1. OPERATING SYSTEMS - REAL AND VIRTUAL MEMORY 2

2. MEMORY STRUCTURES 2

2.1 LAYOUT IN MEMORY 2
2.2 DIMENSION ORDERING DEFAULTS AND WHY 3
2.3 HOW SPARSITY IS HANDLED IN MEMORY STRUCTURES 3
2.4 WHY SPARSE MEMORY STRUCTURES MAY NOT SAVE MEMORY 4

3. SHARED STRUCTURES 5

3.1 LAYOUT ON DISK 5
3.2 IMPORTANCE OF DIMENSION ORDER FOR CONSOLIDATE/CALCULATE 5
3.3 SETTING THE CACHE SIZES 5

4. HASHED STRUCTURES 6

4.1 OVERVIEW 6
4.2 HASH HEADS, WHAT AND WHY 6
4.3 IMPORTANCE OF <SPARSE> ATTRIBUTE (NO SPARSE ALGORITHMS WITHOUT IT) 6
4.4 BUCKETING 6
4.4.1 Components of Buckets 6
4.4.2 Bucketing Methods 7
4.5 CALCULATION OF BUCKET SIZE 8
4.6 HOW CONSOLIDATION OPERATES 9
4.7 HOW CALCULATION OPERATES 9
4.8 CALCULATING THE CACHE SIZE 9
4.8.1 Cache for consolidation 9
4.9 BENEFIT OF MANY USERS VIA SYSTEM FILE CACHE 10

5. CHOOSING A STRUCTURE TYPE 10

1. Operating Systems - Real and Virtual Memory

In order to fully appreciate how to tune Holos structures, it is important to understand how an operating system handles memory for a process. A process running within a computer system operates with virtual memory. That is it has an address space that appears to the process to all be in memory all the time. For convenience, most systems deal with memory in units of pages. The size of a page varies by operating system. For VMS it is 512 bytes, a number of UNIX systems use 4096 bytes.

A processes virtual memory is implemented by the operating system as two types of memory. The first is real memory, RAM, where the running parts of the process must be in order to be executed. The second is paging and swapping memory. This is an area of disk, reserved by the operating system, where it can put process memory pages when it needs the real memory that it occupies.

As processes are started in a machine, they are allocated some real memory to start with. As a process accesses its virtual address space, it will try to touch pages that not actually in real memory (a page fault). This condition is detected by the operating system and the page is loaded into real memory. In a system that is not too busy, this real memory will come from a list of free memory. As the system gets busier and the demand for memory goes up, the system will start to take pages away from processes that haven't used them for a long time. This means that most, if not all, processes within a busy system will not have all their pages in real memory.

A process is not normally penalised by not having all of its pages in memory. The normal behavior of a process means that only some of the full range of pages are ever accessed at one time. As an example, consider a Holos process that is computing a model. All the code within the Holos process that deals with reports, worksheets, data loading etc. is not required, so the process would not suffer from not having those available in real memory. The set of pages that are needed by a process to complete a particular operation are known as the working set.

It follows therefore that if the operating system is not able to give a process enough memory to hold at least the working set, then the process performance will suffer.

2. Memory structures

2.1 Layout in memory

Holos memory structures are organised in a tree like structure. At the bottom of this tree are the data 'legs'. Each data leg has a space reserved for every field in the last dimension in the structure. Each of these data legs are pointed to by pointers representing the fields of the next dimension above. **Figure 2-1** illustrates the layout for a simple 2 dimensional structure, having dimensions A and B.

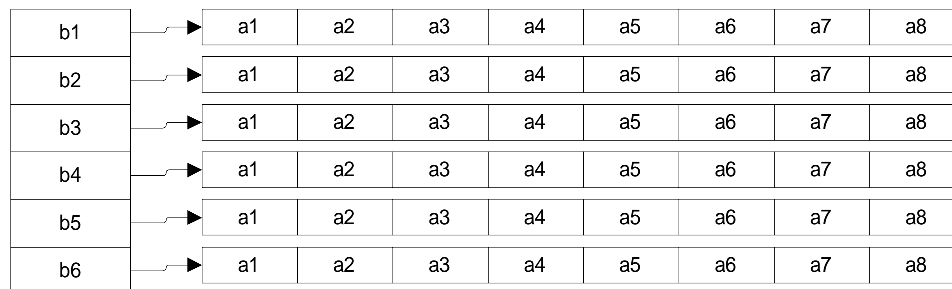


Figure 2-1

Each field in a data leg consists of space for the cell value, either 4 (single precision) or 8 (double precision) bytes, and a 4 byte field for internal use. Each field in a pointer leg consists of just a 4 byte pointer value. From this you can see it is possible to calculate the memory requirement for a memory structure. The resultant figure will only be an approximation, since the data and pointer legs are actually constructed of multiple pieces linked together in order to be able to handle very large dimensions.

Adding a third dimension would simply replicate the structure shown in **Figure 2-1** for each field in the new, C, dimension. It would also add a new pointer leg to represent each field in the C dimension. The resultant structure is shown in **Figure 2-2**.

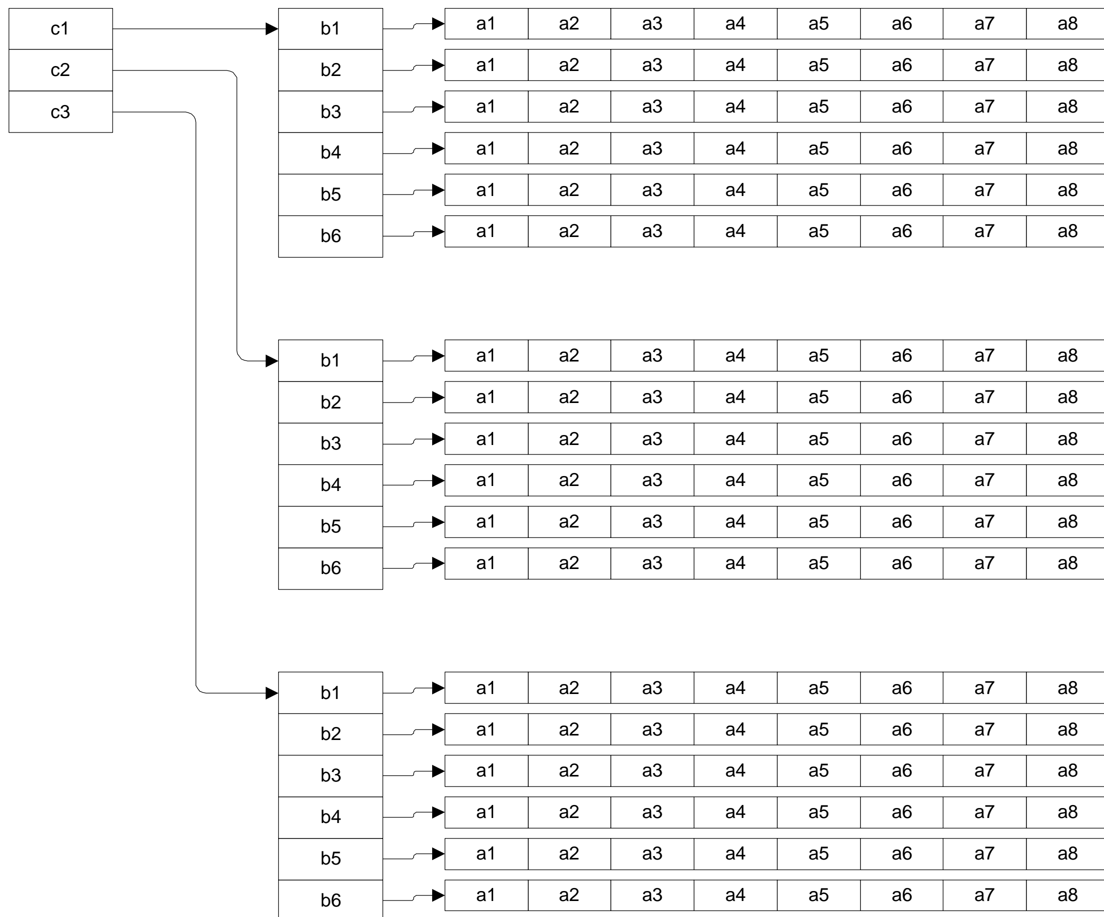


Figure 2-2

2.2 Dimension ordering defaults and why

In memory structures, the dimensions are ordered by Holos to be as efficient in memory usage as possible. This means that the smallest dimension will be placed first in the list, the next dimension will be the next largest and so on until the last dimension which is the largest. This is the default behavior for memory structures, but can be changed by adding the NO SORT option to the STRUCTURE statement. To determine the dimension order used to the create the structure, use the SHOW statement. Amongst other information, this will show the dimensions used within the structure , and the order that Holos arranged them in.

2.3 How sparsity is handled in memory structures

If a Holos memory structure is created using the WITHOUT clause, not all data and pointer legs will be created. As data is added to a sparse memory structure, the data leg to hold the field, if not already present, is created along with any necessary pointer legs. Using an example based on Figure 1, if we created the structure using the statement:

STRUCTURE example a,b WITHOUT {a.*,b.*}

Then no data or pointer legs would exist, and the structure would not be occupying any memory beyond that required for its symbol table entry. If we were then to insert a single cell, {a.a3,b.b4}, the memory used by the structure would look like **Figure 2-3**.

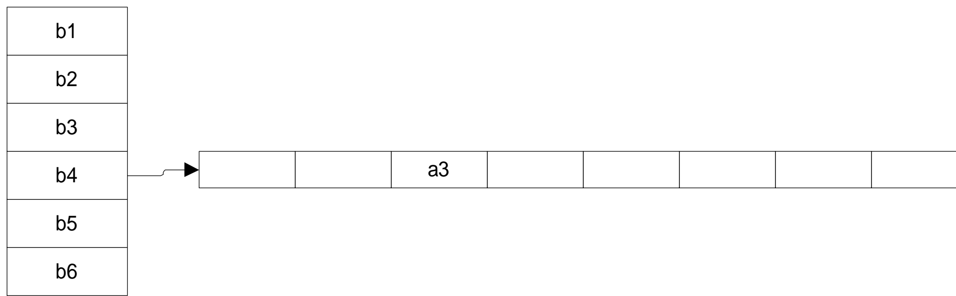


Figure 2-3

Note that the data leg contains spaces for all the other cells in the dimension for field b4, even though there is nothing in them. For all current versions of Holos, allocating space for one cell will always allocate space for all fields in the dimension. The default dimension ordering for memory structures may therefore not be the best for a sparse memory structure. Using this same example structure, but creating it with the NO SORT option, would give a memory layout as illustrated in **Figure 2-4**.

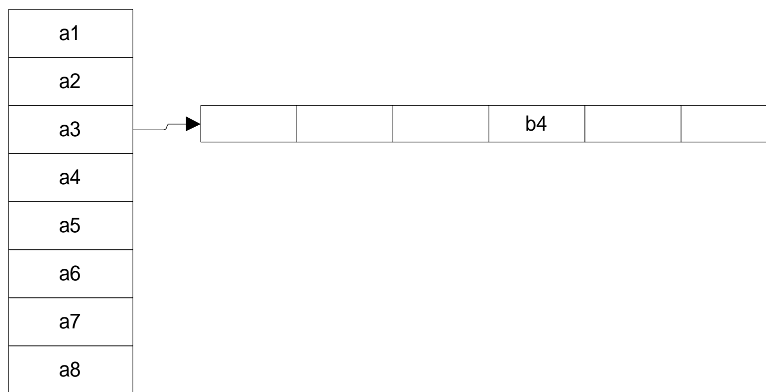


Figure 2-4

From the earlier section, we can work out how much memory is used by each method. For **Figure 2-3**, we have 8 data cells allocated (8 * 8 bytes = 64) and 6 pointer fields (6 * 4 bytes = 24), giving us 88 bytes. The example in **Figure 2-4** is organised as 6 data cells (6 * 8 bytes = 48) and 8 pointer fields (8 * 4 bytes = 32), giving us 80 bytes. This example doesn't give us a very big saving, but it clearly shows that having the largest dimension last in the structure is likely to be a bad choice for sparse memory structures.

2.4 Why sparse memory structures may not save memory

The previous examples show the amount of memory that might be allocated when just one cell is inserted into an empty memory structure. Looking at **Figure 2-5**, you can see how just 6 cells would allocate the same memory as a fully populated memory structure (48 cells). This further emphasizes the need to consider the dimension order when creating a sparse memory structure.

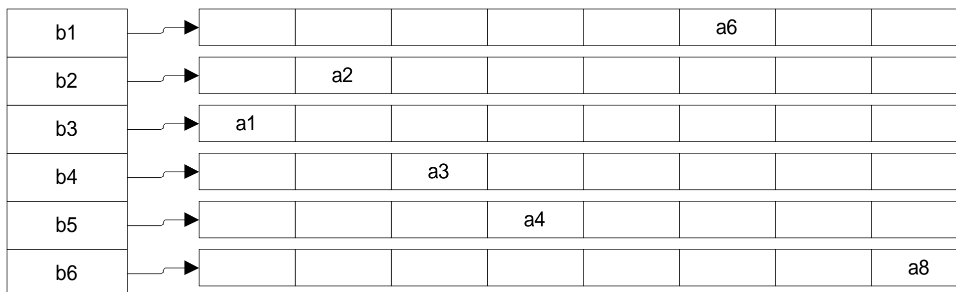


Figure 2-5

3. Shared structures

3.1 Layout on disk

Unlike memory structures, shared structures do not use pointers to locate cells. Instead, a space is reserved in the file for every cell. This means that the size of the file is easy to calculate. Just multiply the number of fields in each dimension together, and then multiply by the size of a cell; 4 for single and 8 for double precision. This storage method is very space efficient for dense structures. It is also very quick to locate any cell, since it only requires Holos to calculate a cell number for each reference to know exactly where in the file the cell is stored.

Since each cell must have a location reserved for it, a sparse shared structure does not save any space. This does not mean that a shared structure should not be considered for sparse data. In some situations, a shared structure may still consume less space than a hashed structure.

Another difference between memory and shared structures is that the dimensions are not ordered by Holos. The dimension order specified on the SHARE STRUCTURE statement is the order used to construct the structure on disk. This is because there is no space advantage or penalty in ordering the dimensions one way over another.

3.2 Importance of dimension order for consolidate/calculate

Although there is no space advantage to ordering the dimensions in a shared structure, it will be necessary to order them yourself, if the structure is to be calculated or consolidated. If we take the example structure from **Figure 2-1** and create it as a shared structure, the layout of cells in the file would be as **Figure 3-1**.

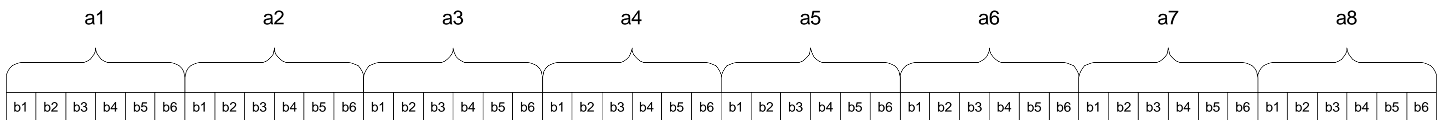


Figure 3-1

If the dimension, B, contained a hierarchy, then the organization above would work well. A single cache buffer of precision*6 would cover all values, b1:b6 for one field in the a dimension (I know the smallest cache buffer for a shared structure is 1K, but bear with me). The consolidation would start by reading all the b values for field a1 into the cache. Since all the input and output values are in the cache buffer the consolidation would occur and then be written out to the file and the b values for a2 read and consolidated, and so on until the complete file was done.

If the hierarchy was in the A dimension however, the consolidation would be a lot slower. Assume a simple hierarchy of a1:a7 summing into a8. This would need to be done for each of the fields in b. To start the consolidation, read {a1,b1} into the cache and get the value. Next value needed is {a2,b1} but that value isn't in the cache, so read it in. This proceeds down the line reading one value each cache read. So after eight cache reads and a write we have only consolidated b1. We have to go through the whole procedure again for each of b2 through b6.

It would be better for the last example to reorder, making dimension A last to give the organization shown in **Figure 3-2**.

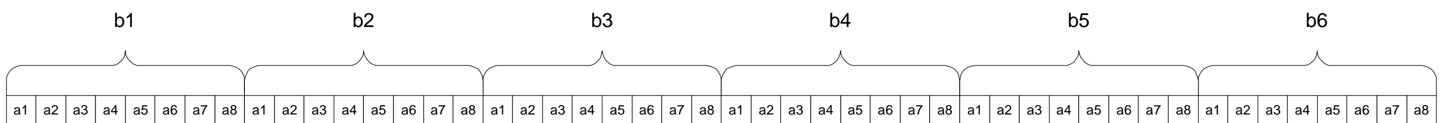


Figure 3-2

We would need to adjust the cache buffer size to cover all A fields so it would be precision*8 bytes.

3.3 Setting the cache sizes

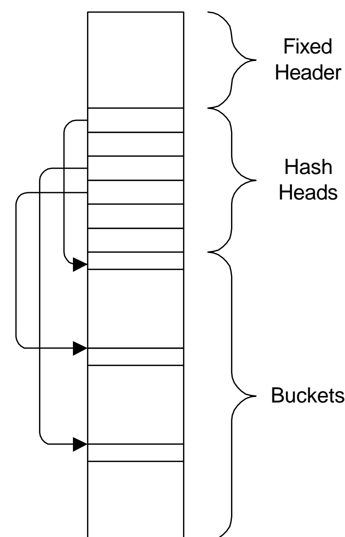
From the discussion in the previous section, it is probably becoming clear how to order the dimensions and calculate the cache size. All practical applications will have more than two dimensions, but the principals described can be grown to match the problem. If the application has 7 dimensions, but only two with hierarchies then the diagrams above represent the lowest level on the disk. The structures above are simply replicated for every field combination of the other 5 dimensions. The ideal cache setting will be enough

space to hold the whole slice, which in the example would be precision*6*8. If you have enough cache allocated, and the dimensions are contiguous, then it doesn't matter too much whether the cache is a small number of large buffers or lots of small buffers, although larger buffers are probably to be preferred.

4. Hashed structures

4.1 Overview

Hashed structures are designed specifically to handle sparse data. Unlike the previously described structure types, hashed structures only store cells that contain data. No space is allocated for cells that do not exist. The internal structure of a hashed structure is divided into 3 pieces.



1. A constant size, 1024 bytes, and is the file's header. It contains information about the number of dimensions in the structure and their order, and the number of fields in each dimension.

2. The hash table. Each entry in the hash table (a hash head) always occupies 4 bytes. The number of hash heads in a file is controlled by the COUNT statement in the ALIAS block.

3. The last piece of a hashed structure file is that containing the buckets. These follow immediately after the hash heads. Buckets contain one or more cells and are linked together by a Next pointer.

4.2 Hash heads, what and why

The purpose of hash heads in the file is to speed up access to the cells in the file. If the cells were simply inserted into the file without hash heads, then each time a cell was requested from the file, all cells in the file would need to be searched to find it. This would be extremely time consuming for even modestly sized files. To make finding a cell faster, the cells in the file are divided into a number of separate lists. Each of these lists is headed by a hash head. Which list a cell is inserted into is calculated by the hashing algorithm. Holos uses the 'cell number' (which can be from 1 to the total number of potential cells in the structure) and the number of hash heads in the file to decide upon a particular hash head to use. This means that when a cell is requested from the file, instead of searching every cell in the file, only (cells in file)/(hash heads) cells need to be searched. Note that this an average, since the actual number of cells on each hash head's list will vary. It's important to know that the best distribution of cells over the available hash heads occurs when the number you select for the hash heads is a prime number.

4.3 Importance of <SPARSE> attribute (No sparse algorithms without it)

It is important, when creating a hashed structure, to declare it with the <SPARSE> attribute if you want the structure to be treated as sparse. If the attribute is not specified, the structure will be consolidated and calculated using non sparse algorithms. This may, at first glance, seem to be an unnecessary inconvenience. Although it seems obvious that a sparse algorithm would always be used, in fact this may not always be the case.

4.4 Bucketing

Bucketing is a term used to refer to the mechanism whereby cells are collected together to provide space and processing efficiencies. The simplest and generally least space efficient method of bucketing is to have only one cell per bucket. This was the only form of bucketing available in Holos 4.0 versions. Although not strictly bucketing, because only one cell is stored, it makes the explanation more consistent and understandable to refer to them as buckets.

4.4.1 Components of Buckets

There are presently 4 bucketing methods in Holos, and they are constructed using 4 basic component types.

4.4.1.1 The Key field

The key value is an important concept in the understanding of hashed structures. It is a value that is unique for every cell in the structure. Since it is derived from the 'cell number' it has to have enough space reserved in the bucket to accommodate the largest possible number for the structure. Hence why Holos gives you the number of potential cells when you create a hashed structure. The key value stored in a bucket is always a multiple of 2 bytes. Thus knowing the potential number of cells for a structure will tell you the key size that will be allocated.

Potential number of cells	Key size in bytes
65,535	2
4,294,967,295	4
281,474,976,710,655	6

For the 2 bucket dimension methods, the last dimension (the bucket dimension) does not need to be included in the calculation for the Key, since fields in that dimension are uniquely identified by another method. Therefore, if you are using BUCKET DIMENSION you can divide the potential number of cells by the number of fields in that dimension, and use the result in the table above to find the key size. It may not always make a difference, but for large bucketed dimensions, the key size could be reduced from 6 to 4 bytes, further saving space.

4.4.1.2 The Next pointer

The next pointer is simply a pointer to the next bucket in the same chain as this bucket. It is always 4 bytes, and there is only ever one per bucket. If this is the last bucket on the chain, this will contain 0.

4.4.1.3 The Value field

This is the most important part of a bucket, the cell value. The size depends on the precision in use when the hashed structure file was created. It will be 4 bytes if created in single precision or 8 bytes if created in double precision. This is the internal size of a floating point value, not to be confused with the precision value that Holos displays from the defaults record, which is the number of digits of displayed, when you WRITE or PRINT a value. The number of values in a bucket depends on the bucket type in use, but will be at least one.

4.4.1.4 The DFN field

The DFN field is only used for type 2 (NK_DV) buckets. The DFN field is always 2 bytes, and there is one DFN field for each value field in the bucket.

4.4.2 Bucketing Methods

4.4.2.1 Single cell per bucket (NVK, type 0)

This is the simplest bucket type, and in Holos 4.0 was the only available method. Each cell inserted into a hashed structure file added a new bucket to the file. Each bucket consists of a Next pointer, a Key field and a Value field.

Next Pointer
Key Field
Value

4.4.2.2 Multiple cells per bucket (N_VK, type 1)

This method is a simple derivative of the NVK bucket type previously described. This type has more than a single cell value, but the cells are only related by having the same hash head, so each value field in the bucket has its own associated key value. As with all other bucket types, there is a single Next pointer. This bucketing method does not really save very much space, but since it groups together cells having the same hash head, searching for cells is much quicker than for the NVK, where the same cells are probably widely spread across the file, resulting in lots of files reads to find cells near the end of the hash list.

Next Pointer	
Value	Key Field
Value	Key Field
Value	Key Field
Value	Key Field
Value	Key Field

4.4.2.3 Bucket Dimension with count (NK_DV, type 2)

This method can give better space savings than the previous 2 methods. Each bucket contains fields from the same row of the same dimension. Since the values are related in this way, they can share a single Key field. To uniquely identify each Value field, the field's DFN is stored with it.

Next Pointer	
Key Value	
Value	DFN
Value	DFN
Value	DFN
Value	DFN

4.4.2.4 Bucket Dimension (NK_V, type 3)

This bucket type gives the best space saving. Each bucket contains every field from a single dimension. In this respect it is very similar to the data leg of a memory structure. The syntax, BUCKET DIMENSION, does not specify the dimension to be bucketed explicitly, but implicitly as the last specified dimension when the structure is created. The bucket contains a Next pointer, one Key field as with the NK_DV method, and then a Value field for each field in the dimension. Since every field of the dimension is in the bucket, no DFN is required. The cell's position in the bucket defines it.

Next Pointer
Key Value
Value
Value
Value
Value
Value

4.5 Calculation of bucket size

From the previous descriptions of the bucket formats and their components, it is very easy to work out how big a bucket will be. In order to calculate the size of a bucket we need 4 pieces of information:

1. The key size. Use the table in section 4.4.1.1. The commonest values are 4 or 6 bytes. (key_size)
2. Whether in single or double precision, to give us 4 or 8 bytes. (val_size)
3. The bucketing type.
4. The number of cell spaces in a bucket. For NK_V, this is the number of fields in the dimension. (cell_count)

For the NVK bucket type, the calculation is the same as for the N_VK type with cell_count set to one:

bucket size = 4 + key_size + val_size * cell_count

For NK_DV bucket type, size is:

bucket size = 4 + key_size + cell_count * (2 + val_size)

For NK_V bucket type, the size is:

bucket size = 4 + key_size + cell_count * val_size

4.6 How consolidation operates

The scalar-mode hashed structure algorithm passes through the input data in the hash file linearly. In order for this to work, the algorithm assumes that the data that it is scanning through does not contains result information from another pass. That is why the first operation that the algorithm does is to scan the input data to delete previous results. Although it says ‘x previous results deleted’, it could also be deleting data loaded into the structure from your load procedure, if that was loaded into ‘result’ cells.

Once the result cells have been deleted, the first pass starts. Holos sets up pointers to mark the beginning and end of the data, and reads the first cell. Note that the reading of the input data does not use the hash pointers. As each cell is read, the hierarchy definition is used to work out the parent cell. Holos then looks to see if the parent cell exists in the output data for the current pass. This does use the hash table and bucket next pointers to locate the parent. If the parent is not found, a new cell is created (it may be in an existing bucket) and the child value put into it. If the parent is found, then this value is added to that already in the parent. At this point we need to explain how and why the results for each pass are kept separate.

4.7 How calculation operates

4.8 Calculating the cache size

Unlike all the other settings in a hashed structure, the size of each cache buffer and the number of cache buffers can be changed after the file has been created, since they are really properties of the file open rather than the hash structure file. The setting of the cache size depends on the task being performed. These can be divided into 2 categories.

1. Loading data, consolidation and calculation. This sort of operation tends to be done once, sometimes overnight, and can be very time consuming. The access to the structure file tends to be linear so cache buffers that are large (I use 10k or 20k) and fewer in number give good performance.

2. Reporting. These operations are usually done by more than one user at the same time on the same file. The cache requirement to access the file is going to be lower than that required to consolidate the structure. It is also going to be more ‘random’ than linear, so small buffers but lots of them will work better. The lower you can get the overall cache size for this the better, since this memory overhead is multiplied by each person using the application. For a system with 100 users, every 1 Mbyte of cache will use another 100 Mbytes of memory in the system.

4.8.1 Cache for consolidation

Calculating the ideal cache for consolidation can only be done once a consolidation has already been performed. This means you need to select some values to start with. Here you might need to throw yourself upon the mercy of the system administrator. You really want to have the machine to yourself and set the cache to the maximum available to you. Err on the low side of the available

memory, because if you set the cache just a little bit too high, it will definitely go slow. To make it easier to calculate the settings, use an expression with 1024 in it for both the WINDOW LENGTH and COUNT, for example:

```
WINDOW LENGTH 10*1024, COUNT 10*1024      ! 10*10 = 100 Mbytes
```

The other thing you must do is to start the consolidate with a log file and using the LOG option. Unless you do this you won't have any information on which to base the calculations. To save yourself having to scan through a lot of information, set the count for the LOG clause to at least the default (10,000) or larger. This means that each consolidate pass should only have 3 lines generated in the log file. The first just says 'Starting pass n…', the third has 'x results created during pass n'. Both of these can be ignored for our purposes. The second line is the one of interest. You need to extract from the end of the line the data size value. This is the total size of the hashed structure file in bytes at that point. When you have the data sizes for the end of each pass, subtract from each value the value from the previous pass. These results will be the size taken for each pass. Select the largest value from the results. This is one of the values we will use to calculate the cache size. The other piece of information needed is the size of the hash table. This is simply the hash count * 4.

Now we have the two values we need, the size of the hash table and the size of the largest pass, both in bytes. Next we decide on a cache buffer size. I usually opt for 10k or 20k, let's use 10k for this exercise. Knowing the size of each cache buffer, we can calculate the number of buffers needed to hold the hash table and the number of buffers needed to hold the widest pass.

Buffers required for hash table = size of hash table / 10k
Buffers required for widest pass = size of widest pass / 10k

Round both results up to the nearest whole number and add them together. This gives us the number of buffers required to hold the widest pass and the hash table at the same time. We also need at least 2 more buffers in order to read the input data. That gives you the total number of buffers required to consolidate this structure for your selected buffer size.

4.9 Benefit of many users via system file cache

Having arrived at an ideal cache size setting to enable a reporting application to run reasonably, you multiply the per user memory for the application by the number of anticipated concurrent users and arrive at an amount of memory considerably larger than the machine has. This usually results in gnashing of teeth, name calling and complaints about how much memory Holos uses. All is not lost however.

Most (probably all) operating systems provide their own cache mechanism for the filing system. This cache is usually dynamically allocated by the system from free memory in the machine. Since this is managed by the operating system, it is shared by all users. This should mean (I haven't proven this theory) that if you have a large number of people all using the same hash structure file, that their individual cache settings can be reduced below the settings you found to be optimal for one or few users. Although Holos will perceive this as going to the file more often than before, because so many users have the same file open there is a better chance that the required bit is in the system's file cache. Reducing the per user cache will also free more real memory for the system to use as file system cache.

In future versions of Holos, it is possible that a shareable cache mechanism will be provided, giving a more controlled way of sharing the cache for large numbers of users.

5. Choosing a structure type

The principal factors used when selecting a structure type are speed and size. A frequent question that is asked is; how many cells should a structure have before changing from a memory to a shared structure? The answer I have seen suggested is a million. I think this qualifies as a wrong answer. The reason I say that is because it implies that the decision point is based purely on the number of cells. The reality is somewhat more complicated.

When considering size as a factor, there are two sizes to bear in mind. The first is the amount of process memory required, either for the whole structure in the case of a memory structure, or the cache for shared and hashed. The other is the disk space required for the file.

It is important to remember that size is relative. Some of our customers have machines with 128 Mbytes total memory. For these people, the size limit for a memory structure is going to be low. Their solution is likely to have disk based structures in it. Other customers have machines with 2Gbytes of memory. These applications could have enormous structures in memory if they needed to.

Bear in mind that a memory structure lives in the process virtual address space. It does not always need to be completely in real memory. Although the structure might be large, if the user is only examining small slices at a time, the performance could be entirely acceptable even with most of the structure paged to disk by the operating system.

Disk based structures are different entirely in their demands on real memory. The disk based structures are accessed via their cache buffers. As with a memory structure, the cache is part of the process virtual address space. But, unlike the memory structure, they must always be in real memory to be effective.

But they may not. The deciding factors are; how much memory is really available for this application? Is it a one off or for every user on the system? How many users? Does it really need to be fast, or is a disk based structure fast enough?

Disk space. Here we have two choices shared or hashed. For a very dense structure, shared is likely to be the best choice, since the overhead per cell is only 4 bytes, but this is for all potential cells. The cell overhead for a non bucketed hashed structure starts at 14 bytes (assuming more than 64k potential cells), so if more than about 25% of the cells are populated, the hashed structure will be larger. If bucketing is used to reduce the cell overhead then the break even point increases.

Another factor that might be relevant is that the shared structure presently has an upper limit of only 63 Mbytes for its cache. If this is too small, a hashed structure, even though it’s larger, might still work quicker because you can give it a much bigger cache,