH.TEST can provide this answer for us. However, we prefer checking ALL possible types using HASH.AID to find the best solution.

To check all possible types for a particular file we decide upon the size of the modulo. We first calculate how much space we need to allocate for the file. Here's an example. Let's say our file has 10,000 data bytes. We divide that number by this formula:

$$\text{Separation} * 512 * .8$$

For this example we have chosen a separation of 4. This would yield a denominator of 1638.4.

$$\frac{10,000 \text{ bytes}}{1638.4} = 6.10$$

The ".8" (we think of it as 80%) is a way of calculating a modulo which is about 20% larger than is needed to hold the data. It's our "growth" space. We also choose the next higher prime number in order to avoid interactions between the type and modulo which can occur when non-prime numbers are used. In this example, our modulo would be 7.

The syntax we use to execute HASH.AID is:

`HASH.AID filename modulo 2,18,1 separation`

filename - the name of the file
modulo - the modulo we calculated (7 for our preceding example)
2,18,1 - this causes HASH.AID to loop from Type 2 through 18, incrementing the type by 1

HASH.AID produces a report which may be sent to the printer with the LPTR command. However, we prefer to use the data which HASH.AID has written into the HASH.AID.FILE. How do you get it??

Here is a sample report from a file which appears to be very poorly hashed.

HASH.AID summary report								Page	1
Type	Modulo	Separation	Records	File Size	Smallest Group	Largest Group	Average Size	Oversize Groups	
2	20	4	501	110,592	0	40,676	3,536	2	
3	20	4	501	112,640	0	41,000	3,536	2	
4	20	4	501	106,496	0	14,908	3,536	5	
5	20	4	501	108,544	0	16,528	3,536	5	
6	20	4	501	110,592	0	40,676	3,536	2	
7	20	4	501	112,640	0	41,000	3,536	2	
8	20	4	501	94,208	1,456	6,064	3,536	18	
9	20	4	501	88,064	1,852	6,608	3,536	17	
10	20	4	501	110,592	0	40,676	3,536	2	
11	20	4	501	110,592	0	40,676	3,536	2	
12	20	4	501	108,544	0	15,432	3,536	5	
13	20	4	501	108,544	0	15,288	3,536	5	
14	20	4	501	110,592	0	40,676	3,536	2	
15	20	4	501	110,592	0	40,676	3,536	2	
16	20	4	501	102,400	0	11,072	3,536	10	
17	20	4	501	92,160	2,380	5,572	3,536	20	
18	20	4	501	110,592	0	40,676	3,536	2	

Type - This is the first column on this report. Note that it varies from 2 to 18. This is the range of Types we requested when we executed HASH.AID.

Modulo - This is the modulo which was used for the modeling process. Note that we used only the modulo 20. In actual production, we would not have used 20. It is not a prime number and there is a particularly strong interaction with most of the hashing algorithms as you can see in this example.

Separation - We have also held the separation constant at 4. Because separation does not affect hashing there is no need to vary it when we are investigating hashing.

Records - This number reports the number of data records in the file. Obviously, all records were in the file for each iteration of HASH.AID.

File Size - This is the number of physical bytes in the file. It varies from type to type because of the number of groups with overflow and the number of pages of overflow. Divide any of these numbers by 2048 (separation * 512) and you will find the number of "data buffers" defined for the file.

Smallest Group - This is the number of bytes in the smallest group. Remember, the number of groups is equal to the modulo. If a group has more than 512 bytes times the separation (2048 in this example) then the group is in overflow. Note that for Type 17, the smallest group is 2,380 bytes and must be in overflow because it exceeds 2,048 bytes. Note also that most of the groups are empty (zero in this column).

Largest Group - This is the number of bytes in the largest group. All of the numbers in this column exceed 2,048 bytes which means that all of these "largest groups" are in overflow. We can calculate the extent of the overflow by dividing by 2,048. Types 3 and 7 produce the largest groups at 41,000 bytes. This represents an overflow chain of 20 data buffers ($41,000/2048 = 20.019$).

Average Size - This is the average number of bytes per group. It is calculated by dividing the total number of data bytes in the file by the modulo. If the data were perfectly distributed across all 20 groups, each group would contain 3,536 bytes. This average is above 2048 which means even with perfect distribution we would have overflow. Obviously the modulo is not large enough for the amount of data in the file.

Oversize Groups - This is the number of groups in the file which have overflow. This statistic can be misleading if the size of the largest group is not taken into account. Because overflow causes unnecessary disk I/O reading in the overflow pages, a file with a lot of overflow generally takes more time to process (typical processes are database inquiry and reporting). One could assume that the more groups in overflow than the worse the performance. This is true as long as the overflow depth is equal. In general, the longer the overflow chain, the poorer the performance of the file. Therefore it is better from a performance standpoint for a file to have 12 groups in overflow by one data buffer (separation * 512) than to have one group in overflow by 12 data buffers.

In this report, we would select Type 17 as the best type for this data file. Our technique is to look for the type where the smallest, largest, and average group sizes are closest in size. This is the case for Type 17. Note that the smallest is 2,380, the largest is 5,572 and the average is 3,536. When there is more than one type where the smallest, largest and average byte counts are very similar we then look at the number of groups in overflow (oversized) and choose the type where the number of groups in overflow is smallest.

In summary, we use HASH.AID to perform the modeling of type. We use the HASH.AID.FILE to choose the best type. Our layman's definition of best type is the type which produces the most even distribution of data bytes across the allocated groups. Simply stated, we want the same number of bytes to be placed into each group. Practically speaking this seldom happens because data records vary in size and a data record cannot be split between two groups. Therefore we choose the type which does the best job of approaching this goal.

Hashing is the mechanism which controls into which groups the data records will be stored. Classically, this technique used by Pick, Prime INFORMATION, and uniVerse is called "hashing by division". It's an elegant, efficient design for database structures.

PERFORMANCE CONSIDERATIONS

There are a number of scenarios related to file parameters which can cause performance degradation. The fundamental cause can be described as unnecessary file I/O. Specific to the file structure, this unnecessary file I/O occurs when the file is in overflow (the modulo is too small or the type is not distributing the data evenly) or the physical file is too large for the amount of data stored. Another contributor to overflow which is NOT a FILE MAINTENANCE PROBLEM, but a design problem is large records. A large record is one which is larger than the "data buffer" size used by the file (separation + 512 bytes). To illustrate how all of these issues may occur we offer the following analogy.

In this analogy, a common five drawer file cabinet represents our data file, Modulo 5. There are five drawers in our file, each drawer may be called a data buffer, a bucket or simply a drawer in this example. Each of the file folders contains data (each folder is a "data record"), and each has a unique label. We call this label the record Key.

The owner/user of this file selected a hashing algorithm based upon his intuition and experience when the file was placed in use and the data was stored. This algorithm, or formula for deciding in which drawer to place each data record is based upon the alphabet. Here is the hashing formula:

If the first character in the Key (label) is in the range of A through H then the record (folder) will be placed into the first drawer, group 1. If the first character in the Key is in the range I through L the record will be placed into the second drawer, group 2. M through P will be placed in group 3; Q through S into group 4 and T through Z will be placed into group 5.

This hashing algorithm was selected based upon the user's experience. He expected this file to have the same number of records and for the assortment of labels (keys) to be similar to those he had used in the past.

Each drawer has a fixed physical size. The maximum number of records each drawer will hold is a function of how many bytes of data are in each record. For the sake of our analogy, think of a sheet of paper as a byte. Let's pretend that this hypothetical file has drawers that will hold only 2,048 sheets of paper in each drawer. The total number of sheet that the entire file will hold is 10,240. But, due to the hashing algorithm we have selected, we may not be able to make that much data fit. There are several problems to be considered. First, what would be the outcome if there were 5 records with labels beginning with "S" and each record contained 1000 sheets (5 records * 1000 sheets (bytes) = 5000 bytes). Drawer 4 will hold only 2048 sheets. Additionally there are other records which hash to this drawer also causing the drawer to be well over 5000 bytes. Where will we place this "extra" data that we have no room to store in drawer 4? Until we have time to reorganize the file, we will use a box (which once held computer paper) and mark it as "continued from drawer 4." We will place a note in drawer 4 explaining where the box is located.

Before continuing our analogy, we need to explain that the records in each drawer are not ordered, this means they are not in alphabetical order or any order other than perhaps the order in which they were created. This also means that when we need to retrieve a record we must start at the front of the drawer and read each label until we find the record we want. We cannot assume that because the label begins with "SMITH" that it will follow the record with the label of "SAND".

Any record created after the drawer became full will be placed into the box. The box has the same capacity as the drawer. If we search the drawer and can't find a particular record then we must search the box. The generic name for the box is "overflow". This box may be used only for a continuation of group 4. Should another drawer need extra space, another box must be found. If both the drawer and the box for group 4 are full and another "S" record is created, a second box is added. The first box has a label that says there is another box containing data. The label also documents the location of the second box.

The file users have become quite irritable because they must search through one drawer and two boxes when they need a record from the fourth drawer. It's slow.

One of the users, the database administrator, noticed that there was only 500 sheets in drawer 3 and 10 sheets in drawer 5. By changing the hashing algorithm, the labels on the front of the drawers, he could divide the data differently and get all of the data, including the data in the two boxes into the 5 drawers, eliminating the need for the boxes. This scenario describes a file where the hashing algorithm needs to be changed because the records are not evenly distributed through the drawers. One drawer is in overflow by two boxes and two other drawers are less than 25% full.

This user proceeds to change the hashing algorithm to one where all of the records will fit in the drawers -- he changes the rules of what goes where and then he moves the records to their new locations. If this were a uniVerse file, we would describe this process as choosing a new type (hashing algorithm) and RESIZEing the file (moving the records). The program called RESIZE is used to actually change the type, modulo and/or separation of a file.

We could describe the problem of having enough drawers, the number of groups (modulo) for the records but not filling them evenly as "Poor Hashing". Poor hashing causes overflow. Overflow takes more time to search. Remember the overflow boxes.

Let's consider a second scenario where there are the same number of sheets in each drawer and each drawer also has a box of overflow. Five drawers just won't hold the data. The users aren't very happy with this file either. They have to search through a drawer and a box every time they need to find a record. Another five drawer cabinet is needed. By adding another cabinet, increasing the modulo to 10 (adding five more drawers) we will subdivide the data into ten groups and place the records in the respective drawers. We could describe the problem of simply not having enough space as "Modulo Too Small".

The third scenario occurred because the database administrator noticed that one file contained a few dozen records with more than 3000 bytes (sheets). He noticed that each of these drawers had two overflow boxes. Because 3000 sheets won't fit into a drawer which

holds 2048 sheets, a rule was established which stated that if the record was bigger than the drawer, then the record would be stored in the overflow boxes, as many as it took to hold the large records. The administrator, believing that the overflow boxes were inefficient discarded the drawers which held 2048 (separation 4) bytes and replaced the with drawers which held 4096 bytes (separation 8). Now the drawers are big enough to hold those large records and there is no overflow. However, because the drawers are twice as big as they once were, it takes more time to search through each drawer. In changing the size of the drawer to accommodate a few dozen large records, every search requires more time because the drawers are larger. Remember the search always starts at the front of the drawer.

The fourth and last scenario that we will describe is the case where there are 25 file drawers (modulo 25) used by one file. There are only 25 records, one in each drawer. The database administrator decided he was no longer willing to face the overflow problem with this file. It would be big enough. If a user only needed one record the search was very quick. However, if a user needed to read something from each record to produce a report, he spent a lot of time opening drawers and moving through the room. This took a lot of extra time and the users were not happy. This file had too much space, the modulo was too big.

The "moral" of our story is that the most efficient file is one which has little overflow, there are enough drawers to hold the data but its not so big that it takes extra time accessing more drawers than are needed. Additionally, the drawers are not too large so that the time to search an individual drawer is relatively small.

We have presented the concepts of hashed files and the parameters which define them, type, modulo, and separation. We have also illustrated that file access becomes less efficient if the file is not sized properly or if the hashing algorithm does not distribute the data evenly over the allocated space. We have presented these hashed file concepts because we believe that it may assist you in making better, more efficient file maintenance choices for type, modulo and separation.

PERFORMANCE AND OVERFLOW

Resizing files is very time consuming and therefore expensive. However, this is necessary to insure that the most efficient parameters are maintained as our uniVerse databases grow and change. Is it worth the effort? In an attempt to provide an answer we ran a series of tests designed to test file performance over a range of overflow conditions.

The testing was done on an IBM PS/2 running SCO's version of UNIX and VMark's uniVerse. We were the only user on the system at the time of the testing. We created a file with a modulo of 720 to use as a base. We designed the item IDs of our data items such that they hashed perfectly in spite of our using a non-prime number for the modulo, something we would never do otherwise. We filled the file with 6,480 data records containing 40 bytes of data each, including the item ID. The uniVerse environment adds a twelve byte record header to each record, therefore each record takes up 52 bytes of space on disk. Nine records hashed into each of the 720 groups of the file, causing each group to contain 468 bytes total.

We used a separation of 1 which allocated data buffers of 512 bytes to our file. The base file was without any overflow. We then ran a BASIC program which read each of the items in the file. The program accessed the items in a very random order. We timed the elapsed time to run the program.

The modulo of the file was then made smaller so that the same data caused each group to go into overflow by a calculated amount. We reran the program using the new file and timed the elapsed time. Then the modulo was further reduced. This process continued until the file was so undersized that each group of the file needed 73 overflow buffers to hold the data. The timings for each test file are shown in Figure 5.

As you can see, the elapsed time for the program to run increases dramatically as the amount of overflow in the file gets larger. From the best to worst cases the time went from 122 seconds to 475 seconds, a performance degradation of 289%!

The degradation due to overflow in the file can be seen even more clearly in the graph shown in Figure 6. This sort of demonstration clearly reveals that file maintenance is worth the time that it requires.

<u>Modulo</u>	<u>Overflow</u>	<u>Elapsed Time</u>
720	0	122 secs
360	1	124 secs
240	2	130 secs
180	3	133 secs
144	4	139 secs
120	5	145 secs
90	7	153 secs
80	8	162 secs
72	9	169 secs
60	10	174 secs
48	13	185 secs
45	14	202 secs
40	16	204 secs
36	18	215 secs
30	21	226 secs
24	27	255 secs
18	36	296 secs
9	73	475 secs

Figure 5 -- *File access times as a function of overflow.*

Access Time as a Function of Overflow

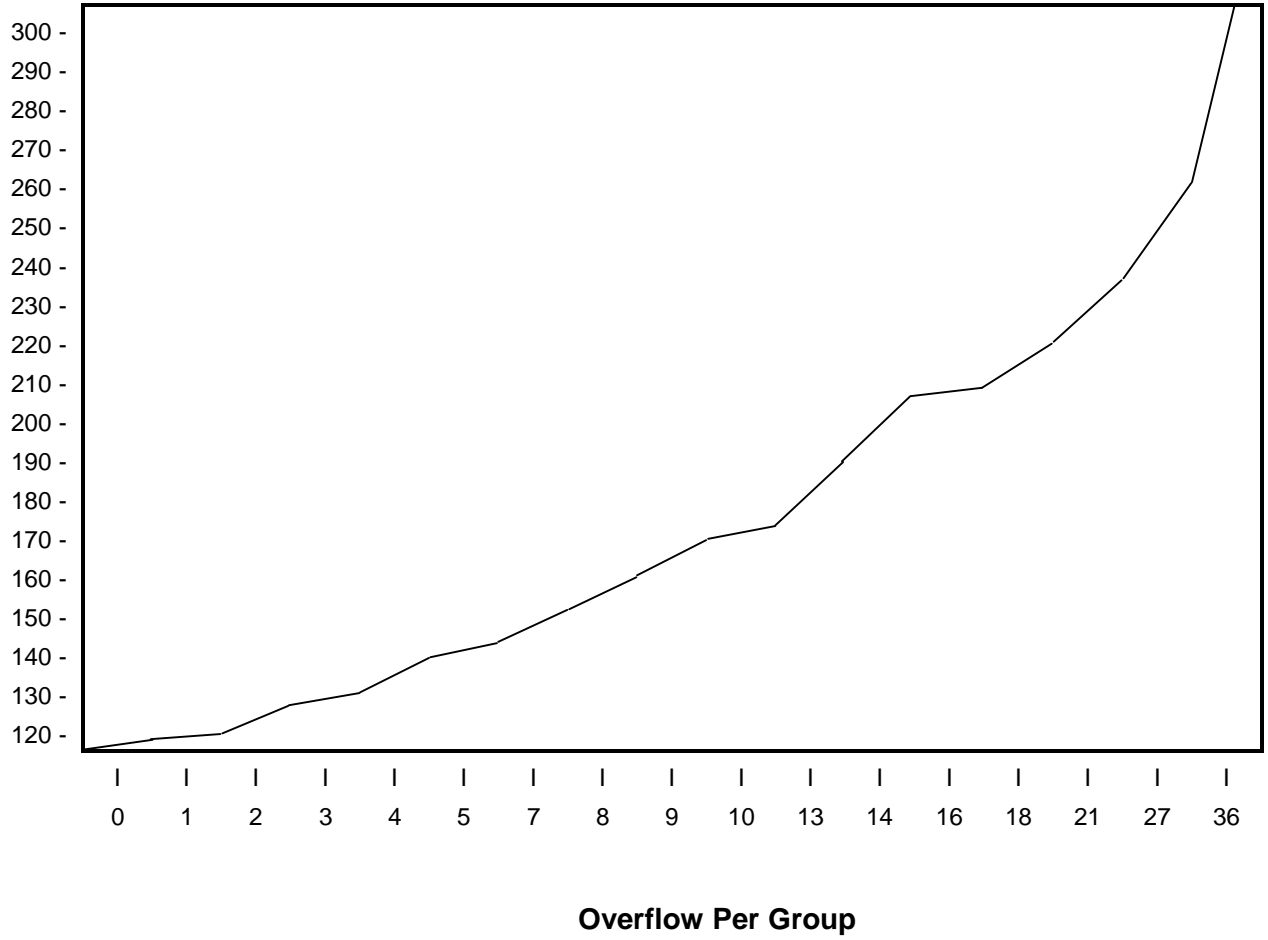


Figure 6 -- *Access Time as a Function of Overflow*

ACKNOWLEDGEMENTS

Portions of this presentation have been previously published by the authors in *International Spectrum Tech*, November 1990, *SSELECT Magazine*, March/April 1991, and *SSELECT Magazine*, July/August, 1991

"uniVerse" is a trademark of VMARK Software, Inc., Natick, Massachusetts.

"UNIX" is a trademark of Bell Laboratories.

"PRIME INFORMATION" is a trademark of ComputerVision, Inc., Natick, Massachusetts.

ADDITIONAL REFERENCES

"The uniVerse of Hashing" (Part II)

SSELECT, Vol. 5, No. 2, July/August 1991, pp. 20-26

"A Hashing Tale Part I - A Guide to uniVerse Hashed Files"

SSELECT, Vol. 5, No. 1, March/April 1991, pp. 10-16

"Hashed Files - Structure and Performance Using PICK"

INTERNATIONAL SPECTRUM TECH, November 1990, pp. 5-9