

# Eliminating Overflow

We all know that overflow in files is “bad”—therefore, when we see files with lots of overflow we’d like to get rid of it. But, where is the overflow coming from and what is it going to take to get rid of it? More importantly, will the side effects of what we have to do to get rid of the overflow cause us more problems than the overflow? That’s the issue we want to describe here.

First, let’s briefly review a description of overflow and what causes it. Overflow occurs in uniVerse and UniData database files when there is not enough space in the selected file buffer to write/update data. The file system then allocates another buffer to the file. The primary buffer is then linked to the overflow buffer and the excess data are written into the overflow buffer.

There are ONLY three causes of overflow. One, the number of buffers allocated to the file by the modulo is too small to hold all of the data we try to store in the file. Two, the hashing algorithm in use (the TYPE) is not distributing the data evenly among the buffers so some buffers have more data than others and thereby overflow into additional buffers. Three, there are data records which contain more bytes than a buffer will hold.

Eliminating overflow caused by the first item is simple. Just increase the modulo. Finding the best TYPE is more difficult and sometimes there is not a “good” choice but increasing the modulo can minimize overflow and may improve performance when the even distribution of data per group is extremely poor. Increasing the modulo will increase the physical size of the file. Processes which must read the entire file such as SORT, LIST, and SELECT take longer to execute as the physical size of the file increases.

Eliminating overflow caused by large data records can only be addressed by increasing the separation, the buffer size. The side effects of increasing the separation may be detrimental to system performance. As an example we’ll use a real file (copied from a live application) which appears to have excessive overflow. We will explain why overflow caused by large records should be considered as a special kind of overflow. It does not create the same performance degradation due to overflow that may be caused by a bad distribution (wrong type) or a file which is too small (modulo too small).

Our example is a typical cross-reference file from a “live” application. It is an “inverted list” file to provide alternate key access to another data file. A customer master file might use a customer number as a record key and have a cross reference file where each word in the customers’ name field is used as a key to the cross reference file. For example our customer number might be 1234 and there would be a reference to “1234” in the cross reference record which has the key of FITZGERALD. These files usually have a very non-homogeneous mix of record sizes. For instance, a name cross reference file will have many more SMITHs than ZBELINSKIs in most cities in the US. Therefore the SMITH record might be very large and the ZBELINSKI record might be very small. Our “live” file has a wide range of record sizes, as shown here:

<b>Number of Records</b>	<b>Record Size</b>
3,157	< 1000 bytes
2,053	> 1000
1,291	> 2000
880	> 4000
635	> 8000
378	> 16000
102	> 32000
5	> 64000

In reading this chart we would say that there are 3,157 data records with data record lengths (number of bytes) which vary in size from 1 to 999 bytes (less than 1000 bytes).

The largest record is around 78,000 bytes. A separation of 164 (83,968 bytes) would be needed to store a record of this size WITHOUT overflow. This would be a block size of 82 for UniData files and the maximum allowed is a block size of 16! Is there a message here? It is interesting to note that the file contains a record with the key of "-" which is 65,925 bytes long. (This record is probably not very useful. Many cross-reference systems have the capability of "no-oping" certain frequently used words, etc. For example, in a name cross-reference one might "no-op" the words "Mr.", "Co.", "Inc.". )

We sized the file according to FAST's recommendations. Using the FILE.STAT utility after resizing, it was obvious that the file had a lot of overflow:

<b>% Full</b>	<b>25%</b>	<b>50%</b>	<b>75%</b>	<b>100%</b>	<b>125%</b>	<b>150%</b>	<b>175%</b>	<b>200% full</b>
<b>No. of Groups</b>	2903	413	368	217	150	87	85	966

Note: This chart describes the number of groups which are less than 25% full. This means that 2903 groups in this file (with a total of 5,189) have between 0 and 500 bytes in each group. The separation is 4 which creates a buffer size of 2,048 bytes. In the 50% full column there are 413 groups which contain between 500 and 1000 bytes of data. There are 368 groups containing between 1000 and 1500 and 150 containing between 1500 and 2000. I've dropped the few extra bytes to make this easier to read. Groups represented in the columns 125%, 150%, 175% and 200% have overflow. There are 150 groups which have between 2000 and 2500 bytes of data. Because one buffer can hold 2000 bytes, we know there will be two buffers, one of which is an "overflow" buffer. In the 200% column there are 966 groups which have more than 3500 bytes of data per group. Each of these groups will have a primary group plus at least one overflow group. The overflow group will contain at least 1500 bytes of data (capacity 2000). We cannot tell how large the largest group is from this chart.

The uniVerse utility HASH.HELP recommended a modulo 5227, type 4 and separation of 32 for the file. I resized the file according to these recommendations. The overflow was somewhat less:

<b>Groups</b>	<b>25%</b>	<b>50%</b>	<b>75%</b>	<b>100%</b>	<b>125%</b>	<b>150%</b>	<b>175%</b>	<b>200% full</b>
	4331	258	133	125	54	35	82	209

I then attempted to find a set of parameters which would eliminate overflow completely. I found that a modulo 4003, type 3 and separation of 164 did this.

<b>Groups</b>	<b>25%</b>	<b>50%</b>	<b>75%</b>	<b>100%</b>	<b>125%</b>	<b>150%</b>	<b>175%</b>	<b>200% full</b>
	3679	283	35	6	0	0	0	0

Okay, great! So why not use these parameters since they eliminate overflow? Because they exact a HUGE penalty! First, look at the disk space used by the file:

FAST's parameters	-	26,957,824 bytes
HASH.HELP's parameters	-	93,896,704 bytes
No Overflow paramters	-	336,207,872 bytes

The HASH.HELP file is 3 ½ times the size of the FAST file. The No Overflow file is 12 ½ times the size of the FAST file. It's obvious that the HASH.HELP and No Overflow versions contain a lot more wasted space.

Okay, well disk drives are a lot cheaper than they used to be. So maybe the disk space price is worth it if the file performs better. But does it? I constructed a series of small benchmarks to look at performance of the test file. I first did a simple SELECT of the file and timed it. Then I did a SSELECT of the file in order of the first field. Finally, I ran a program and randomly accessed each record in the file, read it, added a new value and then wrote the record back to the file. This read and update process would be similar to the way the file is accessed by programs. This simple benchmark is by no means well-controlled or scientific, but still is revealing, I believe. Compare the times of the three sets of parameters:

FAST's parameters	-	SELECT	1 second
		SSELECT	1 second
		Read & Update	2 seconds
HASH.HELP's parameters	-	SELECT	1 second
		SSELECT	2 seconds
		Read & Update	8 seconds
No Overflow parameters	-	SELECT	3 seconds
		SSELECT	3 seconds
		Read & Update	5 minutes 35 seconds

So FAST's parameters were a little better in the SELECT and SSELECT but not by a whole lot. However, the Read & Update test was revealing—the HASH.HELP file was four times slower than the FAST file and the No Overflow file was 167 times slower!

There is also a theoretical explanation for this. In both the HASH.HELP and No Overflow files we increased the separation, attempting to reduce overflow caused by the presence of large records in the file. The increased separation artificially disguises overflow by changing the buffer size. This is somewhat similar to daylight savings time—by changing the clock we aren't really saving anything, we are just agreeing to wake up at a different time. The larger buffer size, however, causes ALL file access to do more I/O because more disk space must be read with each system call made to retrieve data.

uniVerse has a mechanism that handles large records fairly efficiently by storing only the key and a small part of the data within the mainstream of the group. The remaining data is in an overflow area that is only accessed when that data is needed. Increasing the separation defeats the large record mechanism by redefining what a large record is.

Bottom line: Yes, there is overflow—mostly due to large records. We do not feel that increasing the separation has any benefit—in fact, it makes the file slower to access! Our suggestion is to follow FAST's recommendations and accept the fact that large records will cause overflow in the file. By using FAST you will be minimizing the overflow to the extent that it is possible without going to harmful extremes.

If you are interested in a quick way to check a particular file for large records, see our article on "Overflow Due to Large Records". We offer you two I-descriptors and a Retrieve/UniQuery sentence to help you inspect your files for large records.