Optimizing IMAGE/TurboIMAGE Blocking and Buffering

by David Merit, Bradmark Computer Systems, Inc.

Introduction

Somewhere beyond the performance considerations of IMAGE and TurboIMAGE database design everyone knows--integer-keyed masters, sorted paths, and so forth--are the important fundamentals of blocking and buffering. These rudiments are less known and therefore often ignored, but they are significant in determining overall database performance and disc utilization.

For our purposes here, blocking involves the division of data storage on disc and its transfer to memory; buffering concerns the storage of data in memory in IMAGE/TurboIMAGE's internal control block.

Blocking and buffering are important not only for new databases but also for existing ones, and modifications normally do not impact existing programs (non-programmers and users of third-party "canned" software take heart).

Like all things tunable, changes in blocking or buffering can make things better, worse, or the same, and there are no easy answers to the questions about what to do. Every change involves a trade-off, with the type, frequency, and volume of database access, the database size, structure, and complexity, system configuration, and version of IMAGE/TurboIMAGE all significant factors in determining what needs to be done.

This article is, therefore, not an attempt to provide definitive "yes/no" answers. It attempts to do something better: equip you with the knowledge required to reach your own correct conclusions about optimizing blocking and buffering in your databases.

The block

The block is the basic unit of storage for a dataset. Each disc I/O against a database consists of a block--no less, no more (irrespective of disc caching). Each DBGET against a dataset reads not only the entry being "gotten" into the internal buffers in memory but its entire block; a DBPUT, DBDELETE, and DBUPDATE reads in the block that contains the desired entry, updates it, and writes it back.

The amount of data that will fit into a block is determined by the block size, which represents the length of a disc block and is equal to the record length of the MPE file that contains the dataset. In fact, as far as MPE is concerned, a dataset disc block (which probably contains several entries) is just one long record. A database block always begins and ends on a sector boundary and since a sector is 128 words, the block size must be a multiple of 128 words (256, 384, 512, 640, and so forth).

Blocking factor

The blocking factor represents the number of entries which are contained in each block, and therefore the number of entries gotten with each I/O and transferred into the buffers.

This blocking factor is internal to IMAGE/TurboIMAGE and should not be confused with the blocking factor of the MPE file that contains the dataset, which is always 1.

BLOCKMAX

The block size and blocking factor are automatically calculated by DBSCHEMA.PUB.SYS based on the BLOCKMAX declared in the beginning of a schema with a **$CONTROL BLOCKMAX** command. The BLOCKMAX sets the upper limit for the block sizes of all the datasets in the database, which does not mean that DBSCHEMA will set the block size for every dataset equal to the BLOCKMAX; rather, it will not exceed it and will assign a smaller block size to a dataset if this results in more efficient disc utilization.

By default, the BLOCKMAX is 512 words; if no **$CONTROL BLOCKMAX** command is specified, the default is assigned. The lowest allowable BLOCKMAX is 128 words, the highest is 2048 words for IMAGE and 2560 words for TurboIMAGE.

DBSCHEMA assigns each dataset's blocking factor based on its block size by dividing the block size by the media record length (the length of the data entry plus pointers).

Bit map

DBSCHEMA makes sure this blocking factor leaves room in the block for a bit map of between 1 and 16 words, the length depending on the blocking factor.

The bit map, used to keep track of free space within each block, is located at the beginning of the block. It occupies one word of disc space for every blocking factor multiple of 16 (since a word consists of 16 bits), so blocking factors of 1 to 16 require one word, 17 to 32 require two words, and so forth. The maximum bit map size is 16 words (since the maximum blocking factor is 255). If there is not enough space for the bit map, DBSCHEMA reduces the blocking factor as required.

DBSCHEMA

So, the BLOCKMAX, it seems is the only control you have in determining the disc storage of your database--DBSCHEMA takes it all from there: it calculates the block size for each dataset and its resultant blocking factor.

This calculation would be fine if DBSCHEMA did a good job of it, but DBSCHEMA is bent on optimizing disc usage at the expense of throughput. As mentioned, DBSCHEMA will assign a lower-than-BLOCKMAX block size if this results in less wasted disc space than that wasted if the BLOCKMAX had been assigned. Unfortunately, a lower block size usually means a lower blocking factor, and a lower blocking factor means that less entries are contained in each block. So more blocks must be read to get the same number of entries.

For example, for a dataset with a media entry length (data + pointers) of 63 words, DBSCHEMA would assign a blocking factor of six, which results in a block size of 384 words with five words wasted (63 * 6 + 1 = 379) - even if the database BLOCKMAX is 512 words. It would be more I/O efficient, although slightly less disc efficient, to instead size the block at 512 words, which yields a blocking factor of eight with seven words wasted (63 * 8 + 1 = 505).

A less obvious circumstance in which DBSCHEMA's assigned blocking factor may not be the best is in datasets which are created with very low capacities, since DBSCHEMA will not assign a blocking factor higher than the capacity. The problem becomes apparent when you increase the capacity but retain the existing blocking factor.

Assigning blocking factors

It is normally preferable to trade off some disc space for a higher blocking factor, since this improves throughput, which is more costly than disc space. Usually, the disc savings from lower-than-BLOCKMAX block sizes are not very significant.

Fortunately, it is possible to override DBSCHEMA's calculation and assign your own blocking factors, as you would want to do in the noted scenarios. You can do this for all the datasets in the database or only for particular datasets by enclosing the blocking factor in parentheses on the dataset's **CAPACITY** line in the schema; for example:

```
    CAPACITY:  100000(8);
```

This does not mean you have to recalculate and declare blocking factors for every dataset in your database--you need to override only those which result in datasets blocked less than the BLOCKMAX.

For an existing database, do a **:LISTF,2** to identify the lower-than-BLOCKMAX sets. Look at the record lengths of the dataset files--those which are lower than the BLOCKMAX are candidates for reblocking.

For a new database, add a **$CONTROL NOROOT** at the top of the schema which suppresses the creation of the root file but displays the dataset summary table at the bottom, as shown in this example from HP's TurboIMAGE Reference Manual:

DATA SET NAME	TYPE	FLD CNT	PT CT	ENTR LGTH	MED REC	CAPACITY	BLK FAC	BLK LGTH	DISC SPACE
EMPLOYEE	M	4	1	7	17	500	30	512	72
PROJECT-MASTER	M	2	1	10	20	75	19	382	15
LABOR	D	4	2	10	18	10024	28	506	1436

Here, the EMPLOYEE set is blocked perfectly--a block length (BLK LGTH) of 512 is an even multiple of 128, so there is no wasted space. DBSCHEMA arrived at this figure by multiplying the media record length (MED REC) by the blocking factor (BLK FAC) and then added two words for the bit map. The LABOR set's block length is 506 words which when fit into a block size of 512 words (the nearest 128-word multiple), wastes only 6 words per block (512 - 506 = 6).

For PROJECT-MASTER, DBSCHEMA is saving disc space--it chose a buffer length of 382 words because it is very disc efficient, since it fits snugly into a lower-than-BLOCKMAX block size of 384 words (384 - 382 = 2 words wasted per block). This results in a blocking factor of 19. This dataset could be reblocked into a 512 word block, which would waste 10 words per block instead of two, but would increase the blocking factor from 19 to 25 (20 * 25 = 500 + 2 word bit map = 502).

So, to determine for which sets DBSCHEMA assigns lower blocking factors, compare the BLK LGTH with the BLOCKMAX. Those which fall short of the BLOCKMAX by more than one media entry length (MED REC) should be examined to determine whether the blocking factor could be increased--thus increasing the block size--while staying within the BLOCKMAX.

In doing your calculation, make sure to use the media entry length instead of the data entry length (ENTR LGTH), since the data entry length does not include the pointers. Also remember that an increased blocking factor may require a larger bit map. Make sure there is sufficient space in the block.

Effective blocking by design

It is a good idea to keep blocking effectiveness in mind when designing a database, since it is quite easy to design a dataset which cannot be blocked efficiently.

When designing a database it is best to avoid very long records since they result in very low blocking factors and may require use of a large BLOCKMAX (even up to the IMAGE maximum of 2048 words or TurboIMAGE maximum of 2560 words) to get a decent blocking factor.

It can be tempting to pack something like a customer or part record full of so much data that these datasets have very long record lengths and therefore blocking factors of only 1 or 2. Normally, it is those datasets that are heavily accessed and in which you can ill-afford performance problems. The ramifications are worsened for master sets because the impact of secondaries is greater in masters with low blocking factors.

Another problem is that some datasets just cannot be effectively blocked; for example, a media entry length of 260 words will never block efficiently since it cannot fit into a multiple of 128 without leaving a lot of residual space. If, on the other hand, a media entry length of 254 could be used, it and a one- word bit map would fit quite snugly into any valid block size.

This, of course, does not suggest that you should seriously compromise your database structure, but rather that there is often enough flexibility so you can bend into near-128 word multiples.

Again, remember that you must calculate blocking factors based on media entry length--not data entry length. To determine the media entry length, either process your schema through DBSCHEMA and look at the table at the end or add in the amount of overhead to the data entry length as follows:

master (IMAGE):	5 words, plus 5 words per path
master (TurboIMAGE):	5 words, plus 6 words per path
detail:	4 words per path

Also, be sure to leave room for the bit map. In the example, if the media entry length was 256 words, the extra word needed for the bit map would force the block size up by a sector--128 words--of which only 1 word would be used!

DBCONV

Actually, this scratches the surface of something which caused a quick, painless, and relatively covert loss of blocking effectiveness in thousands of HP3000 sites--the conversion to TurboIMAGE via DBCONV.PUB.SYS.

As we've seen, the media entry length is the basis for calculating a blocking factor, so a change in the media entry length will usually require a change in the blocking factor. Database restructuring utilities are smart enough to recalculate the blocking factor and assign it for datasets in which an item length is changed or an item or path is added or deleted. A structural change via DBUNLOAD/DBLOAD will also result in a recalculated blocking factor because a new schema is processed by DBSCHEMA.

One of the functions of the DBCONV program is to increase the length of the chain count field in non-standalone master datasets from 1 to 2 words (to be able to support chains with more than 64 K entries). This, in effect, causes a change in the media entry length but not a change in the blocking factor.

If a master dataset does not have enough residual space in its existing block to accommodate the extra word or words--namely, master sets which are blocked effectively--DBCONV increases its block size by up to four sectors. One sector, however, is usually sufficient.

There are two problems with the block size increases: first, the newly acquired sector is normally not being efficiently utilized, since it is just an overflow area of which as little as 1 word (of 128) could be used. The overall wasted disc space in a DBCONV-converted database can be quite substantial. These sets should be reblocked for greater disc and I/O efficiency.

The other problem is that the block size is increased for only some of the datasets. Remember, standalone masters, masters that have enough room for expansion, and detail datasets are not increased; also some sets may be increased by one sector while others up to four sectors. The result is that the block sizes of the datasets are inconsistent.

The drawback of inconsistent block sizes leads us into the topic of buffering.

Buffering

Whenever IMAGE/TurboIMAGE reads data from or writes data to a database, it moves the data into buffers which reside in an internal control block shared among all the datasets in the database.

The control block is sized to accommodate the largest block in the database, so if inconsistent block sizes are used in the database, data from some data sets are read into oversized buffers--a waste of valuable buffer space.

A buffer caching scheme not unlike disc caching is utilized in IMAGE/TurboIMAGE. For DBFINDs and DBGETs, the data are kept around in these buffers until all the available buffers are exhausted, at which time the dirty buffers are reused. In doing so, the buffers are simply overlaid with new data--since DBFIND and DBGET read but don't write, they do not update the buffers.

For DBPUTs, DBDELETEs, and DBUPDATEs, however, buffers are updated and must be written back to disc before they can be reused. IMAGE/TurboIMAGE is not comfortable leaving modified buffers lying around in its memory-resident control block, so it abandons its caching scheme and posts the buffers back to disc at the end of every DBPUT, DBGET, and DBDELETE intrinsic call. This is one of the reasons that IMAGE/TurboIMAGE is so reliable--not more than one intrinsic can be lost in the event of a system failure.

Output deferred and AUTODEFER

It is possible to force IMAGE/TurboIMAGE to keep these updated buffers from a DBPUT, DBDELETE, or DBUPDATE hanging around in its buffer control block rather than posting them back to disc until either they are needed for reuse or until the database is closed.

The resultant performance benefits of deferring posting can be substantial, but the risk factor is high because updated disc blocks from several intrinsics that can cover several datasets will be lost if a system failure occurs. In that event, you are certain to have both physical and logical database integrity problems.

The appropriate time to defer posting is for update tasks which can be redone so that if something goes wrong you can restore a backup copy of the database and rerun the process.

A substantial improvement in TurboIMAGE is the introduction of AUTODEFER, which allows output to be deferred for a database and which may be enabled and disabled via DBUTIL. Before Turbo/IMAGE, output deferred mode could be enabled for only one accessor.

Unfortunately, neither ILR nor rollback recovery can be used with AUTODEFER, and both must be disabled before AUTODEFER can be enabled.

Buffer management

In IMAGE, a fundamental throughput bottleneck was that not more than one intrinsic could execute at a time against a database since each intrinsic monopolized the buffer control block. This type of access is referred to as single-threaded, since only a single intrinsic can be processed at a time.

In TurboIMAGE, a new buffer management system was implemented, permitting intrinsics that require only one buffer (DBFIND, DBUPDATE, and some modes of DBGET) to access the buffer control block concurrently. This is called multi-threading. Intrinsics, however, which require more than one buffer (DBPUT, DBDELETE) are still single-threaded and maintain exclusive access of the buffer control block until completion.

To accomplish this, TurboIMAGE records dynamic information about each buffer's use in the buffer's 17-word header. When an intrinsic looks for a particular data block, TurboIMAGE searches the buffer control block to see if it contains that block. If it does not, TurboIMAGE selects a buffer for use based on a least-recently-used algorithm which considers, in this order, if the buffer is dirty, the possibility that the buffer will be needed again, and when the buffer was last used.

The advantages of TurboIMAGE's buffer management scheme is that it provides concurrent access for readers (users doing DBFINDs and DBGETs) and a smarter reuse of available buffers. Another benefit of TurboIMAGE is that more space is available for buffers due to a restructuring of the internal control blocks.

Buffer control block

In IMAGE, the control block which houses the buffers is called the DataBase Control Block (DBCB) and contains the Dataset Control Block (DSCB), a group of tables which reflect the database structure, and the Lock Area, which keeps track of the locks applied to the database. The DSCB occupies a fixed area determined by the size and structure of the database; the Lock Area expands and contracts dynamically between 128 and 4096 words as DBLOCKs and DBUNLOCKs are called. The residual space of about 28 K words is available for buffers.

In TurboIMAGE, because of the increased limits in database structure (199 sets instead of 99 and 1023 items instead of 255) the space required for the DSCB increased substantially and would not even fit into a single extra data segment. So up to five extra data segments may be required to house the renamed Database Global (DBG) control block, which now contains the Lock Area. The buffers were moved into a new control block--the Database Buffers (DBB) control block--which, after some overhead, has about 28 K words available for buffers.

Buffer allocation

Now, just because IMAGE/TurboIMAGE has a particular amount of space available for buffers does not mean that it uses this entire space; rather, it dynamically allocates buffers based on the complexity of the database and the number of users accessing it. The default buffer allocation strategy is calculated based on the average number of buffers required to do a DBPUT to the most path-ridden detail set. The formula is:

```
    4 * number of related automatic masters
+ 3 * number of related manual masters
+ 1
```

So for a database in which the most complex relationship is a detail set related to five masters, three of which are automatic masters, 19 buffers will be allocated ((3 * 4) + (3 * 2) + 1)

This algorithm sets the number of buffers for one or two users. For each additional pair of users, another buffer is allocated. The idea behind this tiered buffer allocation strategy is that only the required number of buffers is allocated at any time.

Default buffer allocation

IMAGE/TurboIMAGE's buffer allocation strategy seems to be sensible on the surface, since it minimizes memory usage by keeping the buffer control block as small as possible while still providing what it considers to be an adequate number of buffers. However, this default strategy is outdated and for several reasons should not be used.

First, the number of buffers allocated is based on single- threaded IMAGE--not multi-threaded TurboIMAGE--and does not take into account concurrent read access, which benefits from more buffers.

Second, the formula does not take into account other factors which may increase buffer demands such as ILR for DBPUTs and DBDELETEs.

Third, while dynamically sizing the buffer control block based on the number of accessors conserves memory, it cheats you out of potential buffers. This minimal memory conservation was far more important on 3000s of the early days, which supported less memory than your PC. Today, memory is cheap and plentiful.

Fourth, dynamically allocating buffers based on the number of accessors requires that the buffer control block be resized as each pair of users log on or off the database.

Lastly, a batch job looks like one user and therefore is, by default, assigned the minimum number of buffers. A batch job may, of course, be performing a lot of database access and should have lots of buffers.

Recommended buffer allocation

A better strategy is to have IMAGE/TurboIMAGE allocate as many buffers as possible all the time for any number of users. This means that the buffer control block is initially build to its full 32K size and does not change regardless of the number of users accessing it.

Having the maximum number of buffers assists both database readers and writers. DBFIND and DBGET benefit because there are more buffers to maintain the various data being accessed concurrently. This also increases the chances of having data blocks which are accessed by different intrinsics as part of one logical transaction still in the buffers (for example, a chain head from a DBFIND which is used for the subsequent DBGET, and then that entry used for the subsequent DBUPDATE).

DBPUT and DBDELETE benefit because the probability that enough buffers exist to complete an intrinsic call in one control block's worth of buffers is higher. If not, significantly more disc I/O is required, since when performing a DBPUT or DBDELETE IMAGE/TurboIMAGE first previews the transaction to make sure it can be completed.

For example, to DBPUT a new detail entry, IMAGE/TurboIMAGE has to check to make sure that free space exists in the detail set; that all related manual master entries exist; and that all related automatic master entries exist (and, if not, that free space exists to create them). To verify these things, IMAGE/TurboIMAGE may exhaust its buffers and require that they be refreshed with the rest of the data. And once it has verified that the intrinsic will complete, it has to again fill its buffers with the first set of data blocks, post them back to disc, and then do the same with the second set.

Naturally, this is not to say that every DBPUT will thrash the buffers this way. The number of buffers required by an intrinsic call varies not only on the intrinsic but the circumstances. For example, a DBPUT to a sorted chain will require a variable number of buffers based on how far up the chain needs to be read. A DBDELETE in a master will take a variable number of buffers based on whether the entry is a primary or secondary, the position on the secondary chain, and whether the chain is contained in a single block. A DBPUT to a detail set will require more buffers if the related automatic master entries do not exist, since they must be created.

Since the number of buffers may vary and exceed the defaults assigned by IMAGE/TurboIMAGE, it is better to have more buffers to absorb those intrinsic calls which require an inordinate number of buffers.

That explains why it is beneficial to have a lot of buffers, but why the same number of buffers regardless of the number of accessors? For each pair of users who log on or off the database, the buffer control block must be resized. To do so, IMAGE/TurboIMAGE locks the DBB, waits until all buffers are released by other processes, and the rebuilds it. This requires exclusive access to the control block and therefore impedes users waiting to process.

BUFFSPECS

Fortunately, there is an easy way of adjusting the number of buffers in IMAGE/TurboIMAGE. Just specify them in DBUTIL, for example:

```
>>SET base BUFFSPECS = 30(1/200)
```

This command allocates 30 buffers for anywhere between 1 and 200 database accessors. This causes a consistent number of buffers to be allocated with no dynamic sizing of the control block, and allocates more buffers than IMAGE/TurboIMAGE's default.

Of course, the obvious question here is "how to you determine how many buffers to allocate?"

For TurboIMAGE, you don't need to make this determination. You can simply set an impossibly high number of buffers (for example, 255) and TurboIMAGE will allocate the maximum number of buffers possible.

Now, don't try this for IMAGE because specifying more buffers than can fit in the available space can cause a buffer supply crisis and resulting problems. The reason is that the both the buffers and the Lock Area are contained in the same control block (DBCB) and both areas are dynamic. The Lock Area expands "downward" while the buffer area expands "upward," and if too many buffers are allocated and the Lock Area needs to expand, the two collide. This problem does not exist in TurboIMAGE because the Lock Area is contained in a different control block.

For IMAGE, it is necessary to calculate the number of buffers. To do this, you first need to know how big the buffers are.

Buffer length

Just as the number of entries that may fit into a block is determined by the entry length, the number of buffers that can fit into the buffer control block is determined by the buffer length.

The buffer length is calculated based on the largest block size in the database, since the buffers are shared among all the datasets with any block being able to be read into any buffer. The method is to take the block length (BLK LGTH from DBSCHEMA's dataset summary table, which represents the blocking factor times the media record length plus the bit map) and add nine

words for IMAGE or 17 words for TurboIMAGE (for the buffer header). Unlike the block size, the buffer length is not rounded up to the nearest 128-word multiple.

How many buffers?

Now that you have calculated the length of the buffer, how many of them can you have? To determine this, you need to figure out how much space is available for buffers. The formulas are different for IMAGE and Turbo/IMAGE.

For IMAGE, you subtract the fixed space needed for tables and the maximum possible Lock Area from 32 K words, and this yields the available buffer space.

Let's try it using the same database from the TurboIMAGE manual. The information in the DBSCHEMA output below the dataset summary table is needed for this calculation; to calculate the available buffer space for this database:

```
ITEM NAME COUNT: 23          DATA SET COUNT:  6
ROOT LENGTH:  1176  BUFFER LENGTH: 505  TRAILER LENGTH: 256
```

goes like this:

32767 words	(32K, maximum DBCB size)
- 150 words	(fixed overhead)
- 1176 words	(ROOT LENGTH)
- 23 words	(ITEM NAME COUNT)
- 6 words	(DATA SET COUNT)
- 256 words	(TRAILER LENGTH)
- 4096 words	(maximum Lock Area)

The result is that 27,060 words are available for buffers. Since the buffer length for this database is 521 words (512 + 9), 51 buffers as most can be allocated (27,060 / 521).

This formula changes in TurboIMAGE. The calculation for the same database in TurboIMAGE is:

32767 words	(32K, maximum DBB size)
- 4000 words	(fixed overhead)
- 156 words	(26 words * DATA SET COUNT)

which results in 28,611 words available for buffers. In TurboIMAGE, the buffer length increases to 529 words (512 + 17), so 54 buffers may be allocated (slightly more than in IMAGE).

Determining block size

Now that we've reviewed what blocking and buffering are all about, how they work, and how to adjust them, let's focus on the fundamental element in all of this: block size.

The block size ultimately determines disc storage efficiency, the amount of data transferred from disc to memory, the size of the buffers, and the available buffer space.

Basically, small blocks mean lower blocking factors and more buffers; large blocks mean higher blocking factors and fewer buffers. Choosing a good block size is fundamental to the performance of your database.

The TurboIMAGE manual lists some guidelines in selecting a BLOCKMAX (and thereby block sizes). It says you should consider efficient space utilization, minimum disc accesses, and program execution time which can be affected by size of the DBB. It recommends using larger blocks if applications run in batch when system resources are plentiful and smaller blocks if applications are large and run concurrently with many online users. This is because the resulting large DBG and DBB may cause applications to run more slowly since a large area of memory is required, and this may not be necessary for small applications.

The concerns about program execution time are generally not significant, since most systems have enough memory to comfortably support large control blocks. As far as efficient disc storage goes, larger block sizes generally store data more efficiently (although DBSCHEMA will ignore the BLOCKMAX to save some disc space).

The concern about minimizing disc access require a closer look.

Disc I/O bottleneck

The most common basic performance problem in IMAGE/TurboIMAGE and other systems on the HP3000 and other computers is that data cannot be moved to and from disc fast enough: the system is ready to process data but it must wait on the disc drives for the data to be retrieved, so a fundamental bottleneck in the system is disc I/O.

It is always beneficial to reduce the amount of disc I/O, and one factor external to IMAGE/TurboIMAGE which must be considered here is disc caching, both in memory (system disc caching) or on the disc drive itself (controller caching).

Disc caching

The concepts of disc caching are quite simple: with each disc I/O, you read more data then you need right now with the expectation that you might need it soon--after all, as long as you're going to spend resources on a disc I/O, you might as well get as much data as you can with about the same overhead. Also, data that is in demand remains in cache while unpopular data is flushed to make room for data from subsequent I/Os.

For system disc caching, the amount of data retrieved is determined by fetch quantums set via the :CACHECONTROL command. For IMAGE/TurboIMAGE, only the random fetch (not the sequential fetch) quantum is applicable--even for sequential access of a dataset. The fetch quantum determines, for each disc I/O performed against a given location, how many sectors following that address are also pulled from disc in the same I/O.

Disc caching acts as a cushion for such deficiencies as low blocking factors and poor data locality. A cached I/O against a dataset with a block size of 512 words and a blocking factor of 3, and with the random fetch quantum set for 96 (the maximum), will transfer 72 records from disc to memory. Remember, that the data still has to be read into the buffer control block a block at a time, but this is many times faster than a disc I/O.

There are situations in which disc caching does not help or is a hindrance to database access. For instance, you may be accessing data which in the dataset physically precedes the data read previously, which is a common occurrence when reading backward up a chain (DBGET mode-6) or when accessing entries whose locations were determined by reusing space on the delete chain. Since caching reads forward only, useless data is swapped into cache and then must be searched through before going back to disc. The result is increased overhead from the memory manager.

Large blocks versus small blocks

In the above scenario, bigger blocks are preferable. Serial reads especially benefit from big blocks because many qualifying entries are read with each disc I/O and a lot of in-demand data are read into the buffers at once. The same is true for chained reads of data with good data locality.

Small blocks mean low blocking factors, which suggest increased disc activity. If less entries are contained in each block, more blocks have to be read from disc to get all the required data. The benefit of disc caching here, however, should not be minimized. Small blocks also mean more buffers, which benefits concurrent read access and cushions DBPUTs and DBDELETEs.

Regardless of the block size you select, calculate the number of buffers required for a DBPUT to the most complex detail set and make sure at least that number of buffers can be allocated. If not, buffer thrashing will result and you should reduce the block size.

Summary

This summary of the recommendations explained in this article. Use them as guidelines--not rules--in optimizing blocking and buffering in your databases:

- override DBSCHEMA's calculated blocking factor when it results in a block size less than the BLOCKMAX
- recalculate the blocking factors for datasets that are initially created with very low capacities and whose capacities are increased
- avoid designing datasets with very long records, since they result in very low blocking factors

- avoid designing datasets with records which cannot be effectively blocked

- reblock DBCONV-expanded TurboIMAGE master datasets to be more disc and I/O efficient

- enforce a consistent block size for all datasets to maximize available buffer space

- use output deferred mode (AUTODEFER) for write-intensive tasks which can be rerun in case of failure

- override buffer allocation defaults to allocate the maximum number of buffers to improve throughput. For IMAGE, calculate the maximum; for TurboIMAGE, assign an unattainable value and the maximum allowable will be assigned

- override the default allocation of buffers based on the number of accessors; assign a fixed number of buffers for any number of users to prevent the buffer control block from being rebuilt

- determine an efficient BLOCKMAX and make sure that it does not limit the number of buffers to the point that they thrash