

The uniVerse of Hashing (Part II)

by

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

Published in SSELECT Magazine
July/August 1991

The uniVerse of Hashing (Part II)

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.

Let's examine all three file parameters, focusing primarily on Type. In understanding how these parameters interact, we may gain some insight into why choosing the best Type may be difficult.

Two of the file parameters, modulo and separation are easy-to-grasp concepts. It's simple. A file holds data which occupies disk space. The disk space assigned to a file is subdivided into a number of "buckets". The modulo defines the number of "buckets" in the file (to hold data) and the separation multiplied by 512 bytes defines the physical size of a "bucket". A "bucket", more accurately called a "group", may vary in size from file to file depending upon the separation parameter. Separation defines the amount of disk space allocated to the group. Space is allocated to a group in units of 512 bytes. For example, a modulo 3 file which has a separation of 4 would consist of three groups with each group sized to hold 2048 bytes of data (512 times a separation of 4).

A good example of a common non-computer related hashing problem is a filing cabinet. Let's imagine one which has three drawers (Modulo 3). Each drawer will hold 2048 sheets of paper (pretend these are bytes). The size of each drawer is defined as a separation of 4. A "separation" is a space which will hold 512 sheets of paper. Therefore, a drawer which will hold 2048 sheets must be a separation of 4. Originally the "frame" size of a PICK system was 512 bytes. The number of frames assigned to a group was called the "separation". Think of "separation" as a measurement of disk space, 512 bytes of space, just as you think of 60

seconds as defining a chunk of time, a minute.

To assist in giving you a cognitive model or "picture" of a modulo three file, let's use a figure of three boxes.

address	0	2048	4096
	Group 1	Group 2	Group 3
	2048 Bytes	2048 Bytes	2048 Bytes

A Hashed File with modulo 3 and separation 4

Each box represents a group. For our example we have defined the separation as 4. Therefore each box represents a chunk of disk space equal in size to 2048 bytes (512 times 4). Implicit in creating a file with a modulo 3, separation 4 is that the allocated space is sufficient to hold the data. The allocated space is 6171 bytes (modulo multiplied by separation multiplied by 512).

Having established the physical size and shape (3 groups) of our file, let's focus upon the process called "hashing" which assigns the data records to the groups. Because there are three groups, a particular data record may be placed into one of three locations: group 1, group 2 or group 3. Choosing a location to store a data record is only a minor consideration. The primary objective of hashing is to provide a process through which a specified data record may be located and retrieved quickly. Just as a paper filing system commonly depends upon alphabetization for easy and efficient document retrieval, the uniVerse file system utilizes a hashing algorithm which accomplishes the same goal as alphabetization.

In a paper filing system, the quantity of data to be stored is usually subdivided into the drawers of a filing cabinet. Theses 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). 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 last 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 control 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.