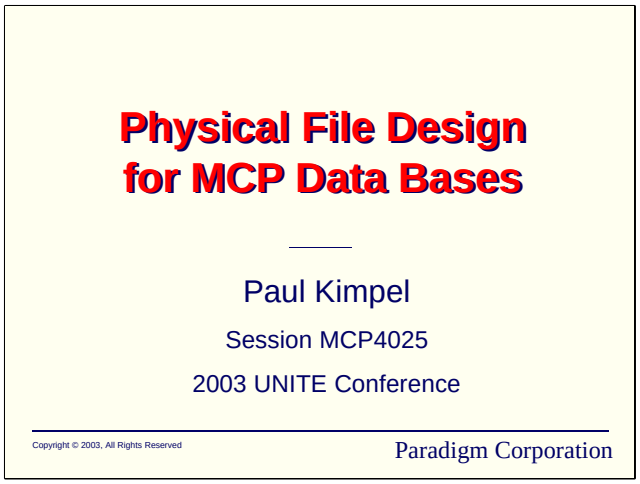


Physical File Design for MCP Data Bases

2003 UNITE Conference

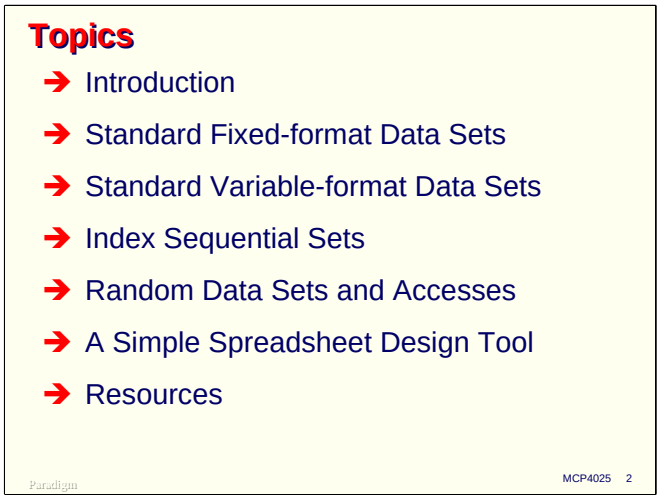
Reno, Nevada

Paul Kimpel

Paradigm Corporation
San Diego, California

http://www.digm.com

e-mail: paul.kimpel@digm.com

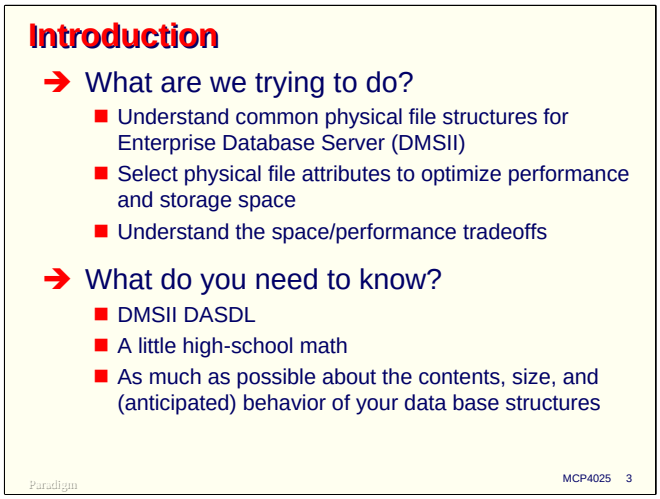


Most of us come to UNITE to hear about new things, but I've taken an old thing as the subject for my talk today. DMSII, now formally know as the Enterprise Database Server for ClearPath MCP, will be 30 years old next year. The basics of how you design data base structures and select their physical attributes, however, has hardly changed during that time. Some new capabilities have come along, and some old issues, such as memory utilization, are not quite a critical as they one were, but physical file design is still a very important part of data base design and administration.

We will discuss design and optimization for a few of the most common DMSII physical structures: Standard fixed- and variable format data sets, Index Sequential sets, and Random data sets and accesses.

I will also present a spreadsheet tool I've developed and enhanced over the years that helps me to calculate the principal physical attributes of these DMSII structures and evaluate tradeoffs between them. I'm making this tool available in the hopes that it will help others as well, or at least provide them with a base from which they can build their own tool.

Finally, we will briefly discuss where you can obtain reference material on this subject.



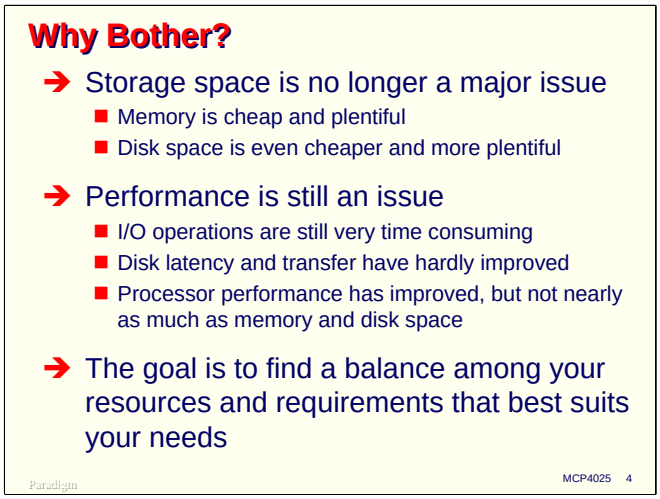
What are we trying to accomplish with this subject? The overall answer is that we want to design better data bases. There are many facets to data base design. This talk will focus on just one—the physical characteristics of data base structures and how you select the attributes that determine how the data in that structure is arranged, stored on disk, and retrieved. In particular, we are trying to

- Understand the most common physical file structures for DMSII
- Understand how to select physical file attributes for these structures so that both performance and storage space will be optimized
- Understand how space and performance issues trade off against each other and how to achieve a balance between them.

In order to do this, you need to know a few things first.

- A basic knowledge of DMSII and its DASDL schema compiler are necessary, since that is the vehicle through which you specify attributes for physical data base structures.
- You also need to be able to do basic high-school math, including logarithms. It would help to have a scientific calculator (one that can do either logs or roots and powers) or a spreadsheet with similar capabilities (most modern spreadsheet programs, including Microsoft Excel, have everything you need in this area).
- Finally, you need to have at least a logical data base design from which to work. That means you need to know what the tables and indexes are, what fields are defined for each table, and what the keys are for the indexes. You also need to have at least some idea of how this data base is going to behave when it's working—how many records each table will have, anticipated patterns of access to the data, frequency of update operations, and so forth. The more you know about the behavior (or anticipated behavior) of your data base, the better you will be able to make decisions about tradeoffs among the various design issues.

I should point out that the topic of physical file design is not just for new data bases. Existing data bases can be candidates for review and modification of their physical file properties. Also, the structure of most data bases changes over time, and it is always a good idea to rework the physical file attributes when a structure changes to keep it in optimum condition.



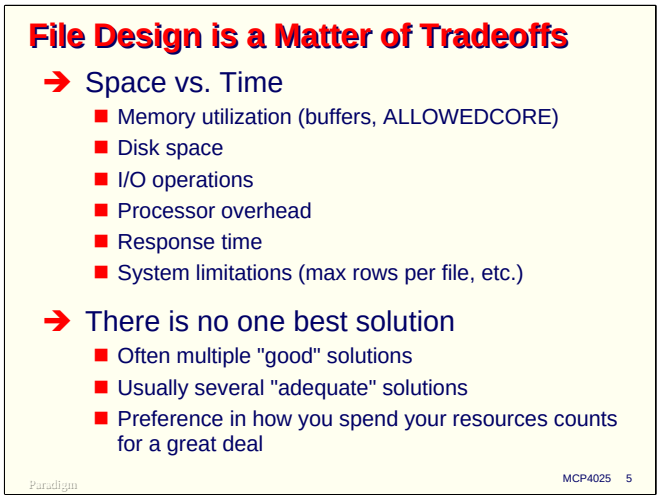
Doing a proper job of physical file design can be a fair amount of work, so why bother? Resources are so cheap nowadays, it is worth the trouble (and cost in person-hours) to spend time on this?

My position is that it's definitely worth the trouble. It's true that storage space is now ridiculously inexpensive, even compared to costs just a few years ago. It is now actually possible to put enough memory on a system, and to have enough disk storage. Memory utilization, in particular, used to be a very worrisome aspect of data base design, but with multi-gigabyte memory capacities in modern servers, it is no longer.

Memory and disk space are only half of the issue, however. Most of the performance aspects of data base operations have improved much more modestly. Processor performance has certainly improved, but not nearly as much as memory and disk capacity. By comparison, the cost and speed of I/O operations has hardly improved at all. Disks are faster than they used to be, but not that much faster compared to processor and space improvements.

In addition, while we have a lot more resources to throw at I.T. problems than we used to, the demands on those resources have also grown. We are being asked to store more data that ever before, and to access more of it (and more frequently) than ever before. Modern user interfaces place a lot more burden on I/O resources as they seek to present users with choices from which to pick, rather than simply provide data when requested. In the transition from YAFIYGI (you asked for it, you got it) to WYSIWYG (what you see is what you get), the disks get much busier.

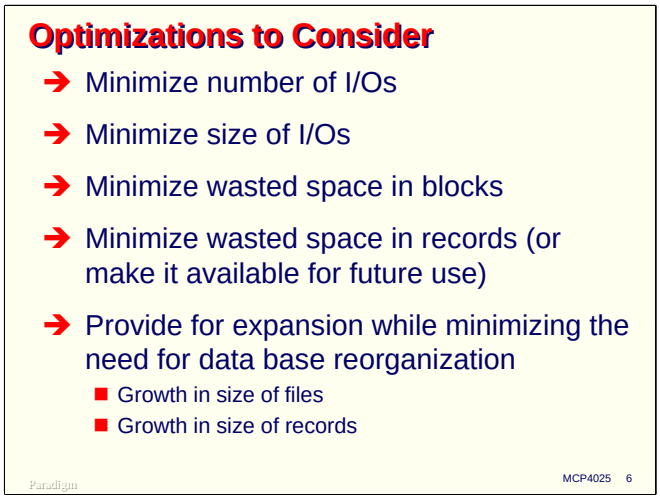
The goal, then, is to find the balance among resource costs that best meets you needs.



File design is a matter of choosing among resource tradeoffs. The fundamental issue in software performance is still, and will remain, choosing between space and time. When you squeeze one, the other is, in some way, going to increase.

It is important to realize that there is usually no one best solution to a particular design problem. There is often more than one good solution, and there are usually at least a few solutions that are acceptable. The differences among the solutions are largely a matter of choices between the tradeoffs involved.

You have to know your own situation and decide for yourself how you want to spend your resources. Since some resources are cheaper than others, you can spend those more lavishly to economize on the others or to improve your overall system performance.



Optimization is nothing more than eliminating the unnecessary. There are a number of types of optimizations you can consider making to physical file structures.

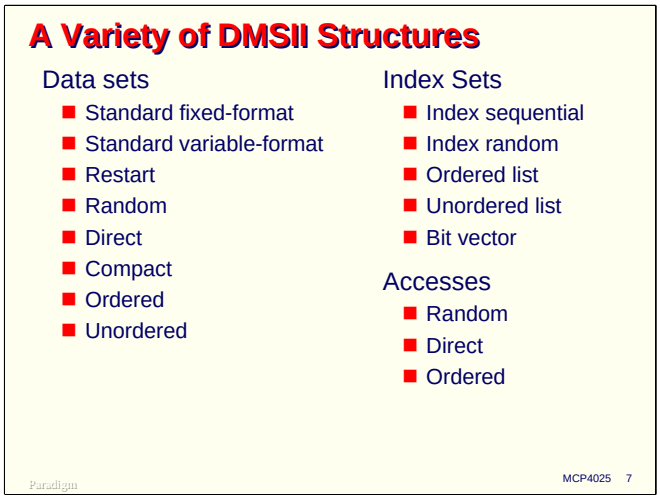
By far the one with the most potential is to reduce the number of I/Os your system must do to meet the needs of your applications. Disk channel queuing, rotational latency, and data transfer time are typically the largest components of on-line response time, whether "on-line" means legacy terminals, client-server arrangements, or the Web.

Probably the second most important candidate for optimization is reducing the size of I/Os. Most files are blocked, which means that during random retrievals you usually read some data that you don't need along with the data that you do. This wastes channel capacity and buffer space, and in most cases clutters the various caches in the system with unproductive entries.

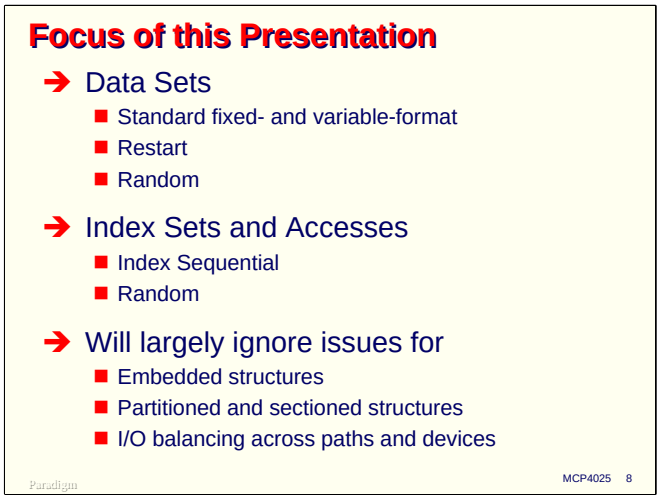
Another significant candidate to minimize is wasted space in blocks and records. While the disk space itself is very inexpensive, the disk heads still have to move past it, so it slows down overall retrieval rates. We also have to back all of this stuff up, and the wasted space generally gets backed up with the good data.

Sometimes due to the structure of the file or the fixed sector size of the disk, you can minimize wasted space only so much. In those cases, you can at least try to turn that wasted space into space that can be used for record expansion. Properly done, this can greatly reduce the need to perform data base reorganizations.

Finally, while not directly an optimization issue, it is always a good idea to think ahead as to how a particular structure is likely to grow, and to plan for that. Many data structures grow over time both in terms of the number of fields they carry and the number of records they contain. Proper planning here can reduce the need to perform expensive and disruptive data base reorganizations.

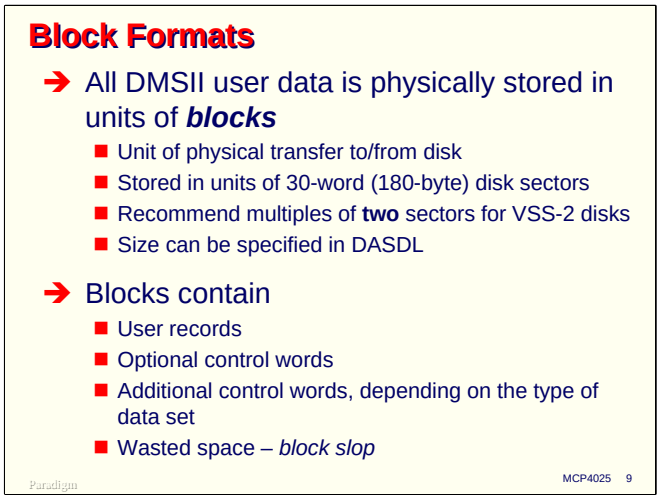


DMSII supports a variety of data structures, each of which has different physical characteristics and design needs.



The focus in this presentation will be on just a few of the more common structures—Standard and Random data sets, and Index Sequential sets.

Even within that restricted group of structures there is not time to talk about all of the options and possibilities that you could encounter. We will largely ignore the design issues surrounding embedded data sets, partitioned and sectioned structures, and the potentially very important topic of balancing I/O for large structures across multiple I/O paths and devices.



In order to evaluate physical attributes of DMSII structures and made decisions about them, we need first to understand how those structures are physically arranged and stored on disk.

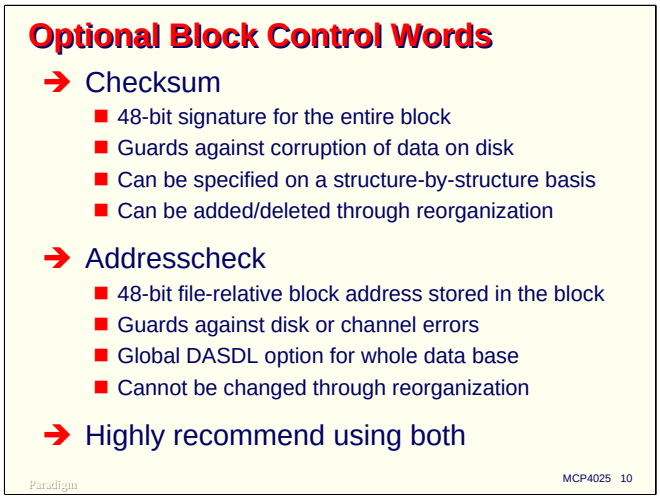
The first aspect of this is **blocks**. All user data is stored and retrieved by DMSII in units of blocks. A block typically contains multiple records or other entities.

A block is the physical unit of transfer between memory and disk. Because of this, blocks always occupy a multiple of 30-word (180-byte) disk sectors. A block always starts on a disk sector boundary. If the size of the block is less than a multiple of 30 words, that portion of the last sector after the end of the data is wasted. I call this wasted space ***block slop***.

Disks with physical 30-word sectors are no longer being manufactured, so modern MCP systems emulate this sectoring in multiple ways. One of those ways, used with industry-standard 512-byte sectored disk, is termed Virtual Sector Size-2 (VSS-2). This technology stores two 30-word MCP sectors in one 512-byte physical disk sector, using 360 of the 512 bytes. I/O to this type of disk is much more efficient if the data transfer always starts on even (MCP) sector boundaries, so it is a good idea to restrict blocks on these types of disks to 60-word (360-byte) increments.

You can specify the size of blocks for a structure in DASDL. If you do not specify a size, DASDL will supply a default one.

In addition to user data, blocks can also contain a couple of optional control words. Depending on the specific type of data set or index structure, there may be additional control words required by that structure.



Each data block can have two optional control words, a checksum word and an addresscheck word. The presence of both of these is determined by options you can specify in DASDL. By default, neither will be present in a block.

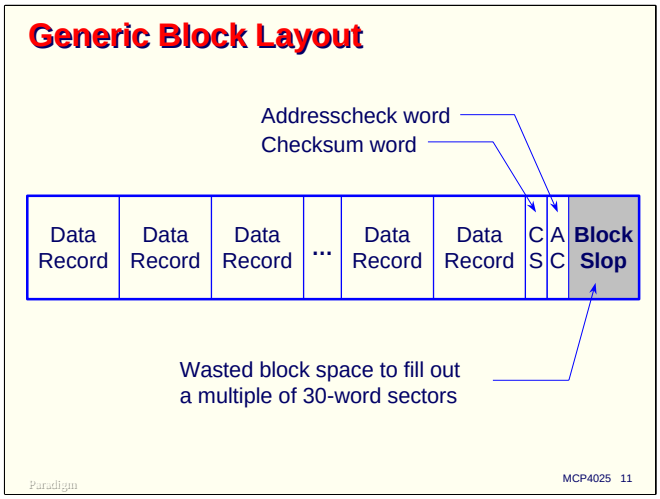
Checksum is a 48-bit signature value for the data in the block. Its primary purpose to to guard against accidental or intentional corruption of data within the block. DMSII computes the checksum and stores it in the block each time before writing the block to disk. It re-computes the checksum each time after reading the block and compares the computed value to the one stored in the block. If the values do not match, DMSII retries the I/O. If after a number of retries the values still do not match, the block is considered to have an irrecoverable error and the read-error bit is set for that row of the file.

Checksum can be specified in DASDL on a structure-by-structure basis for data sets, index sets, and the data base audit trail. It can be added to or deleted from existing data sets and index sets by way of reorganization.

Addresscheck stores the file-relative sector address of the block in the block itself. It guards against certain types of disk access errors that return the data for the wrong address on disk. DMSII stores the address word the first time the block is allocated in the file. Each time it reads the block, it compares the address in the block with the address it intended to read. If they are different, DMSII retries the I/O, and if after a number of retries the values still do not match, it declares the block to have an irrecoverable error and sets the read-error bit for that row of the file.

Addresscheck is a global DASDL option that can be set only for the entire data base. It cannot be added to an existing data base using reorganization, so it can be put into effect only for new data bases.

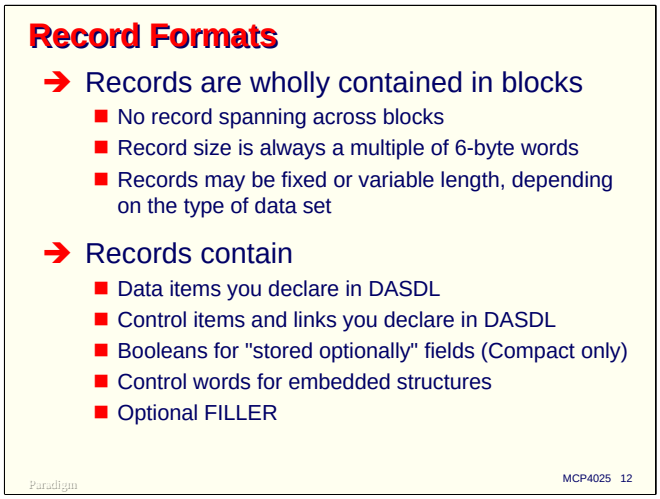
I strongly recommend that you use both of these optional control words whenever possible.



This diagram shows generically what a data block looks like in a DMSII file. It shows that the block contains a number of data records. Assuming that the Checksum and Addresscheck options are set in DASDL, their corresponding control words are at the end of all other data in the block.

If the particular type of DMSII structure requires additional control words, they generally appear at the beginning of the block, before any data records. We will see some examples of this later when we look at specific types of structures.

If the data records and control words together amount to a size that is not an exact multiple of 30 words, the remainder of the last disk sector for the block will be wasted space, or block slop. Minimizing this wasted space is one goal of good physical file design.



Records in DMSII structures must be wholly contained within their blocks. They cannot span blocks. Records are always allocated in multiples of 6-byte words and always start in on a word boundary within the block.

Most DMSII data sets support only fixed-length, fixed-format records, but two (Standard and Unordered) support variable-format records and one (Compact) supports variable-length records.

Records obviously contain the data items you declare for them in DASDL, but may contain other types of fields as well.

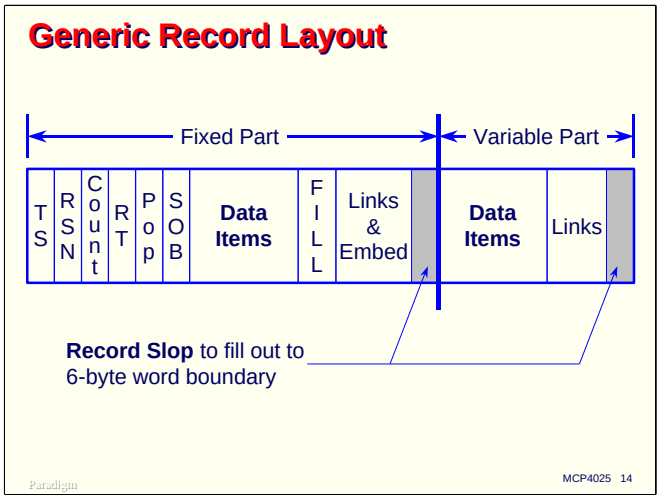
- Some types of DASDL declarations are classified as "control" items. Generally, these are items for which you cannot directly set their value. DMSII manages the value instead. These include count items, population items, aggregates, record types, and links.
- Compact data sets allow you to indicate that some items are "stored optionally." For such items DASDL allocates hidden Boolean fields in the record to indicate whether the item is really there or not.
- Embedded data sets and index sets may require a 12-digit (48-bit) control value in their master record. Embedded standard and compact data sets do not require this control value; all others do.
- You can declare a single FILLER item in DASDL to reserve space for future expansion of the record. This FILLER takes space in the record and is initially assigned a value of all one bits (COBOL HIGH-VALUE).

The different types of items are always allocated space in the physical record area in a predetermined order. This order is used regardless of the order in which you declare the items in DASDL.

Every data set has a fixed part of the record. Items in this part are allocated in the following order:

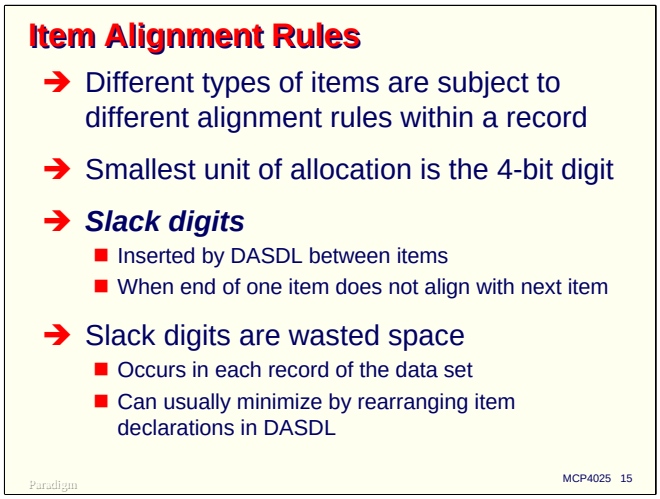
- If the EXTENDED option is set for the structure, the Transtamp word is first, followed by the Record Serial Number (RSN) word
- If the record is referenced by counted links, the count field is next.
- If this is a Standard or Unordered variable-format record, the record type item is next.
- If any population items are declared in DASDL, they will come next in the record.
- If this is a Compact data set with stored-optionally fields, the Boolean flags for those fields will come next.
- All non-control data items will be allocated next in the record in the order they were declared in DASDL.
- If a FILLER item was declared, it will follow the data items, regardless of where it was declared in DASDL.
- Finally, links and control fields for embedded structures will appear at the end of the record in the order they were declared in DASDL

For variable-format records, the variable part will follow the fixed part of the record, aligned on a word boundary. It will contain the data items in the order they were declared in DASDL, followed by any FILLER, followed by any links.



This diagram shows generically what a record looks like on disk. Probably no record will have all of these things in it—most records will have just data items, filler, and possibly the Transtamp and RSN words for XE structures.

Note that since records are always allocated on 6-byte word boundaries, it is also possible to have wasted space – ***record slop*** – at the end of both the fixed and variable parts. This wasted space will always be less than 6 bytes long. It is often possible to incorporate it into a FILLER item so it could potentially be used for record expansion.



In addition to appearing in a certain order, data set record items are subject to alignment rules that specify the address boundaries on which they are allocated. Some types of items can be allocated on 4-bit digit boundaries, the smallest unit of storage DMSII supports. Other types of items must be allocated on 8-bit byte or word boundaries.

If the end of one data item does not fall on the alignment boundary of the next item, DASDL must insert extra space between the items so the next one will obey its alignment rule. This extra space is sometimes referred to as ***slack digits***. The most common occurrence of this is when a number or field item ends on an odd digit boundary and is followed by an alpha or group item, which must be byte-aligned.

Slack digits are wasted space in the record. This waste will occur in each record of the data set. You can usually eliminate this wasted space (or at least minimize it) by simply rearranging the way the items are declared in the record.

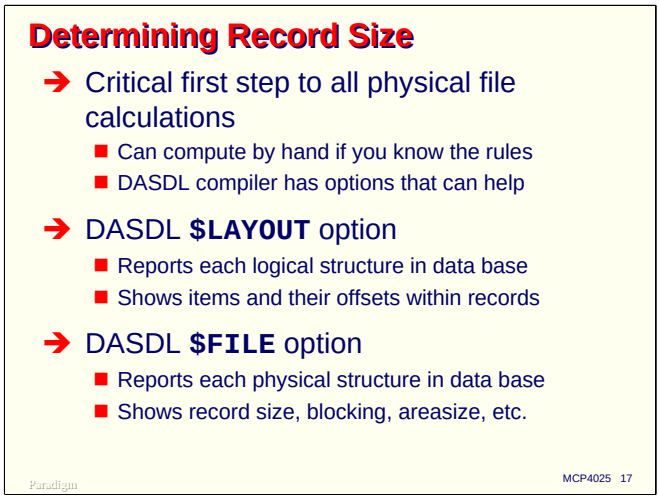
Alignment Rules, continued

Alignment	Entities
Digit	Number, Boolean, Field, Link, Embedded structures, Population, Count, Record Type, Stored-optionally Booleans
Byte	Alpha, Group, Real, Filler, First non-control data item in record
Word	Fixed part of record, Variable part of record, First link or embedded structure in fixed part, First link in embedded part

Paradigm

MCP4025 16

This table shows the DASDL alignment rules for various types of items. Note that in some cases it matters where the item is in the record more than what type of item it is.

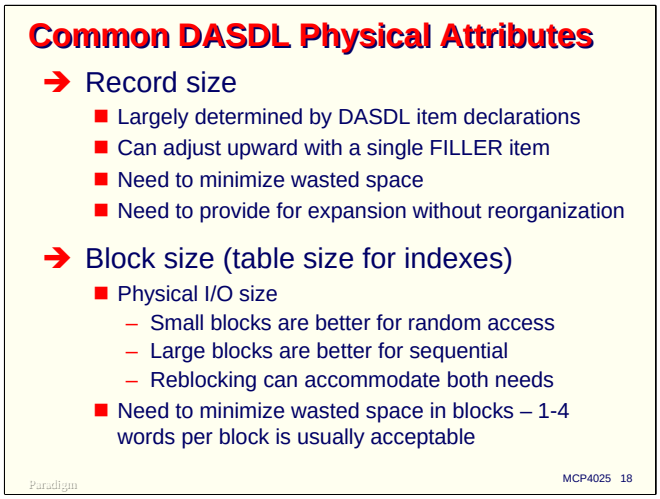


In order to determine physical file attributes for data base structures and evaluate them, we must first do some calculations. Essentially all such calculations are based on the size of the record for a particular structure. Therefore, determining the record size is always the first step.

Based on knowledge of the items declared in the record, their respective sizes, the required ordering of items in a record, and the alignment rules, it is usually not very difficult (although a bit tedious) to calculate the record size.

The DASDL compiler is the final authority on record size, however, and can generate two reports that will help you verify your calculations. In many cases, it may be easier simply to do a preliminary compile just to let DASDL tote up the sizes for you and generate these reports:

- Setting the **$LAYOUT** option generates a report for each logical structure in the data base showing the items, their sizes, and their relative offsets within their records. Adding the size and offset of the last item in the record yields the record's size.
- Setting the **$FILE** option generates a report for each physical structure in the data base. It shows the actual record size, block size, areasize, and so forth that DASDL determined from your input, after making any adjustments it deems necessary. This report is more useful for verifying your record, block, and area calculations after you have done them than as a preliminary design tool



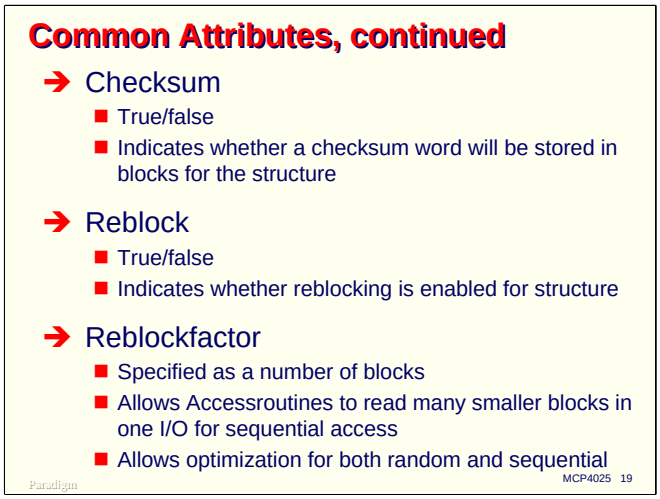
Most DMSII structures have a number of physical attributes in common. Before delving into the details of specific structures, we'll first do an overview of these common attributes.

The first of these is **record size**. In most cases the record size is entirely a function of the item declarations you make in DASDL. You can expand the defined size of the record and reserve room for expansion by including a single FILLER item. Records are subject to two kinds of wasted space: slack digits and the slop at the end of the record necessary to round the size to a multiple of six bytes. As mentioned above, slack digits can usually be minimized or eliminated by rearranging the item declarations to avoid misalignment of fields. The only thing that can be done with record slop is to absorb it into a FILLER item so it can be used for future record expansion.

The **block size** defines the physical unit of data that will be read to and written from disk. Generally speaking, small blocks are better for random access to the data, while the performance of sequential operations is enhanced by large blocks. DMSII lets you have it both ways, however, using a feature called *reblocking*, which is discussed next.

The major optimizations available through block size are:

- Keeping blocks relatively small to favor random access and effective use of cache memory.
- Minimizing wasted space due to block slop by choosing a blocking factor that is compatible with the record size.
- For structures that will be stored on VSS-2 devices, assigning block sizes in two-sector (60-word, 360-byte) increments. It is not a bad idea to use the VSS-2 convention in all cases, regardless of the type of disk device.



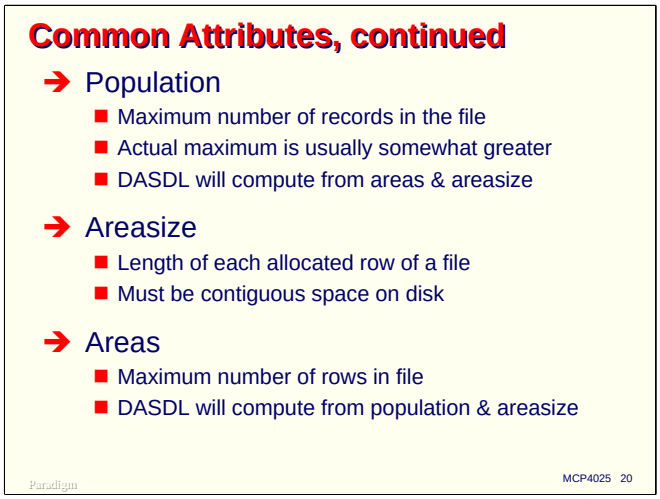
The **Checksum** attribute can be specified on a structure-by-structure level and can be added or deleted through reorganization. It causes an additional word to be stored in each block, so calculations to determine the blocking factor must take this extra word into account. In addition to the space overhead, using this attribute generates extra processor overhead when reading and writing blocks.

The **Reblock** attribute determines whether DMSII will be able to perform reblocking for the structure at run time. This can be set in the DASDL defaults and individually for each structure. It can also be changed at run time through the Visible DBS interface.

The **Reblockfactor** attribute indicates the extent of reblocking that DMSII will do. When the Accessroutines detects a sequential pattern of I/Os for a structure which is enabled for reblocking, it begins reading multiple blocks in a single I/O. The Reblockfactor specifies the maximum number of blocks that can be read in one operation. For long sequential processes (such as reading an entire data set directly) reblocking can improve performance by a factor of almost the Reblockfactor, as it eliminates the overhead of multiple I/O operations and the disk latency associated with each one. Like Reblock, the Reblockfactor can be specified in the DASDL defaults, specified individually for each structure, and modified at run time through the Visible DBS.

Note that reblocking is subject to the following additional restrictions:

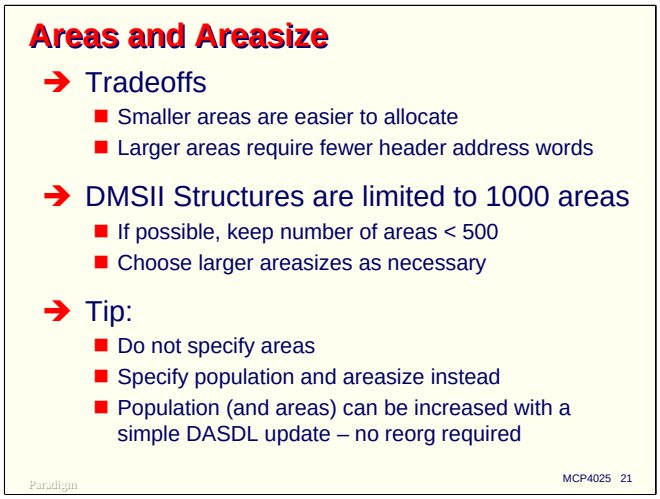
- The structure must have a serial buffers setting of two or higher.
- Reblocking is effective only for Direct and Standard data sets.



The **Population** attribute indicates the maximum number of records that can be stored in a data set or spanned by an index set. Normally you specify this for data sets but not for spanning index sets. It may be useful to specify it for a subset if the subset spans only a small portion of the data set records. The actual maximum population is usually somewhat higher than that specified in DASDL, as DMSII will make full use of all areas in the file, and DASDL usually adds some extra areas to accommodate available space tables. If you do not specify a population for a data set, DASDL will compute a population from the Areas and Areasize attributes. If you do not specify Areas or Areasize, DASDL uses a default of 10,000 records.

The **Areasize** attribute specifies the length of each allocated area (row) of the disk file. This area must occupy a contiguous sequence of sectors on the disk. It can be specified in terms of records, blocks, or sectors.

The **Areas** attribute specifies the maximum number of areas the file can contain. DMSII structures are limited to a maximum of 1,000 areas. If you do not specify this attribute, DASDL will compute it from Population and Areasize. As mentioned above, DASDL usually tacks a few additional areas onto the ones you specify to accommodate available space tables and other control blocks in the structure.

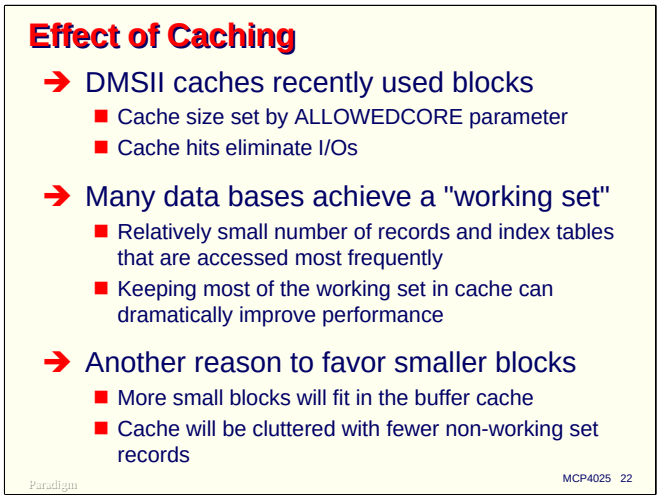


There are some tradeoffs with areas and areasize. Since an area must occupy a contiguous sequence of sectors, smaller areas are easier to allocate than larger ones. Using smaller areas, however, means that the file will have more areas.

There are two problems with having more areas. The first is that the disk header grows by one word for each row in the file. Disk headers are memory resident while the file is open, so this increases the memory requirements for the data base. While not the cause for concern that it was 20 years ago, a large data base could easily account for a couple of hundred thousand words of disk headers. To paraphrase the late Senator Everett Dirksen, a couple of hundred thousand words here, a couple of hundred thousand words there, and pretty soon you're talking about real memory.

The second problem is that DMSII structures can have at most 1,000 areas. This places a limit on how large the file can grow before you have to increase the areasize anyway and do a reorganization. It's a good idea to keep your initial design below 500 areas so the file will have some room to grow.

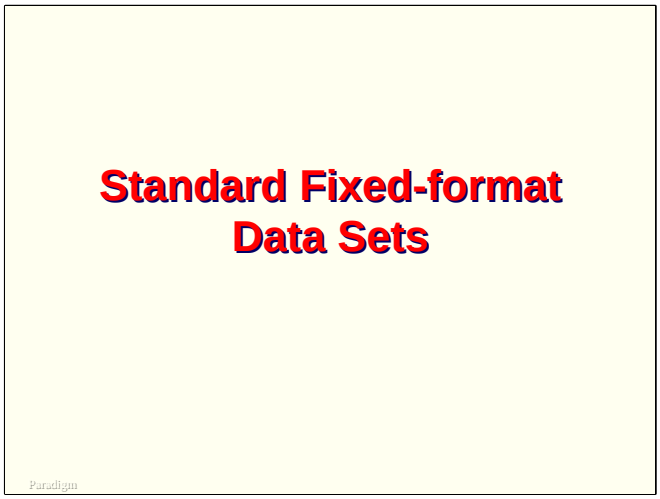
Here's a tip: I almost never specify Areas and Areasize. Instead I specify Population and Areasize, letting DASDL compute the number of areas for me. If the file has to grow, I simply change Population and let DASDL recompute the number of areas. Simply adding areas to an existing structure requires only a DASDL update, not a reorganization.



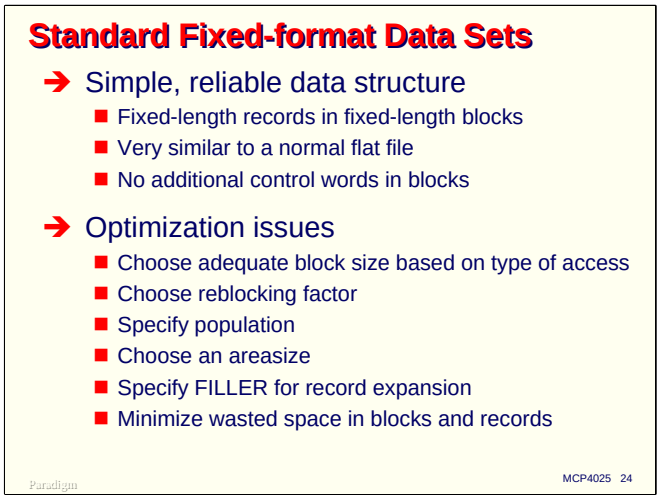
Another tradeoff concerns caching. DMSII caches blocks in a most-recently-used buffer pool that is controlled by the ALLOWEDCORE parameter. The Accessroutines always look in this pool for memory-resident blocks before initiating a physical I/O for data. The MCP has an optional global disk cache, and some disk subsystems also provide caching.

Small block sizes (and table sizes, for index structures) promote more effective use of cache memory. The reason for this is that with smaller blocks, there are fewer records loaded into the cache that are unrelated to recent retrievals. Since many data bases operate with a "working set" of active records, this means that the cache is more likely to hold records that will be requested again in the near term. Cache hits are hundreds or thousands of times more efficient than physical I/Os, so making effective use of the cache memory can substantially improve performance.

For this reason I am a big fan of relatively small blocks and table sizes. I try to pick the smallest block or table size of the choices I consider viable.



Now let us take a detailed look at physical file design issues for the most common type of data set, Standard fixed-format.

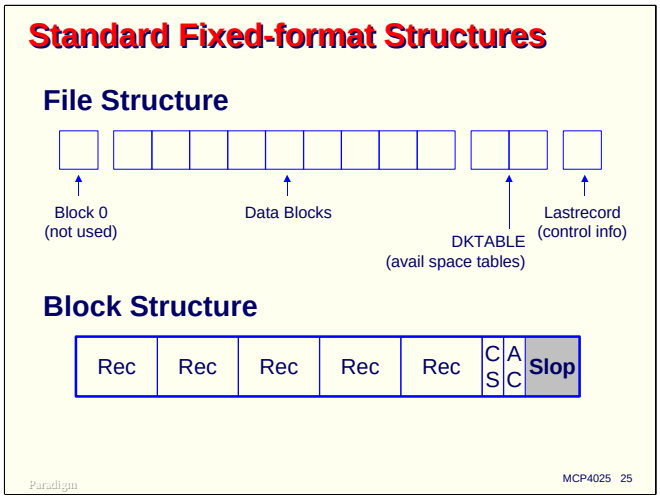


Standard fixed-format data sets are a simple, reliable data structure. They are very similar to ordinary flat files. There are no control words in blocks for this type of data set, other than the optional Checksum and Addresscheck words.

Optimization of Standard fixed-format data sets is fairly easy. The record size and population are usually dictated by application requirements, so you need to choose a block size (or blocking factor) and an areasize for the file. You may also want to specify reblocking for the file, although in most cases I set a Reblockfactor of eight in the DASDL defaults and just leave it at that.

One other thing I try to do with data set blocks is minimize the wasted space due to block slop. You can do this partly by playing with the blocking factor, but often this will minimize the waste only so much, especially if you are trying to keep the block sizes relatively small. In those cases, I try to divide the wasted space by the blocking factor and distribute it to each record as extra filler. Initially it's still wasted space, but it's waste that can potentially be used in the future without requiring a reorganization.

In addition to distributing block slop among the records as filler, I try to think about how the structure might grow in the future, both in terms of new records and in terms of new fields within the records. If the requirements are not very settled, or I suspect that new requirements will be coming along fairly soon, I normally include some additional filler in anticipation of that growth. As we will see, you can play with the filler size and blocking factor to get a balance of both good disk space utilization and provide for growth within the record.



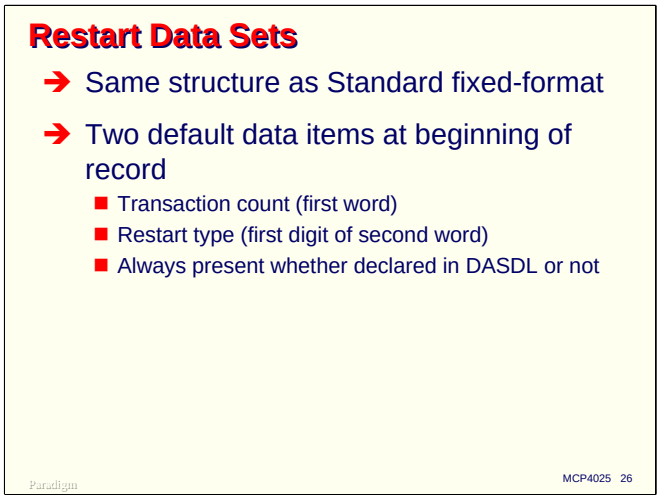
This diagram illustrates the physical structures for Standard fixed-format data sets.

Most of the file looks like an ordinary flat file you would create with MCP Logical I/O. Data records start with the second block in the file. The first block (block 0) is not used in this type of structure.

Following the data blocks, there may be a number of blocks that comprise the DKTABLE, or available space tables. These keep track of deleted records within the structure so they can be reused. DMSII will reuse deleted records in a data set before extending the EOF of the file.

The last *sector* of the file (located by the file's LASTRECORD attribute) contains control information used by DMSII.

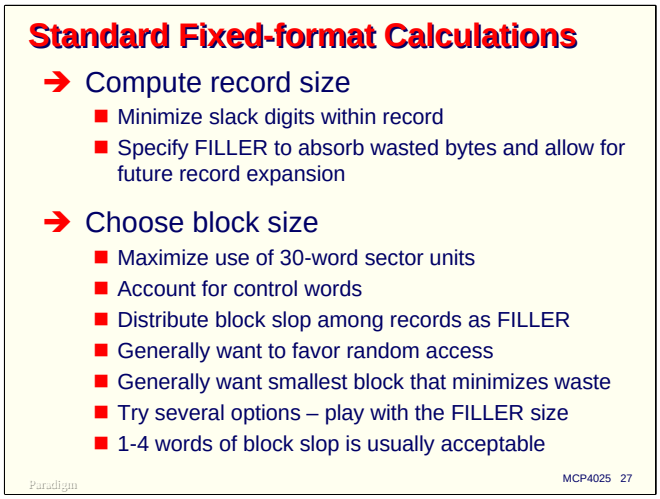
The blocks in the file have exactly the same structure as the generic block format we examined earlier. Data records are simply placed in the block after one another, optionally followed by Checksum and Addresscheck words. There is no other control information. Deleted records are identified by having NULL values in their required fields.



Restart data sets are closely related to Standard fixed-format data sets and use the same physical file and block structure. The only difference between a Restart data set record and one for a Standard fixed-format data set is the presence of two control items in the beginning of the record:

- The first word contains a binary transaction count that is incremented by DMSII for each transaction performed by the program.
- The first digit (four bits) of the second word contains a restart type code, which identifies the type of restart record. The codes for this field are defined in Appendix C of the DASDL manual.

All other physical characteristics of Restart data sets are identical to Standard fixed-format data sets.



When designing Standard fixed-format structures, there are a number of calculations to perform.

The first is to compute the record size. You need to take into account any slack digits in the record and any record slop necessary to round the size up to a multiple of six bytes.

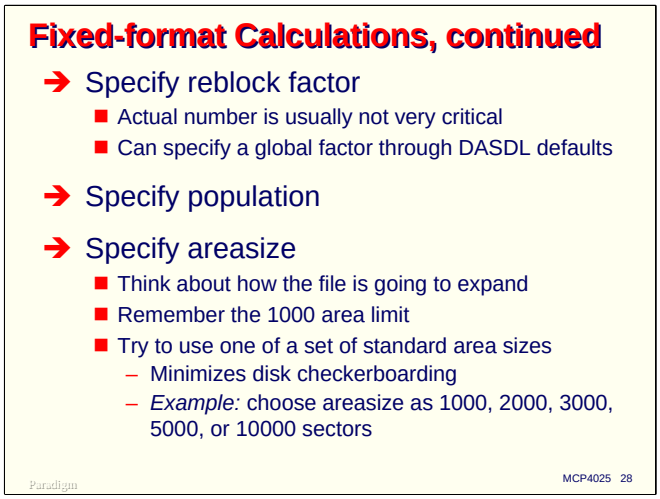
The second is to choose a block size or blocking factor. Since the block on disk will take a multiple of 30-word sectors, you need to take that into account in order to minimize wasted space. You also need to account for the Checksum and Addresscheck words if you have them configured. If the number of words of block slop is greater than or equal to the blocking factor, you should consider distributing this wasted space among the records as filler.

Generally speaking, you should choose smaller block sizes for data sets. This will favor random access and help make most effective use of cache memory. Making the block too small can result in more wasted space, so there is a tradeoff to be considered.

It is usually better to use DMSII's reblocking capability to enhance sequential access rather than define large block sizes. Remember that generally you can access data set records sequentially only when you read the data set directly (i.e., FIND NEXT <data set name>), not when you read sequentially through an index to the data set.

Since you can adjust record sizes by adding or subtracting filler, you can play with different blocking factors and filler sizes to get a good balance of block size, wasted space, and room for record expansion. We will look at a tool shortly that can help you do this.

I am usually satisfied if I can get the block slop down to four words or less. Often I can make it disappear entirely by adjusting the filler by only a small amount.

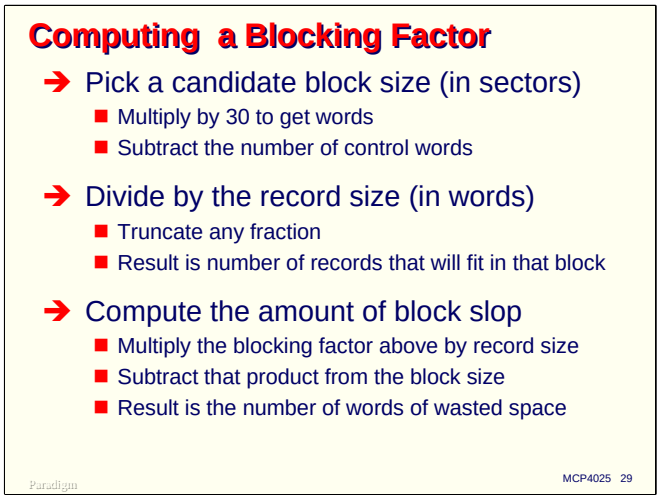


You will probably want to specify a reblock factor for every data set. Unless you do a lot of sequential passes through your data sets, you can probably get by with specifying a common Reblockfactor in the DASDL defaults and let that apply to each data set. If necessary, you can override the default for individual data sets. In the absence of a special need for reblocking, I usually specify a factor of eight.

The population of the data set will probably be dictated by application requirements more than anything else. You probably have a good idea of what the "working size" of the file should be. I usually base my calculations on a "design size" that is somewhat larger, to allow a reasonable amount of room for growth.

Finally, you should specify an areasize for the structure. I try to take into consideration two things when deciding on an areasize:

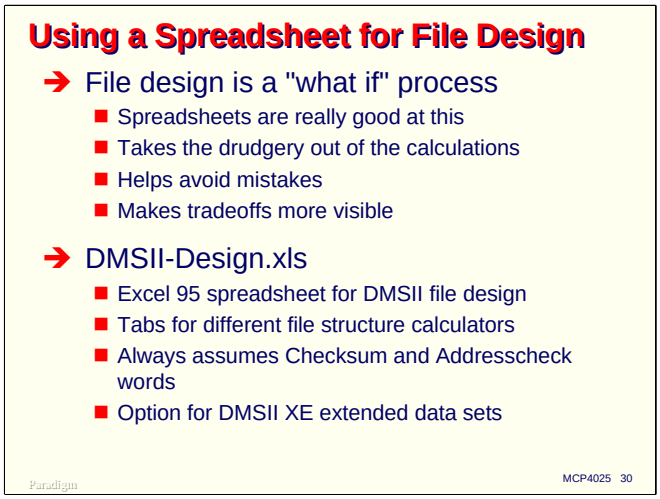
- How is the population of this file going to grow? It's easy to add areas to a file (and if you specify the POPULATIONINCR attribute in DASDL, it's even automatic), but there is that pesky limit of 1000 areas per DMSII file. I try to choose an areasize that will result in at most a few hundred areas.
- The method the MCP uses to manage disk space is very efficient, but subject to the phenomenon of *disk checkerboarding*, where many small areas of available space are distributed throughout the disk, making allocation of relatively large areas difficult. You can help minimize the effect of checkerboarding by assigning areasizes in multiples of a common unit. I normally choose 1,000 sectors as the common unit and try to design areasizes that are as close as possible to 1,000, 2,000, 3,000, 5,000, or 10,000 sectors. The actual unit you choose is not all that important, but the fact that your areasizes are multiples of that unit is. I usually apply this technique only to medium-to-large data sets. Most data bases have a number of relatively small structures used for code tables, dictionaries, etc., and for those I will typically use a much smaller areasize.



Here is the method I use to compute a blocking factor for a data set:

- First, pick a candidate block size in sectors (two-sector units for VSS-2). Multiply this by 30 to get a block size in words. Subtract the number of control words in the block to get the effective size.
- Next, divide this effective block size by the size of the records (in words, including any filler) and truncate any resulting fraction. The quotient is the number of records that will fit in a block of that size.
- Now compute the amount of block slop. Multiply the blocking factor just obtained by the record size (again in words), then subtract that product from the effective block size. The difference is the amount of block slop.

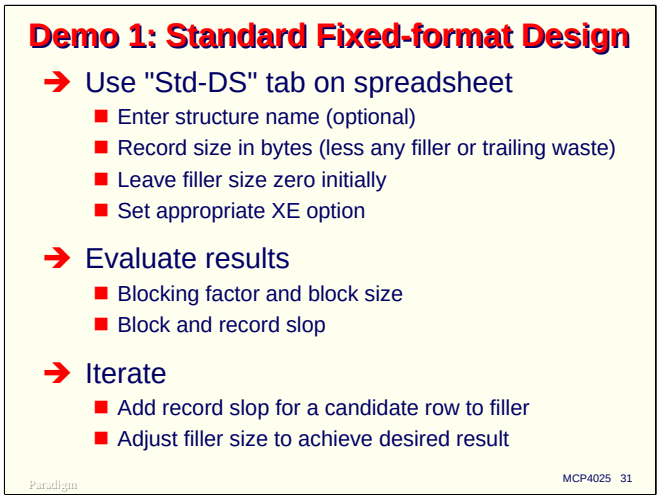
If you do this for a number of candidate block sizes, you can see how the amount of block slop changes with different blocking factors. You can also play with the record size (by adjusting the filler size) to minimize wasted space. These calculations are not difficult, but they are tedious, so it would help to use a tool for this.



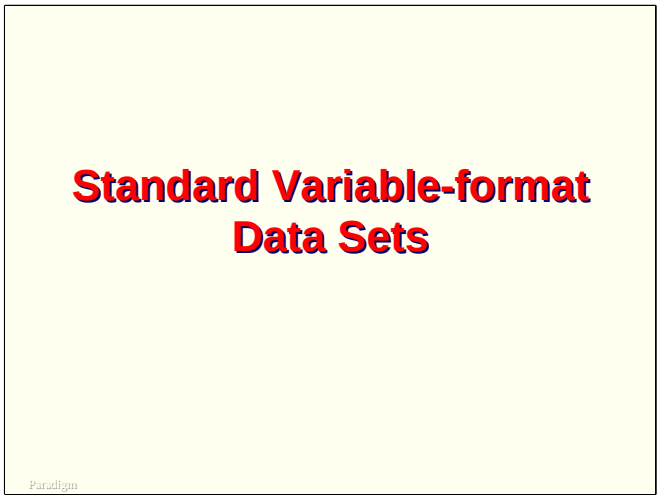
As just shown, even the simple calculations for Standard fixed-format data sets can involve iteration over a number of candidate choices and "what if" manipulation of the variables involved. There is a great tool for this type of work—the common spreadsheet.

It is quite easy to set up a spreadsheet to do these types of calculations, and to do them for many candidate cases at once. This takes the drudgery out of the calculations, helps prevent mistakes, and makes the tradeoffs more readily visible. Properly done, the spreadsheet will present you with a series of choices, from which you can just evaluate the tradeoffs and pick one.

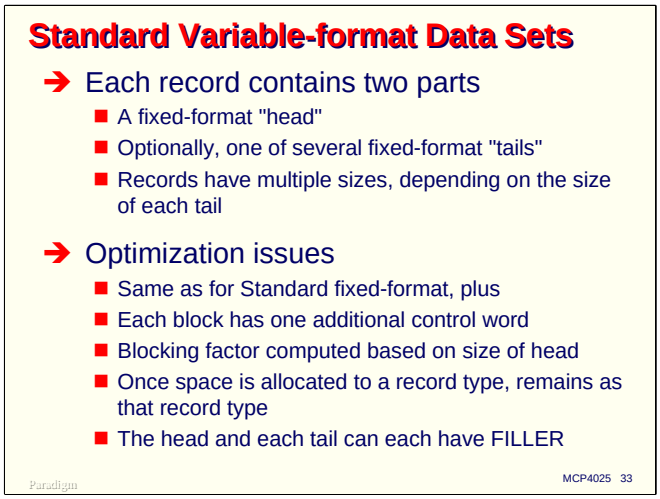
I have such a spreadsheet that I have used for years in designing DMSII data base structures. It has evolved over time to meet the needs of the moment and and does not attempt to be a universal tool. I'll use it throughout the rest of this presentation to illustrate some design case studies and evaluation of design tradeoffs.



For our first use of this tool, let's look at the design of a typical Standard fixed-format data set. We will fill in the parameters on the tab for Standard fixed-format calculations, evaluate the results, and do a few iterations to refine our choices.



Next, we will discuss a variation on the Standard fixed-format data set, the Standard variable-format data set.

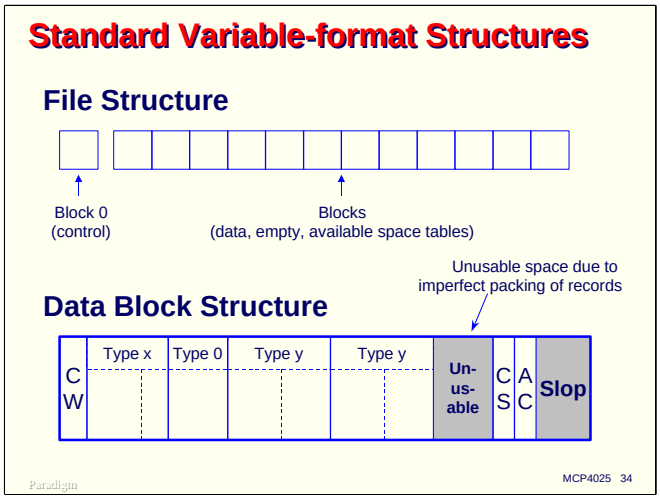


Standard variable-format data sets differ from fixed-format in that each record can have two parts. The first, or fixed part looks just like a Standard fixed-format record. The second part is optional. If present it is one of a set of predefined fixed-format records. A special type of item in the first part (the record type) indicates which type of second part, if any, is present in the record.

The first part of the record is often called the "head" of the record. The second part is called the "tail." A record always has a head. It may have at most one tail. Because the tails can be of various sizes, the record size varies on a record-by-record basis.

The design and optimization issues for Standard variable-format are much the same as for fixed-format, with the following additional considerations:

- Each block in the file has an additional control word in the front of the block.
- Since the records are of varying sizes, "blocking factor" does not have the same meaning as for fixed-format data sets.
- When DASDL computes blocking for a variable-format data set, it uses only the size of the head. This means that the population needs to be overstated, since fewer records will fit in the file than DASDL's internal calculations would otherwise indicate.
- Once space in a block is allocated to a certain type of record, that space can be reused only for records of the same type. In effect, DMSII maintains a separate available space list for each record type.
- Both the head and tail portions of the record can have filler. The size of the filler can differ between the head and tails and among the tails themselves.

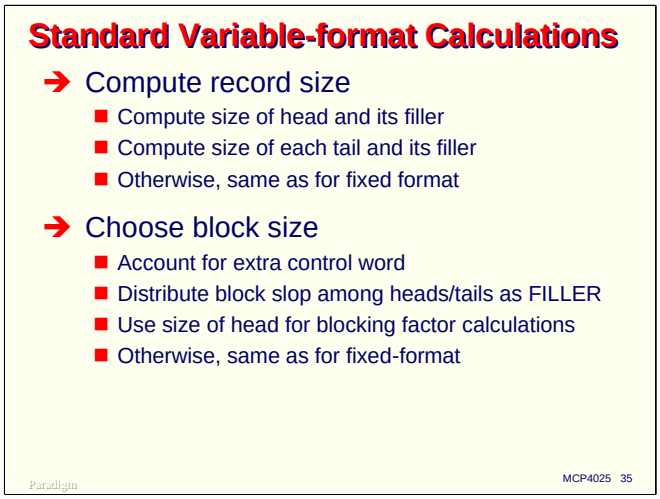


The physical structure of Standard variable-format data sets is somewhat more complex than for fixed-format, due to the necessity of managing variable record sizes.

The first block of the file (block 0) contains control information used internally by DMSII.

The remainder of the file consists of a series of uniform blocks. Some blocks hold data records, some blocks contain available space tables, and some blocks are empty and available for reuse.

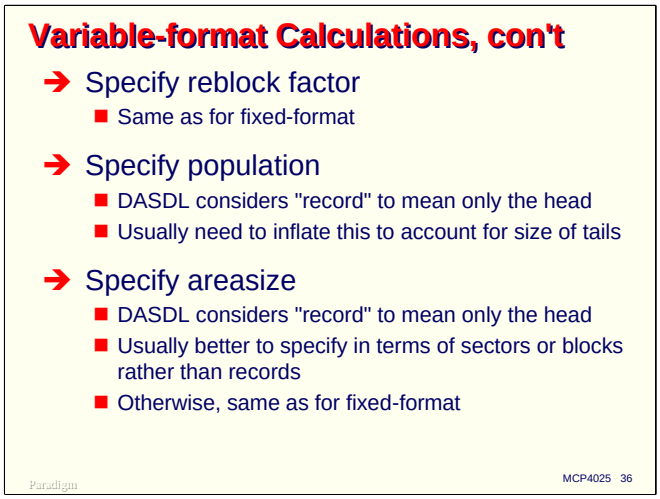
Within data blocks, the first word is a control word, followed by data records. Each record consists of at least the head portion. Since the records are variable length, it may not be possible to completely fill the block, and there will typically be unusable space in addition to any block slop.



Performing calculations for Standard variable-format data sets is complicated by the existence of records with different sizes. You should compute the size of the head portion and its filler as for Standard fixed-format records. Similarly compute the size of each tail type.

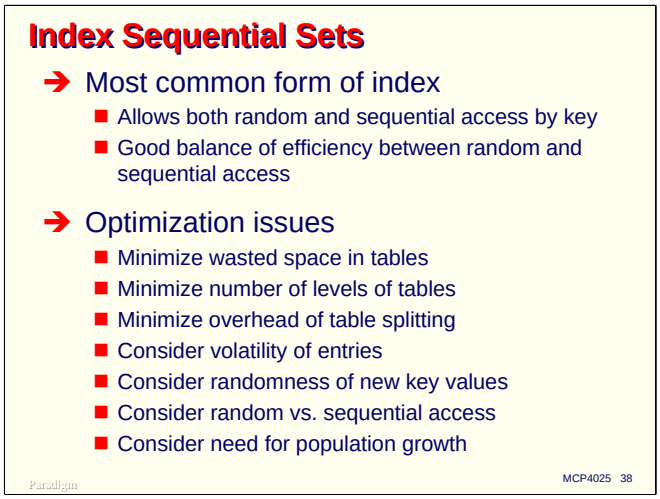
Choosing a block size for the structure requires some understanding of how the various record types will be distributed in the file. Since DASDL uses only the size of the head portion in computing a blocking factor, you should too. You then need to take into account that the record size from DASDL's point of view is smaller than it should be (since it does not include the size of any tails), and adjust the population upward accordingly.

Controlling wasted space in blocks with this type of structure is difficult, since you cannot usually predict which combinations of record types will exist in any given block, and therefore cannot predict how imperfect the record packing will be.



The rest of the calculations and attribute specifications are similar to those for Standard fixed-format, as long as you keep in mind that for sizing purposes, DASDL considers a record to consist only of the head portion. For this reason, you may wish to express block sizes and areasizes in terms of sectors rather than records.

Having discussed the three most common types of data sets, we now turn to the most common type of index structure, the Index Sequential set.



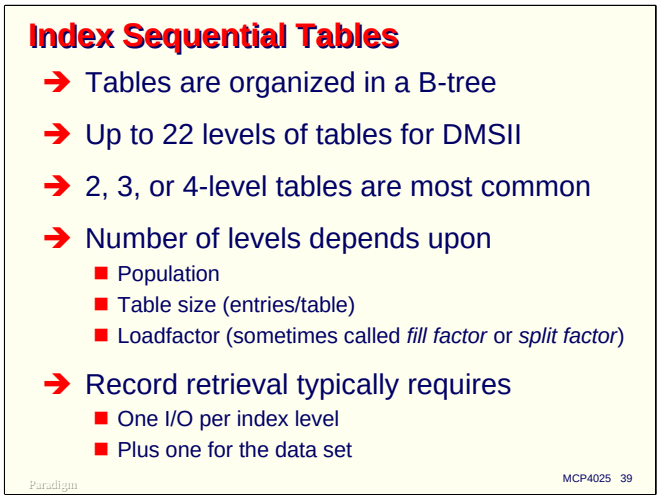
Index Sequential is the most common form of index structure for a good reason. It allows both random access and retrieval of records in sequence by key values. In addition, it does this with reasonable efficiency for both types of access.

Designing index sequential structures is more demanding than for the data sets we have previously discussed, partly because the data structure is more complex, and partly because in most cases more I/Os will be directed at index sets than data sets.

As always, we would like to minimize the amount of wasted space in blocks (called "tables" for index sets). More importantly, we want to minimize the number of I/Os necessary to retrieve a record (or to determine that the record does not exist). As we will see, this involves minimizing the number of levels of tables in the structure and (for update operations) minimizing the overhead of table splitting.

There are some other things to consider when designing an index sequential table:

- Volatility. This is a measure of how frequently records are added and deleted from the index. Higher volatility indexes generally need to be designed with looser space restrictions that those with lower volatility.
- Randomness of new key values. This is a measure of how insertions of new records are scattered throughout the key space. Indexes with a highly random distribution of new keys also need to be designed with looser space restrictions that those where new key values are inserted in a more sequential manner.
- Random vs. sequential access. Random access requires passing through all levels of the index structure, generally requiring one I/O per level, so it is very sensitive to the number of levels in the index. Sequential access through the index is much less sensitive to the number of levels.
- Need for population growth. The number of levels in the index is a function primarily of population and table (block) size. A table optimally designed for one population can become much less efficient if the population increases dramatically. You may wish to design the index for a much larger population than initially anticipated to accommodate growth and forestall the need to reorganize the index.

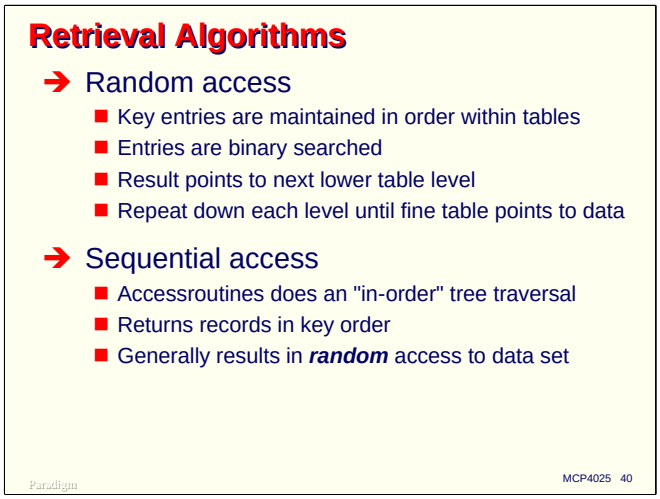


The tables (blocks) of an Index Sequential set are organized in a hierarchical structure called a B- (for balanced) tree. For DMSII, this tree can be as many as 22 levels deep, although typically trees are no more than four levels deep.

The number of levels in the tree is dependent on three factors:

- The population of the index. For a span set, this is the population of the corresponding data set. For a subset, this population may be much lower.
- The number of entries per table. This in turn depends on the size of the key fields, some options you declare in DASDL, and the size of the tables.
- The loadfactor. This is sometimes called the "fill factor" or "split factor", and is a measure of how full the tables will be after they fill up and split. We will discuss the loadfactor in more detail later.

Retrieving a data set record using an index sequential index generally requires one I/O per level in the index, plus an additional I/O to retrieve the data set record. In practice, the actual number of physical I/Os may be lower due to the presence of table and data set blocks in the ALLOWEDCORE cache.



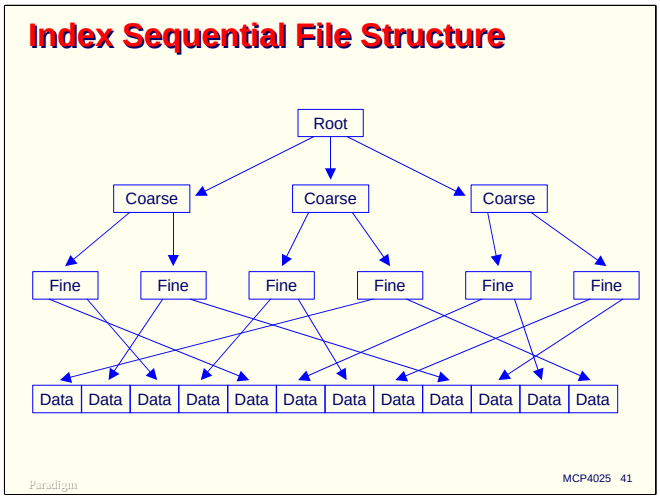
Index sequential can be used for both random and sequential retrieval of data set records. The algorithms used for these two modes are quite different.

Key entries within tables of the index are kept in ascending sequence. Each key entry basically consists of a set of key values and a pointer word. The pointer word addresses either a table at the next lower level in the tree, or if the table is at the lowest level (a "fine table" or "leaf node"), it addresses a record in the data set.

To find a record randomly, the top-level table in the index (called the "root" table) is binary searched. The address word from the highest entry with a key less than or equal to the requested key is used to read a table from the next ("coarse") level. That table is binary searched in the same way, and the address word from its entry used to find the table in the next lower level. When this process works its way down to the bottom (fine) level of the tree, either there is an entry that matches the requested key or there is not. If there is, its address word points to the desired record in the data set.

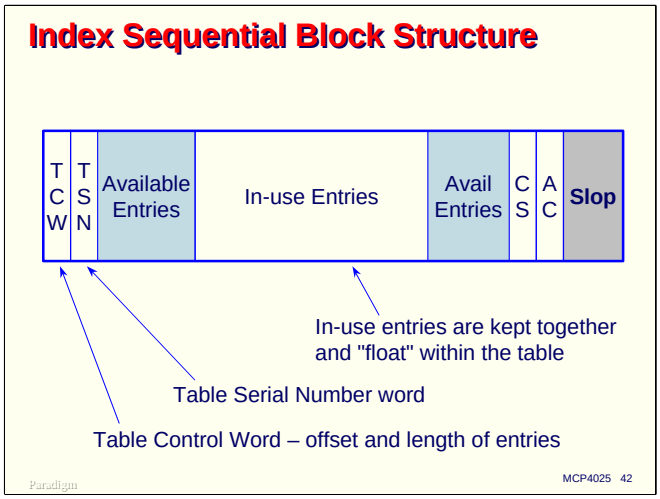
To retrieve records sequentially, the index is randomly searched as above for the starting point. Once that point is found, the Accessroutines does an "in-order" traversal of the tree to retrieve subsequent records in key order. Since there are typically many key entries per table, and the key entries within a table are in ascending order, this requires far fewer I/O operations to the index per data set record than random retrieval.

Note again, however, that sequential access of the index set usually implies ***random*** access of the data set, since the physical ordering of records in the data set may not (and probably does not) match the ordering of the index set keys. You can force a data set to have the same ordering as one of its indexes using reorganization, but as you add and delete records from it, the ordering will gradually become more random.



This diagram illustrates a typical three-level Index Sequential set and its associated data set. The top table is always called the "root" table and the bottom level of tables are always called the "fine tables." The "coarse tables" may occupy several levels or may not exist at all. In fact, it's possible to have an index that consists of only a root table (which is then also the fine table). This is what you get right after an index is first created, as with a DMUTILITY INITIALIZE operation.

How do you get more levels if only the root exists after the structure is initialized? The answer is through table splitting, which we will discuss shortly.

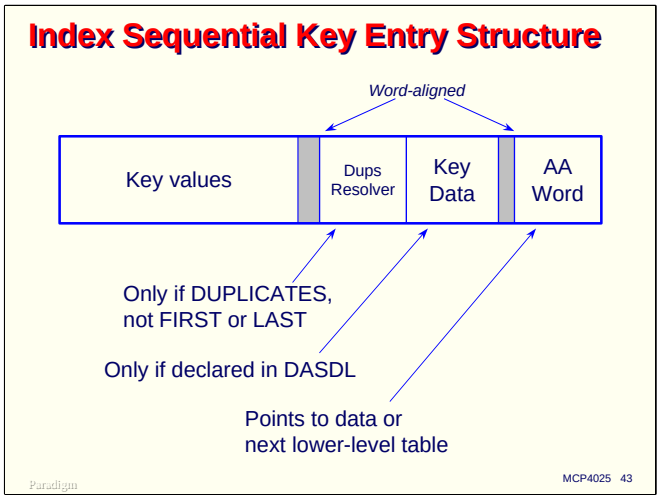


This diagram shows a typical table from an index sequential structure. There are two control words in the front of the block, the Table Control Word (TCW) and Table Serial Number (TSN). The format of these words is described in Appendix C of the DASDL manual.

Most tables at any one time are not completely full. Unlike a data set block that has some deleted records, the key entries in an index sequential table are kept together. There may be unused space on either side of this group of entries. The TCW keeps track of where the entries in the table start and how many of them there are. The area containing in-use entries may "float" within the table as entries are added and deleted.

Like all other blocks, an index sequential table may have Checksum and Addresscheck words, and may have some block slop at the end.

In addition to keeping track of the position of the in-use entries within the table, the TCW has a bit that indicates whether this table is a fine table. The TSN is used by the audit and recovery elements of DMSII.



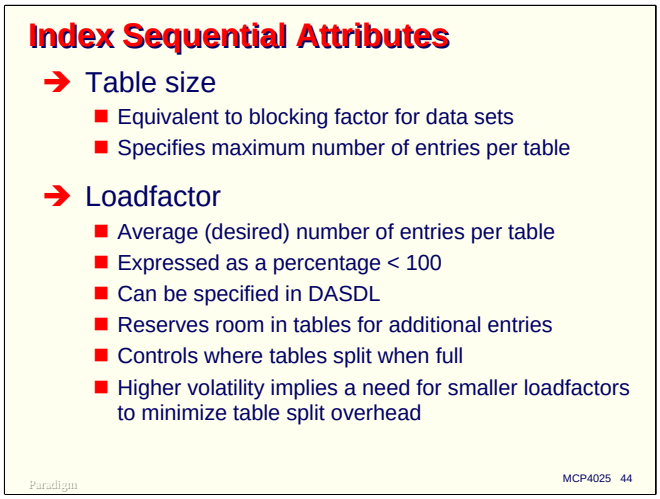
Within a table are key entries, which from a physical structure point of view are a lot like records in a data set.

At a minimum, each key entry consists of two parts: the key values and a pointer called the Absolute Address (AA) word. AA words point to a block in a structure, and optionally to a record within that block. For fine tables, this word points to a data set record; for all other tables it points to the next lower level table in the index.

Key entries optionally can have two additional parts:

- If the structure is declared in DASDL to allow DUPLICATES, but not DUPLICATES FIRST or DUPLICATES LAST, there is an additional word between the key values and the AA word called the "duplicates resolver."
- If key data is declared in DASDL for the index, it is stored in the key entry immediately before the AA word.

Fields are placed in the key values and key data portions of the entry using the same alignment rules as for items in data sets. This means that slack digits may exist in both of these areas. In addition, DMSII requires that the duplicates resolver word (if present) and the AA word be word-aligned, so some slop may exist after the key values and key data portions if they do not occupy a multiple of six bytes.



Index Sequential sets have two physical options we have not encountered before.

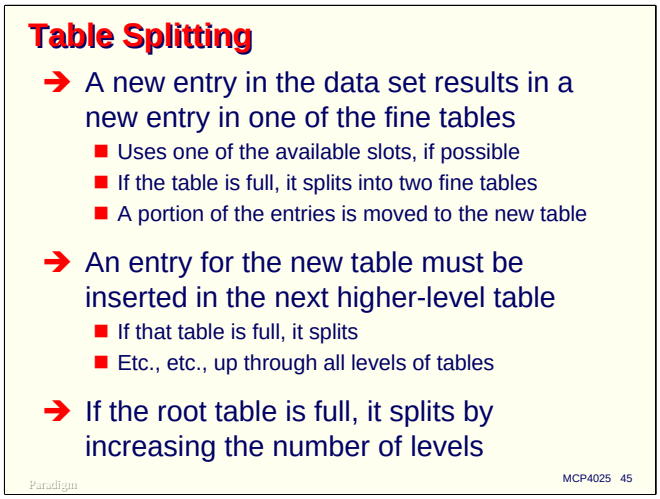
The **Table size** attribute is equivalent to a blocking factor for data sets. It indicates the size of the table (block) in terms of key entries.

The **Loadfactor** attribute is a number between 1 and 99. It specifies as a percentage the average number (or desired number) of key entries that will be present in tables. It can be specified in DASDL and defaults to a value of 66 percent.

The Loadfactor has two related uses:

- It essentially reserves room in table for additional entries. As records are added to a data set, the key entries for the corresponding indexes need to go somewhere. It is much more efficient if they can be inserted into an existing index table.
- When index tables become full, they split into two tables. We will discuss table splitting in more detail next. The loadfactor helps determine where a table will be split, how many entries will remain in the original table, and how many will be moved to a new table.

As we will discuss in more detail shortly, indexes with higher volatility should have lower loadfactors to minimize the frequency of table splitting.



As previously discussed, when new records are added to data sets, corresponding entries must be inserted into the spanning sets for the data set, and may need to be inserted in subsets for the structure. The Accessroutines searches the index for the appropriate fine table (based on maintaining key order) where the insertion is to take place. Most of the time there is room for the new entry, and it is simply inserted in sequence.

If the table is full, however, extra steps are required. The table is split, keeping a portion of the key entries in the old table and moving a portion to a new table that is allocated within the index structure. The new entry is inserted into either the old or new table, depending on its relation to the other key entries (see DASDL Appendix C for the algorithm).

Since this is occurring at the fine table level, the new table needs an entry in the next higher level, so an insertion is made there. If that higher-level table is full, it also splits, causing an insertion at the next higher level. This process continues until either an insertion can be made without splitting a table, or the root table is encountered. If the root is full, it splits into two new tables and ***the depth of the index tree increases by a level***.

This explains how an index sequential set grows levels. It starts out simply as a root table (which is also the only fine table). Once the root fills up, it splits and creates and two-level table. Now there can be a number of fine tables equal to the table size. It will take longer to allocate all of these fine tables and fill them up, but eventually the root will fill up with entries pointing to the fine tables, and it will spit again, creating a three-level table. Now there are potentially a number of fine tables equal to the square of the table size. This continues as more levels are added, with the number of fine tables and the number of data records the index is able to address growing by successive *powers* of the table size.

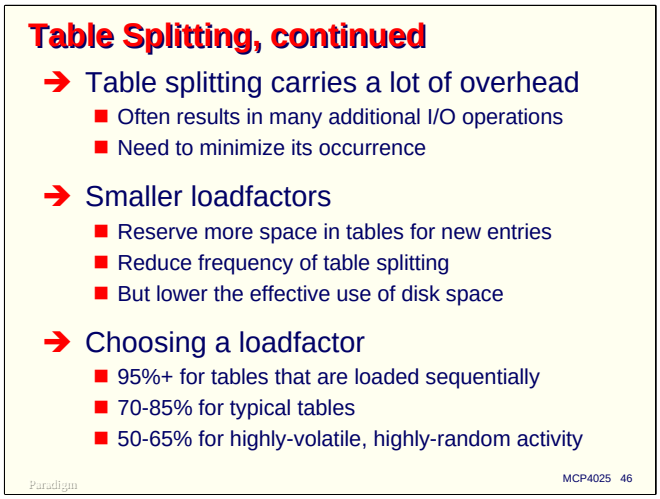


Table splitting is a very effective technique for keeping the index tree balanced, but it carries a lot of overhead, especially when the splits must recurse up through more than one level. Each split operation typically results in a number of I/O operations. These occur, of course, in the middle of a transaction, so we would like to keep this frequency of such splits at a minimum.

One way to reduce the frequency of table splitting is to use a lower loadfactor. This effectively reserves more space in the table for new entries when it splits, which means the tables will fill up more slowly.

There are two downsides to lower loadfactors:

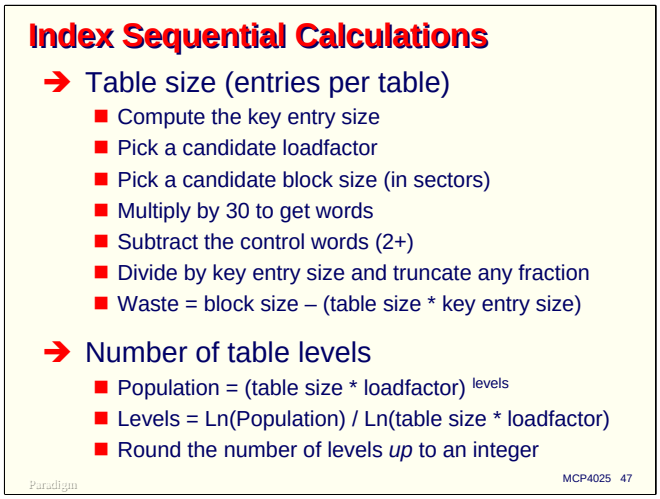
- Lower space utilization. Since we are intentionally reserving space for new key entries, the tables do not make good a use of either disk space or cache memory.
- Deeper tables. Since the tables hold fewer entries on average, a deeper tree is needed to span the data set records.

Here is another tradeoff: you can choose a lower loadfactor to get less frequent table split, or you can choose a higher loadfactor to get shallower tables and faster retrieval time. Since most data bases do more retrieval than update, the choice generally leans towards shallower tables, but this needs to be tempered with an understanding of how volatile and random the updates are.

For indexes that are loaded with sequentially (or nearly so) increasing key values, you can use a very high loadfactor—95% or more. All of the splits will occur at the end of the index. When loading an index sequentially, the split algorithm guarantees that the original table will be filled to its loadfactor, regardless of the value of the loadfactor.

For typical tables, loadfactors in the 70-85% range are usually a good compromise between compactness and split overhead.

For highly volatile tables, where there are lots of random insertions, you may want to consider loadfactors in the 50-65% range. Loadfactors lower than 50% are seldom worthwhile.



Here are the steps to calculating physical attributes for Index Sequential tables. The first step is to compute the table size.

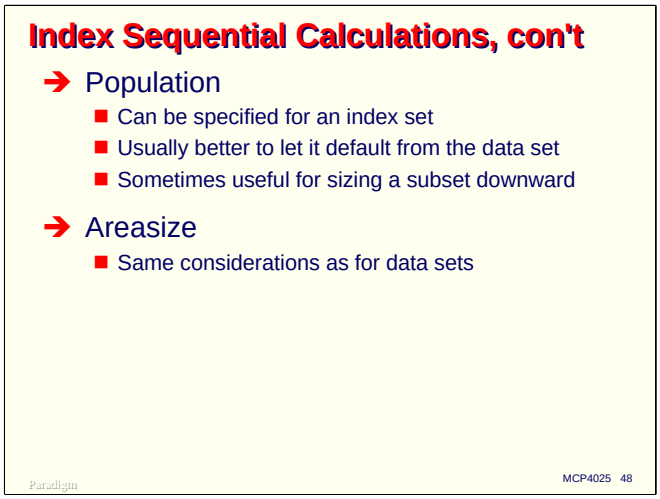
- Calculate the key entry size. You can do this manually using the alignment rules for data sets, or you can use the report from the DASDL **$LAYOUT** option. Don't forget to include the AA word and (if present) the duplicates resolver word and key data.
- Pick a candidate loadfactor.
- Pick a candidate block size for the table in sectors (or multiple of two sectors for VSS-2).
- Multiply this candidate block size by 30 to get words and subtract the control words (two for index sequential, plus Checksum and Addresscheck if you are using them) to get an effective block size.
- Divide this effective block size by the key entry size to get the table size.
- Compute the block slop for the table by subtracting the product of key entry size and table size from the effective block size.

Knowing the table size, loadfactor, and expected population of the index, you can now calculate the number of levels.

- The total population the index can address is the product of the loadfactor (expressed as a fraction < 1) and the table size, with this product raised to the power of the number of levels in the table. The problem is that we know the population and want to determine the number of levels.
- By taking the logarithm of each side of this equation, solving for the number of levels becomes easy. It's simply the log of the population divided by the log of the product of table size and loadfactor, as shown on the slide. Since the result of this calculation will usually be non-integral, round this number *up* to get the number of levels. You can use logarithms of any base in this calculation; natural (base-*e*) and base-10 are the most common choices.

By choosing different values for the block size and loadfactor, you can evaluate the number of levels, size of blocks, and amount of wasted space due to block slop. There are tradeoffs among these, naturally, and you have to decide which ones work best for you.

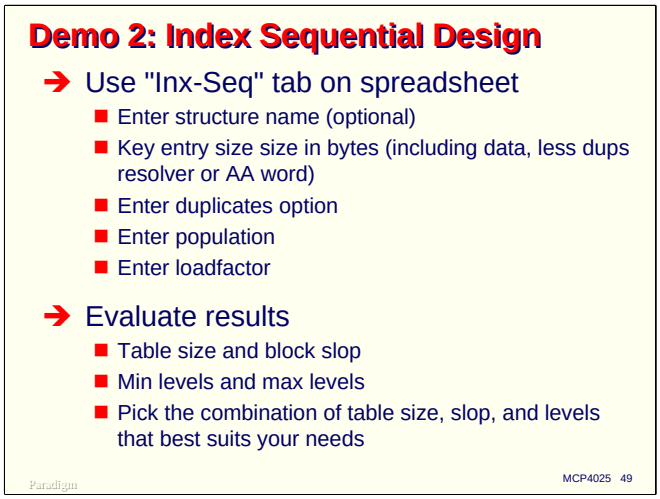
In most cases, you should be able to settle on a two- or three-level table that has reasonably-sized tables. Very large files (or files with very long keys) may require four levels. It is seldom necessary (let alone efficient) to have an index with more than four levels.



The only other attributes to consider for Index Sequential are Population and Areasize.

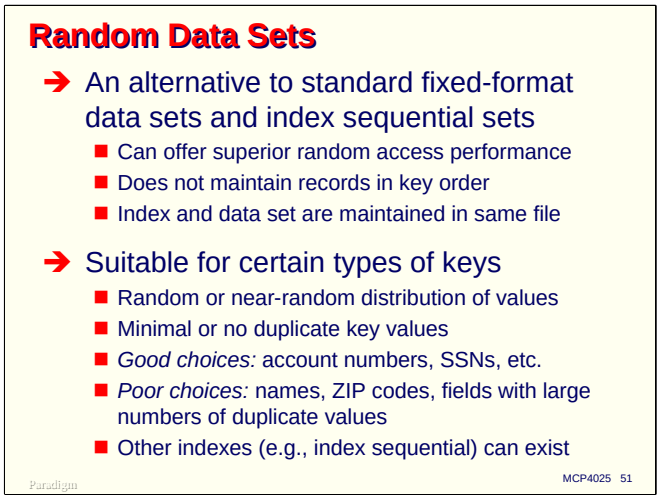
Normally you can let population for an index default from the population of the data set. If you have a subset that will span only a relatively small portion of the data set, you may want to lower the population for that index.

You can compute areasize for index tables the same way you do for data sets. It's a good idea to allocate areasizes from the same standard set of sizes discussed earlier, unless the index is very small.



The spreadsheet tool can also perform Index Sequential calculations. In this demo we will fill in the parameters for the sheet and try some different combinations to evaluate the tradeoffs for a typical index.

For our final topic, we will look at design issues and calculations for Random data sets.



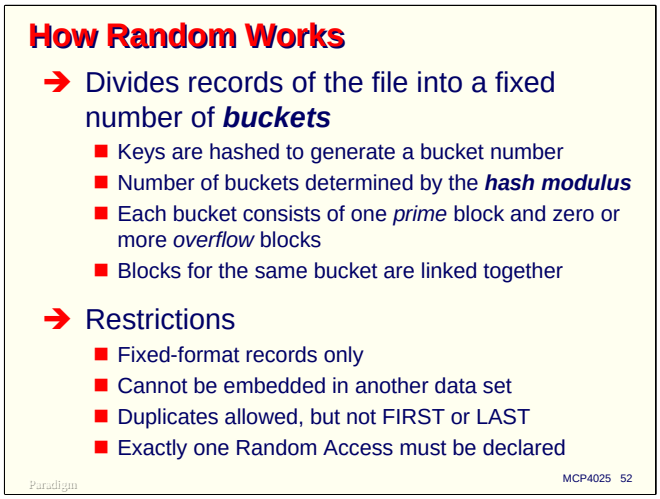
Random data sets and accesses are a viable alternative to Standard data sets and Index Sequential sets where you need primarily, well, random access to the data. Properly designed, they can offer much superior performance for random retrieval.

One of the costs for this superior random performance is that Random does not maintain an index in key order. The index has an order, of course, just not a very useful one.

Another characteristic of Random is that both the data and the index are maintained in one file—the data set. Index Sequential sets are maintained by DMSII in a separate file from the data set.

Random is suitable only for certain types of keys. It works best when the keys have a random, or near-random distribution of values, and there are no (or relatively few) duplicate key values. Good choices for random are usually account numbers, Social Security Numbers, and most other types of unique identifiers. Poor choices are fields with lots of duplicates or poorly-distributed values, including names, ZIP codes, and fields that have only a small range of values.

Thus, Random is often a good choice for a key field (or set of fields) that has a unique value and is heavily used to retrieve records randomly. There can be only one Random access for a Random data set. You can define additional types of indexes for Random data sets, including Index Sequential, to give access to the records by other keys.



The basic idea behind Random is very simple—divide the file into a large number of small pieces, access just one of the pieces randomly, and then search just that piece sequentially for the desired record.

Each of the small pieces of the file is termed a ***bucket***. The method Random uses to divide the file into pieces is a hashing function. In DMSII, the hashing function has two parts:

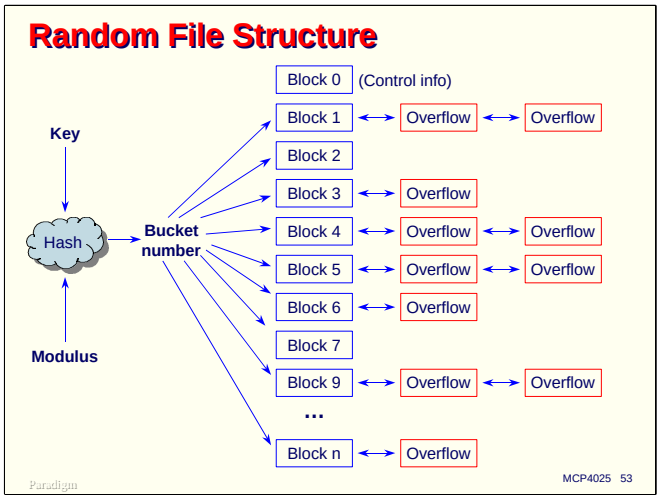
- A key folding algorithm reduces keys of any length to a 48-bit signature value known as the ***folded key***.
- A hashing algorithm performs a calculation on the folded key and divides the result by an integer that you specify, known as the ***modulus***. The remainder from this division is the ***bucket number***.

With this scheme there are a number of buckets equal to the value of the modulus. The first modulus number of blocks in the file (known as the ***prime blocks***) can be addressed randomly by their bucket number. If more records hash to the same bucket number than will fit in the prime block, an ***overflow block*** is allocated in the file after the prime block area and the prime block links to that overflow block. There can be as many overflow blocks linked from a prime block as necessary to hold the records which hash to the same value. The prime block and its overflow blocks together make up the bucket.

Random data sets come with a number of restrictions. They support only fixed-format records. The must be disjoint and cannot be embedded in another data set. The access can specify either DUPLICATES or NO DUPLICATES, but not DUPLICATES FIRST or LAST. Finally, exactly one Random access must be declared for each Random data set.

Appendix C of the DASDL manual describes the folded key and hashing algorithms in detail.

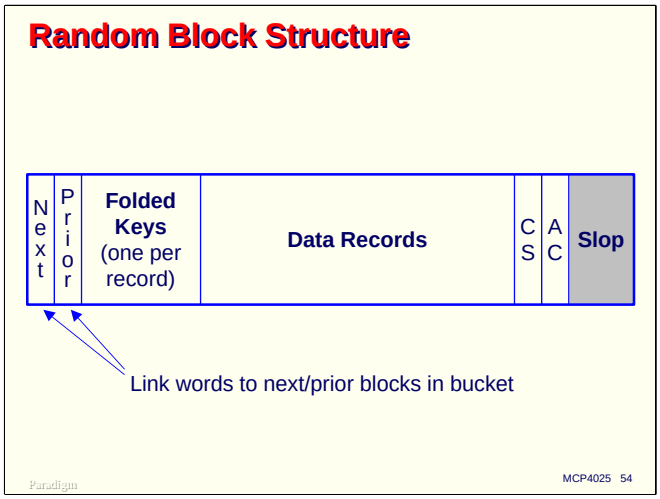
Note that when you initialize a Random structure with DMUTILITY, it must initialize all of the prime blocks in the data set file. If you have a large modulus defined, this process can take a noticeable amount of time.



This diagram illustrates the logical structure of a Random data set. The key and modulus values are parameters to the hashing function, which produces an integral bucket number having a value between 1 and the value of the modulus. That bucket number directly addresses one of the prime blocks in the data set. DMSII searches the prime block associated with the bucket number for the desired key. If found, the record is resident in that block and can be returned to the requestor. If not found, DMSII searches the chain of overflow blocks until the key is located or the end of the chain is reached.

Clearly, if all records reside in the prime blocks, every record can be retrieved with a single I/O. It is difficult to arrange this optimal case (at least not, as we shall see, without wasting a large amount of disk space), so there will usually be at least some overflow blocks.

Since each access to an overflow block generally requires an I/O operation, efficiency of retrieval for Random data sets is primarily a function of the average level of overflow within the buckets.

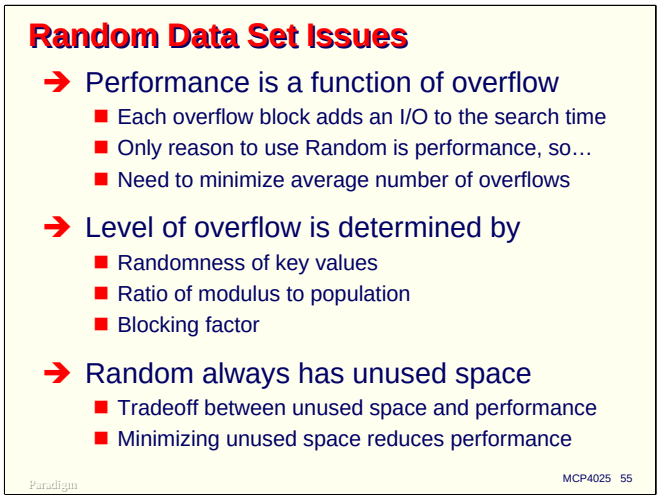


This next diagram shows the block structure within a Random data set. There are two control words at the front of the block, which implement the forward and back links for the bucket's overflow chain.

Following the link words is an area where folded keys for the records are stored. The size of this area is equal to the blocking factor, one word per record. The presence of the folded keys in the block allows DMSII to rapidly search the block for candidates. If the folded key value for the record being sought matches one of the folded keys in the block, then there is a high probability that the keys themselves will match. If the folded keys do not match, then it is guaranteed that the actual keys will not match. When it finds a folded key match, DMSII then compares the actual key values in the corresponding record.

Following the folded keys are the data records. The relative position of the data records matches the relative position of the folded keys, so DMSII knows which folded key goes with which data record.

At the end of the block are the optional Checksum and Addresscheck words, followed by any block slop to fill out to a disk sector boundary.



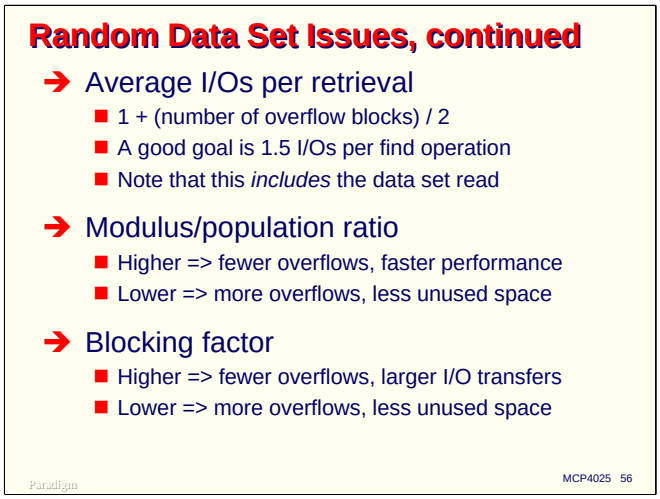
As mentioned earlier, performance of Random data sets for random access is primary a function of the level of overflow within the buckets, because each level of overflow requires an extra I/O to search its block. The only reason to use a Random data set is high-performance retrieval, so one of the main optimization goals in designing such a data set is minimizing the number of overflows.

The average level of overflow for a Random data set is a function of three factors:

- The randomness of the key values that will be stored in the data set. The more uniformly distributed the key values are, the more even the distribution of records across buckets will be, and the lower the level of overflow, all other things being equal. In the extreme case where all records have the same key value, Random will degenerate into a sequential search of the whole file, since all records will hash to the same bucket.
- The ratio of the modulus to the population of the data set. Dividing population by modulus yields the average number of records per bucket. If this average is smaller (i.e., the modulus is higher), there will be fewer records per bucket, and fewer overflows.
- The blocking factor for the data set. With higher blocking factors, more records fit in a block, so fewer blocks are needed to make up the bucket, and fewer of the blocks will be overflow blocks.

Obviously there is some interplay among these three factors, especially between modulus and blocking factor.

The big tradeoff in designing Random data sets is that, in general, you can't perfectly pack records into the buckets. Random data sets almost always have unoccupied record slots in their blocks. This space is not so much wasted as unused, since potentially a new record could be stored in an unoccupied slot. Trying to minimize this unused space almost always increases the level of overflow in the file. You can either spend disk space to get better performance, or spend I/Os to get more compact data storage. With the low cost of disk storage today, the choice in this tradeoff is generally clear.

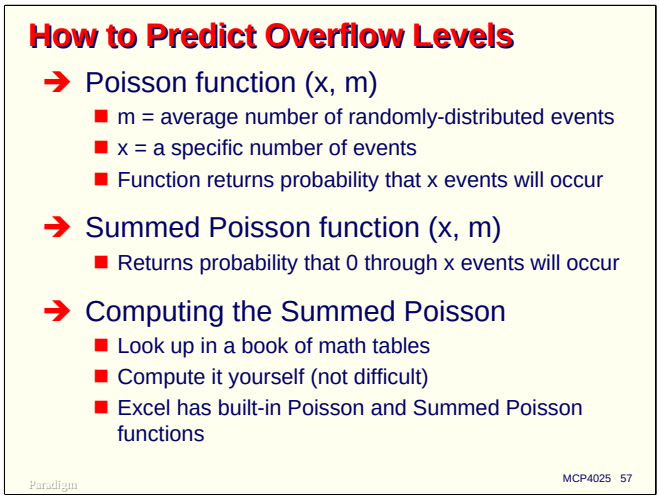


When searching for a record that exists, the average number of I/Os for retrieval from a Random data set will be one (for the prime block) plus half the number of overflow blocks. This is because, on average, only half the overflow blocks will need to be searched to find a record. When searching for a record that does not exist, the entire overflow chain will be searched, so the number of I/Os is equal to the number of blocks in the bucket.

A good goal for designing Random data sets is 1.5 I/Os per retrieval. This works out to an average overflow level of one (one prime block plus one overflow block per bucket). Note that this goal includes the read of the data set. Compare this to random retrieval using Index Sequential, where the average is typically 2-3 I/Os for the index set plus one for the data set. All of these averages, of course, discount any benefits that may accrue due to caching.

The main factors to adjust in a Random data set design are the modulus and blocking factor. A higher modulus/population ratio means fewer overflows and faster retrieval performance. A lower ratio means more overflows, slower performance, but less unused space in the file.

A higher blocking factor also means fewer overflows and faster performance, but the increase in performance is tempered somewhat by having larger I/O transfers and less effective use of cache memory. A lower blocking factor causes more overflows, but less unused space.

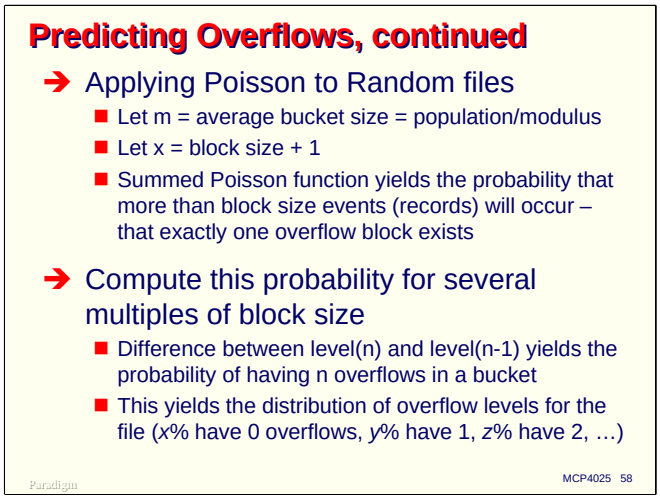


Since the primary goals of Random data set design are to minimize I/Os while achieving reasonable space utilization in the file, we need to be able to predict the level of overflow in the file. It is possible to do this using something called the Poisson distribution.

The Poisson function computes the probability that, given an average number of events which will take place randomly in a certain interval, exactly "x" events will take place in that interval. The function takes two parameters, the average and the specific number of events to be predicted.

There is a related function, the Summed Poisson, which computes the probability of *at least* "x" events occurring in an interval. This is the one we will actually be using.

You can compute the Poisson and Summed Poisson yourself or look up its values in a book of math tables. Microsoft Excel has both Poisson and Summed Poisson functions, which is how the spreadsheet tool I'm demonstrating computes them.

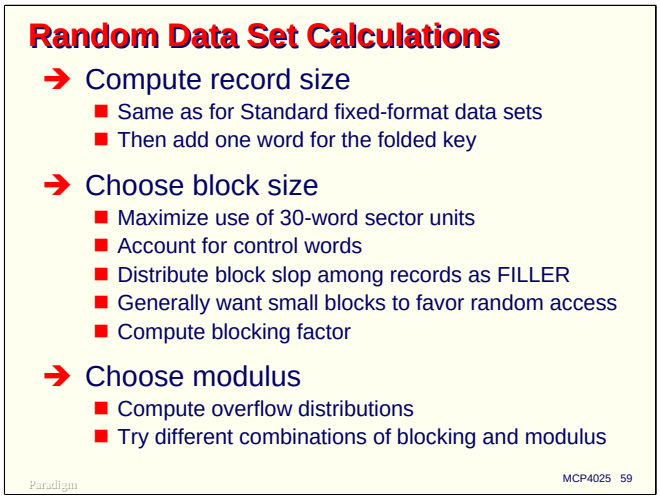


The Poisson function is normally used in operations research and queuing theory, so how does it apply to computing Random data set overflows?

The answer is that, while we are concerned with bucket population and not intervals, they are equivalent concepts. The average value fed to the Poisson function is the average bucket size, or population divided by modulus. We want to know the probability of the prime block overflowing, so "x" is the block size plus one.

Plugging these factors into the Summed Poisson function, we get the probability that there will be exactly one overflow. We need to know more than that, however, so we need to compute this probability for several multiples of the block size. Subtracting the probabilities of adjacent multiples, we get a distribution for the probability of overflow at each level.

This distribution will tell us that *x*% of the records do not overflow their prime block, *y*% will have one overflow block, *z*% will have two overflow blocks, etc. From that, we can compute the average number of I/Os per retrieval and (because the distribution tells us the number of blocks required) compute the amount of space the data will require. This allows us to evaluate the effectiveness of our choices for blocking factor and modulus, along with the tradeoff on performance vs. disk space.

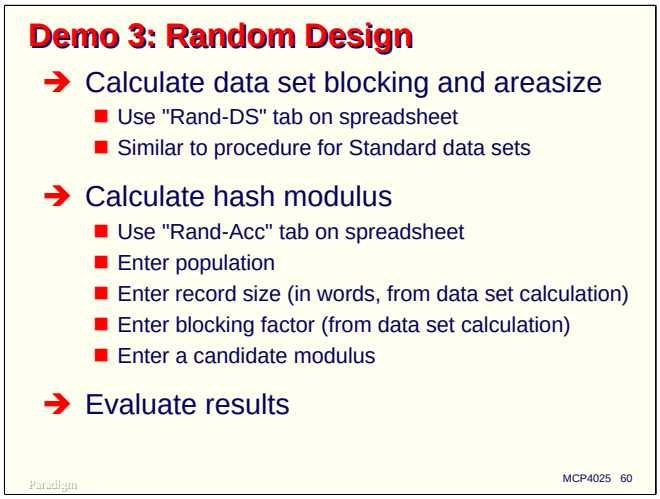


The first step in designing for Random data sets is to compute the record size. This is done exactly the same way as for Standard fixed-format data sets, accounting for slack digits and filler, and rounding up to a multiple of six bytes. Once you have this size, add one word to it to account for the folded key that will also be stored for the record in the block.

Next, choose a block size and compute a blocking factor for the file, taking into account the two control words at the front of the block. The same considerations apply here as for the other types of data sets—minimizing block slop and distributing what you cannot minimize among the records as filler. Since Random data sets are used primarily for random access, you normally want to keep the block size relatively small.

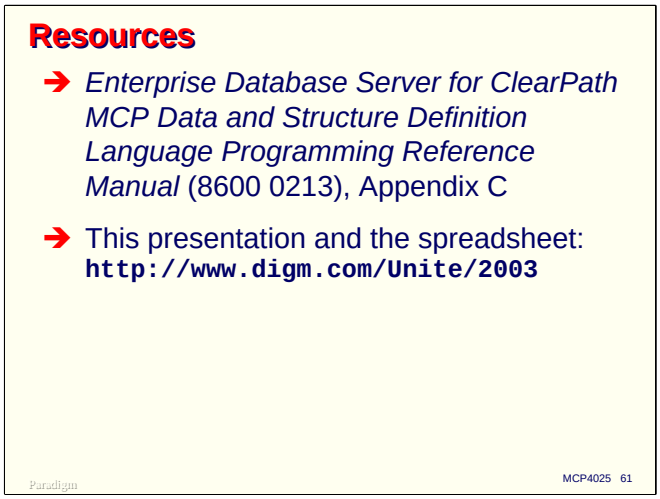
The next step is to choose a modulus and compute the distribution of overflows for the file. You should try several values of modulus, and may want to go back and try different block sizes as well.

This brief summary glosses over the fact that the calculations for a single choice of blocking factor and modulus are very involved, and the calculations for several combinations of them are only more so. It's generally not worth the trouble to try to do this by hand, but using a spreadsheet makes it relatively easy.



For our final demonstration, we will use the spreadsheet to perform trial computations to find a suitable blocking factor and modulus for a Random data set.

Unlike the other demonstrations, this one is in two parts. One tab on the spreadsheet is used to perform blocking and areasize calculations. Another tab is used to compute the overflow distribution given a blocking factor and modulus.



The best resource for information on the physical structure of DMSII files is Appendix C in the DASDL manual. This gives a thorough description of each of the data set and index types, along with some useful discussion on efficiency and design issues.

You can download a copy of this presentation and the spreadsheet from our web site at the address shown on the slide. I would also welcome comments and questions on this presentation and spreadsheet tool. My email address is paul.kimpel@digm.com.

End

Physical File Design for MCP Databases

Session MCP4025

2003 UNITE Conference