Optimizing AS/400 Batch Performance[1]

by Rick Turner (rtai01@attglobal.net)

In recent months, I've worked with some AS/400 sites who needed to increase their back-office batch workloads anywhere from 5 to 50 times. One case was a payroll application that needed to grow five-fold from processing 100,000 employees to processing 500,000. Another was a bank that, as a result of consolidation, needed to grow from 600,000 accounts to 30,000,000. In both cases, the increased workloads had to be processed with no increase in elapsed time.

To meet their processing time objectives, these clients had to change their application programs to optimize their programs' ability to exploit the power of the AS/400. This approach included using multiple copies of the processing jobs, each working on separate parts of the input data to complete more work in the same amount of time by increasing the CPU use. This is a workable approach because, given the proper application programming techniques, the AS/400 handles multiple jobs at once very well.

But even with the changes, the users were often unable to drive their batch applications' throughput to their systems' resource limits. As a result, they couldn't meet their processing time requirements. Their question for me was, "How can I get a lot more work done in the same amount of time?"

The answer is to get the system's CPU to run at as high a utilization as possible as well as to drive the disk usage up to the recommended utilization threshold values. You accomplish this by using resources to do useful work and reducing unnecessary system and application overhead. Fortunately, with the new 8*xx* models, IBM Rochester has removed many former hardware capacity and performance restrictions, enabling a much more formidable system that exploits the new Silicon-on-Insulator (SOI) processor architecture, a 96 GB main memory, and an awesome amount of available disk space.

[1] This paper originally appeared as a two part article by the author in NEWS400, September & October 2000.

This article discusses some of the processing options that can improve batch throughput. I'm not going to tell you how to write your application or use SQL or get the most out of Query Optimization. I simply offer some general ideas about structuring an application and setting up an optimal runtime environment (both hardware and software) to complete as much useful work as possible in the shortest amount of time.

What is Batch Processing?

Many "experts" in the world of IS punditry seem unaware that the AS/400 handles *two* types of information processing quite well. The traditional scenario involves large volumes of short, fast processing associated with input from 5250 clients or other client/server devices. A similar scenario involves the processing of message-driven jobs that handle one input request at a time.

The second (and lesser known) type of processing that the AS/400 does well is batch. Some business people still believe large volumes of batch work are best left to big, isolated ("glass-house") mainframes. But in fact, the AS/400 can complete a lot of work at a much lower cost per unit than some of the big guys can achieve.

OS/400 defines several job types, including type B for batch. Another job type is I for interactive, which identifies jobs that work with input data originating from a 5250 workstation or another device that emulates a 5250 data stream. The WRKACTJOB (Work with Active Jobs) command shows an active job's type designation.

The distinction between type-B and type-I jobs ensures the routing of a work unit to the proper *subsystem* (e.g., QBATCH for batch, QINTER for interactive). Also, when a type-I job uses the CPU, the amount of processor cycles consumed are debited against the machine's capacity to perform interactive work. (This value is called the Interactive Commercial Processing Workload (ICPW) rating.) However, when a type-B job uses the CPU, the amount of processor cycles consumed are debited against the machine's total capacity. (This value is called the Processor Commercial Processing Workload (PCPW) rating.)

Any type of job can process database or other types of data. To process database data, the job may read a record, change it, and write it back to the file; read some data, process it, and write a new record back to the file from which it read the data; or originate new data and add it to either an existing file or a new file. My focus here is improving the performance of batch jobs that process database data (e.g., read, update, add, delete) on a single system (or within a single partition on an logical-partition (LPAR) system).

Analyzing Your Application's Performance

To improve an application's performance, you must be able to measure it in a controlled environment before and after you make changes. It's imperative that you have a repeatable runtime environment, a set of input data, and a backup of that data.

Before each measured run, you should restore the input data to disk and remove any residual data from main storage so the number of physical disk operations isn't unduly influenced. To remove the database files from main storage, use the SETOBJACC (Set Object Access) command specifying the file name and the *PURGE option.

Two criteria establish batch job performance. One is the amount of completed work versus the elapsed time required to complete it; the other is the amount of completed work versus the amount of system resources required to complete it. With batch jobs that process database data, you determine the amount of work done by counting the number of logical disk I/O operations that occur during the job's lifetime. Logical disk I/O operations are OS/400 Data Management's way of counting database file accesses.

To collect the data you need to evaluate your application's performance, you can use either the OS/400 Performance Monitor or (as of V4R4) Collection Services. For each job, you need

• the job name

• the user ID

• the job number

• the job's elapsed time

• the CPU time required

• the number of physical disk I/O operations

• the number of logical disk I/O operations (If you know the number of primary input database records (e.g., the number of customer account records processed), you should use this value instead of the logical disk I/O record count.)

Additionally, you need the total CPU time used for all jobs and tasks in the system during the runtime of the job(s) in question, the number of collection intervals, and the length (in time) of each collection interval. The system collects this information and stores it in the Performance Monitor's QAPMJOBS file or Collection Services' QAPMJOBL file in the performance data collection library.

Finally, you also need to know your system's CPU model and feature code and the number of CPUs in use. This data is in the report header lines in either

These numbers can help you determine what type of performance to expect after you make changes. For simplicity, assume your batch job(s) will be run standalone so the machine's total CPU capacity is available for the application. If this isn't the case, estimate what percentage of the total capacity is used by other jobs and adjust your application's performance expectations downward accordingly.

If you know the number of application records you have to process in your application test, use that value to establish your throughput and resource-cost values. For example, the application may consist of a single job (or a string of individual jobs running one at a time) that process 100,000 customer account (CA) records per hour. You might want to normalize the throughput and resource usage rates. The processing rate normalized to one second is 27.7 CA/second (100,000/3,600). If the job used 1,000 seconds of CPU time to process the records, the CPU resource cost is 10 milliseconds per CA record (1,000/100,000).

Set Your Sights

What are your application's performance requirements? Using our example, assume you must grow the workload by 400 percent and still run it in the same amount of time (one hour). This means you must process 500,000 records in one hour. It also means you must use 5,000 seconds of CPU time (because the CPU time per CA record was 10 milliseconds).

Now you must ask yourself:

1. Is there enough CPU time available to do the work in the prescribed elapsed time?

2. What must I do to increase the throughput rate by 400 percent?

3. How many disk arms do I need to maintain my throughput rate without the disks becoming a bottleneck?

Regarding CPU availability, you have a couple of options. One is to reduce the amount of CPU resource used by the job so you can process 500,000 records in 3,600 seconds. In our example, only one job is running at a time; this is key because on the AS/400, a single job that isn't using multi-threading can use only one CPU at a time. Assuming, for now, that the job could drive the processor to 100 percent busy, you'd have a shortfall of 1,400 seconds (500,000 records at .010 seconds/record is 5,000 seconds). With 3,600 seconds available in an hour, the requirements exceed the machine's current capability.

Continuing with the assumption that the job could drive the processor at 100 percent busy, the job's CPU usage must be reduced from .010 seconds per CA record to .0072 seconds per CA record (or 28 percent) to meet the requirement. If you can't reduce the CPU usage this much, you'll need more CPU time to meet the processing time objective. You need a machine that's at least 28 percent faster so you have the equivalent of 5,000 seconds of the current system's CPU time available in an hour.

The key assumption here is that the application job can drive the CPU to 100 percent busy. In almost all cases of batch database processing, this isn't the case — at least not initially. Most batch work runs at CPU utilization rates between 2 and 8 percent. In our example, the job used 1,000 seconds of CPU time in an hour, which is 27.7 percent utilization.

So how do you drive up the CPU usage? The answer is to run multiple concurrent jobs (i.e., use batch-job parallelism) by restructuring the runtime environment to have more than one copy of the job running at once.

If your single job is using some percentage of the CPU in a single job environment, and you want to increase your application's throughput, your objective is to get *k* copies of the job running, where *k* is 100 divided by the job's CPU utilization. In our preceding example, *k* would be about four (100/27.7). You need to distribute the processing workload (e.g., input data) evenly over four jobs and let them run. In some applications, this is easy to achieve with the OVRDBF (Override Database File) command; in others, it requires some study and reworking of the application.

What if the value of *k* is so large that you over-commit the processor and try to drive it at 150 percent or higher? In this case, you need to have more CPU time available, which means moving to an *n*-way processor. The *n*-way processors have multiple CPUs, which provide you with (*n* x 3,600) seconds of CPU time per hour (or *n* seconds of CPU time per second).

Thus, with a demand for 5,000 seconds of CPU time to complete the job in one hour, you need at least a two-way processor that has 7,200 seconds of CPU time available in an hour (3,600 seconds per hour per CPU times two CPUs). If you run five copies of the job simultaneously, they should complete their work in the prescribed time and use about 70 percent of the CPU (5,000/7,200).

Can't I Just Upgrade?

You may be thinking, "This all sounds fine, but what must I do to implement concurrent job execution of the same job? It's probably going to take some work; can't I just upgrade the hardware?"

A proper hardware upgrade can usually provide real improvement in interactive and message-based processing without a lot of time and people investment on your part. This is the AS/400 *n*-way processors' greatest benefit. If you want to increase the number of workstation transactions on your system but you're currently maxed out, you'll probably see a significant improvement in the total number of transactions you can process.

Just remember that moving to a model with multiple CPUs isn't going to help a single batch job very much unless each CPU is much faster than your current system's CPU. Getting a faster CPU usually requires moving to another model group (e.g., from 720 to 730, from 730 to 740).

The AS/400's extendibility to multiple processors greatly facilitates the growth of interactive and server applications, which characteristically process each request in a relatively short time (compared to a batch job). Therefore, you can have a lot of them running simultaneously, each working on a separate transaction for several users. This isn't the case with batch database processing, where a job runs for a long time.

For example, with bigger hardware, a bank can almost certainly process more teller and ATM transactions. But can it process all the customer records that must be handled *and*

complete their backups before the bank opens the next morning? With more hardware, a payroll processing application can handle more employee record updates and online time-card processing. But can it also process the final payroll and disburse funds (as printed checks or fund transfers) to online accounts on time?

Most legacy batch applications and many new database update batch-processing applications are designed to run as single threads of execution. By this I mean that they read an input record, update a file (e.g., pay interest to a bank account, apply withdrawals and deposits), or generate a new database record for a file (e.g., add detail lines to a payroll file). Once the database processing is complete, they go into report-generation mode. In short, they do things one step at a time, and they aren't designed to achieve execution-time parallelism.

If you can't split the job into parallel execution strings, you can't take much advantage of an *n*-way processor. It may help a little — but probably not enough to get the job to run with much higher volumes. What you'd probably see in this case is some of the System Licensed Internal Code (SLIC) tasks running on one processor and the single batch jobs running on another.

In both interactive and batch processing applications, a job or thread uses only one CPU at a time. But when a lot of jobs (e.g., multiple concurrent interactive or server jobs) are running, you can drive more than one of an *n*-way processor's CPUs concurrently.

Where do I go from here?

From here on I discuss a number of options you can use to improve batch performance. In some cases these recommendations could improve interactive also.

Use MAX1TB Database Indexes

To achieve proper database performance in an environment where multiple concurrent jobs are updating/deleting/adding to the same physical file, you must create or change your logical files using the MAX1TB parameter on the CRTLF (Create Logical File) and CRTPF (Create Physical File) commands. Logical files created/changed this way use 4 byte indexes.

Simply converting logical files to MAX1TB indexes alone won't improve performance. But it ensures that you'll have minimum logical file contention and improved processing when the files are changed (e.g., via a record update/add/delete) in a multiple-concurrent-job environment.

The processing improvements occur as a result of the database functions' "levels of locking" algorithms. When multiple jobs need to update an logical file that uses a 3 byte index, any contention for it requires locking at the highest level (or node) of the index. This means that if multiple jobs each need exclusive control of a different record in the index (for an add or delete operation), only one job at a time can have control.

This bottleneck imposes a hard limit on throughput and can cause consistently low system utilization. If you've tried using symmetric multiprocessing (SMP) on a system without converting to MAX1TB indexes, you probably didn't get the improvement you wanted

With 4 byte indexes, locking occurs at the *leaf level*, so multiple jobs can access different index records concurrently without significant lock contention and without having any real effect on database processing throughput. ("Leaf" is a term used by the Rochester database guys; basically, it's a lower-level node of the binary radix tree.)

MAX1TB is the default setting for new logical files created on RISC systems. Logical files migrated from Internal Microprogrammed Interface (IMPI) systems aren't automatically converted to MAX1TB and will be MAX4GB (3 byte) indexes unless explicitly converted.

To find which of your logical files have 3 byte indexes, run the DSPFD (Display File Description) command for each of the logical files in your database library). If you have keyed physical files, each physical file's index may have 3 byte or 4 byte indexes. To find the physical

files with 3 byte indexes, run DSPFD for your physical files. (put in an explicit example of DSPFD for both Logical & Physical files)

To convert logical files from 3 byte to 4 byte indexes, use CHGLF (or CHGPF for keyed physical files). Depending on your system capacity, use of SMP, and file sizes, this may take anywhere from 10 minutes to 12 hours, depending on the size of the files and the hardware and software configuration. Using SMP can improve the conversion time significantly because the work of changing a logical file is spread among multiple SLIC DBL3 tasks and multiple processors. (put in an example of CHGLF and also for changing keyed physical files from 3 byte to 4 byte).

During CHGLF processing, the file is locked exclusively, so you should convert files when other jobs don't need them.

Splitting Input into Multiple Jobs

Having multiple concurrent jobs each perform part of an application's work on a system can drive up CPU usage and increase application throughput. Once you've determined the number of jobs you want to run, distribute the input data evenly among them and start them running.

This may be easier said than done. The methodology you choose to distribute input among multiple jobs may depend on your application's design and require close study of the application to see what steps are necessary.

If you have a single input file of transactions to be applied to your database, and you have one job to process the data, start multiple instances of the job (*k* of them) and spread the processing among them. The first job would process from record 1 to record (1 x nbr_rcds)/*k*, and the second job would process from record (((1 x nbr_rcds)/*k*) + 1) to (2 x nbr_rcds)/*k* (and so on), where nbr_rcds is the number of active records in the physical file. In an example with a 300-record file and three jobs (*k* = 3), the records processed by each job would be:

Job 1: From record 1 to record (1 x 300)/3

(From record 1 to record 100)

Job 2: From record (((1 x 300)/3) + 1) to record (2 x 300)/3

(From record 101 to record 200)

Job 3: From record (((2 x 300)/3) + 1) to record (3 x 300)/3

(From record (201 to record 300)

Use ALLOCATE *YES when Adding Records to a New File

Some high-volume batch jobs require considerable processing overhead due to the system overhead required to add records to a database file. This system overhead occurs even in a single job environment, and using ALLOCATE(*YES) may improve the job runtime significantly.

When you're running multiple concurrent jobs that are all concurrently adding records to the same file, the effect on throughput can be severe. The system's Auxiliary Storage Management (ASM) space-allocation functions spend considerable time extending a file's space on disk as records are added to it. While one job is extending a database file, any subsequent requests from other jobs to extend the same file must wait until the request(s) ahead of it finishes (regardless of priority). (The space-extend processing is, by necessity, a single-threaded SLIC function.) As a result, very little (if any) processing parallelism may be possible.

The AS/400 performance tools can't detect this type of bottleneck explicitly. If multiple concurrent space extension requests are occurring, the Performance Monitor's sample data shows only that the job encountered higher-than-normal seize wait time. If you run the Performance Monitor trace function, the Lock Report will show the contention as a seize wait on

the physical file. Unfortunately, the same type of seize contention can occur for a number of reasons.

Depending on the type of processing you're doing, you can use one of two procedures to avoid this overhead. In the first case, you're creating a new database file and you know that it will have a specific number of records in it. When you create the file with CRTPF specify the maximum number of records it will eventually contain and specify (*YES) on the ALLOCATE parameter. This tells the system to allocate the file's total required disk space when the file is initially created and allows processing jobs to avoid spending time in the file-extend routines during execution.

ALLOCATE (*YES) can help your jobs run much faster; both single and multiple batch jobs can benefit. A recent batch benchmark boosted overall CPU utilization from 7 percent to 50 percent simply by using ALLOCATE (*YES) when database files were created.

Add a Second File Description to your Application Program

The second way to minimize space-extension overhead applies to application programs that add records to an existing database file. First, add a second file description for the file to your processing program, and open that file description for output only. Then, prior to calling the program, issue an OVRDBF (Override Database) CL command specifying SEQONLY (*YES *nnnn*), where *nnnn* is the output blocking factor (i.e., 128 K divided by the record length equals *nnnn*). Be aggressive with the blocking factor and get it as close to 128 K as possible without exceeding it.

PTF for Database Space Allocation

If you're running V4R4 or later, install PTF MF24287 and IPL your machine. This database PTF causes the database space allocation functions to extend database physical files by an amount larger than the default of 1 MB at a time. The PTF works only if the application program is using blocked sequential output processing.

PTF MF24287 causes files under 256 MB to be extended by 10 percent to a maximum of 2 MB. Files greater than 256 MB are extended by 4 MB each time additional space is needed. One benchmark showed an increase from 50 percent busy to 80 percent busy by using this methodology. (Note: If you pre-allocate the physical files to their maximum size when they're created (i.e., specify ALLOCATE(*YES) on the CRTPF command), the PTF isn't necessary.)

Index File Updates/Adds and SMP

Symmetric multiprocessing (SMP) is a one-time-charge feature that improves multiple concurrent processing during database reads/updates/adds/deletes from one or more user jobs. When a database physical file record is updated, added, or deleted, the physical file's associated logical files must be maintained. In fact, when a program calls the database to update/add/delete a physical file record, the database functions perform the logical file processing *before* changing the physical file. This database processing sequence (first, the logical file processing; then, the physical file) holds regardless of whether SMP is used

If SMP isn't enabled, the logical file maintenance occurs synchronously within the job. The logical file records that aren't in main storage are page faulted into main storage, processed, and written to disk. After this processing is finished for all the logical files involved, the physical file is updated.

Logical file page-fault sizes vary depending on the attributes of each of the physical file's index files. The logical page size for indexes that aren't yet *MAX1TB is 4 K. The logical page size for indexes that are native access paths with the *MAX1TB attribute is 4 K. The logical page size for SQL indexes and SQL referential constraint indexes (which look and act just as access paths do) is a whopping 64 K.

With SMP turned on, logical file processing is "farmed out" to SLIC tasks named DBL3…. The processing for each logical file is assigned to a separate DBL3 task, and the user job waits until each of the DBL3 tasks finishes processing the block of records it was given. Then, the job resumes processing and the database functions write the physical file record.

If a physical file has many logical files and there are a lot of record changes, SMP can save a significant amount of time. However, note that the same amount of processing occurs, so there's no savings in CPU time. Instead, the same amount of CPU is used (per record) in less elapsed time so that CPU utilization increases as the total processing time is reduced.

QRYDEGREE Setting

If the SMP feature is on your machine, set the system value QRYDEGREE to *OPTIMIZE. Include a CHGQRYA DEGREE(*MAX) in the calling CL program within the jobs that update/add/delete files to let your system use as many DBL3 jobs as possible. However, use CHGQRYA DEGREE (*OPTIMIZE) in jobs that run SQL, OPNQRYF, or AS/400 Query. (Jobs that have the QRYDEGREE system value set to *OPTIMIZE don't need to include a CL command.) *OPTIMIZE tells the system to use some of the DBL3 tasks but to not fill the system with them.

The "Holey Inserts" Function

To optimize the performance of multiple jobs concurrently adding records to a physical file, you need to use a separate, output-only file description and specify that it's to use sequential processing and output blocking. This will help the job run better, but I guarantee you'll encounter processing bottlenecks when a lot of concurrent jobs (10 to 100 or more) are adding records to the same file in a multiple batch job environment. Once the number of jobs increases past a certain point, the CPU usage and throughput doesn't increase as much as expected. In fact, at a certain number of jobs, the total CPU usage flattens out and each additional job causes all jobs to do less work.

This drop-off occurs due to SLIC database processing serialization occurring while the output record is being built. The database functions must validate that each new record can be added to the file. If the output blocking is high (as it should be — as close to 128 K as you can get) and many jobs are adding records at once (as there should be), the validation processing becomes a bottleneck.

Fortunately, a database function in V4R4 called *holey inserts* lets multiple concurrent jobs add records concurrently without running into the contention caused by serialization. The formal name for holey inserts is ENCWT (Enable Concurrent Writes), but *holey inserts* is more descriptive of the function's methodology (Figure 2).

How to Enable "Holey Inserts"

To optimize database add-record processing on V4R4 or later, you do three things:

1. Call an OS/400 database program to **enable** holey inserts.
2. IPL your system to **turn it on**.
3. Use a separate, output-only file description and specify SEQONLY *YES and output blocking.

The green-screen command-line call sequence to enable holey inserts is CALL QDBENCWT '1'; you'd enter CALL QDBENCWT '0' to disable it. Note that calling this database program enables concurrent adds for *all* database files in the system. A database physical file can benefit from the holey inserts algorithm only if it was created with (or modified using CHGPF to have) the REUSEDLT (Reuse Deleted Records) attribute specified as *YES.

The primary downside of holey inserts is that if you're journaling, some of the journal records' ordinal numbers (e.g., count) can be out of sequence. This shouldn't be a problem with high-availability software.

Holey inserts lets multiple jobs concurrently add records to the same database file even if another job is already adding to the file. The database calculates the new record's insertion point for the subsequent jobs and lets them continue processing. If some records fail the validation tests, the database won't let them be put into the file. In this case, the database inserts a deleted record (logically, a hole in the file) in place of the rejected record — thus, the name *holey inserts*.

If the holey inserts feature isn't turned on, no deleted record is inserted in place of the rejected record. As a result, any subsequent records added to the file are moved up one record position in the file. Because of the "fuzziness" of the next job's beginning insertion point, the *SLIC Data Base functions prevent subsequent jobs from doing their add processing until the current job finishes its add processing.

The advantage of using holey inserts is that it lets multiple jobs concurrently use the CPU. This may not gain much performance on a single CPU system because the validation processing function could be highly CPU intensive and, consequently, there may not be much CPU left for other jobs during validation. However, if you have an *n*-way processor, it lets the other concurrent jobs adding to the file use the other CPUs and moves the data into the file much faster.

Journaling for Performance

For many AS/400 users, database journaling is essential for a proper system backup. Properly configured, the system's journaling functions can provide reliable high-availability information and optimized performance.

Database information is written to a journal receiver. The system's journaling functions "spread" a journal receiver across numerous disk arms. The system blocks and writes data to the journal receiver using a "round robin" approach; each journal data record is written to a different disk arm than the one used for the preceding journal receiver records. If the journal receivers are in their own disk auxiliary storage pool (ASP) and only one journal receiver is active at a time, this approach ensures that that no arm contention occurs when multiple concurrent journal writes are in progress.

The steps for achieving optimal journal performance include:

1. Put the journal receivers in their own distinct disk ASP.
2. Have only one active journal receiver in the ASP at a time.
3. Use mirroring instead of RAID to protect the journal ASP devices.

When a journal receiver is the only active object in a separate ASP, there's minimal disk arm contention. Most often, the disk arm doesn't have to be moved at the start of an I/O operation; it can be moved when a cylinder is full. However, if multiple active journal receivers are in an ASP, or the journal receiver is in the system ASP along with production database files, there's a high probability of arm contention. Each journal write will probably require arm movement (e.g., a seek) before performing the write.

If the journal receivers are in a RAID-protected ASP, each journal write must perform four disk I/O operations to perform the RAID function. For each write from the CPU, there are two reads and two writes at the device and IOP level. In an environment with a lot of write operations (e.g., a batch environment), this can significantly increase the runtime.

On a RAID-protected device, the data exists on the outside (growing inward) and the RAID stripes are on the inside (growing outward). So if you write your journal record to a RAID-protected device, the device must seek to the outer edge, perform the I/O, and then seek to the

inner edge to read the RAID information. This can really slow a high-performance journal receiver device.

Commitment Control & Journaling

Though most database and other write operations are asynchronous, database journal receiver write operations are usually synchronous to the issuing job. This means the job is forced to wait (in the system's disk I/O write functions) for the I/O (write) to complete before it continues processing. The SLIC Journal functions can do the journal writes asynchronously if the job uses commitment control.

When commitment control is in effect, the database journal write functions know that file integrity is required only at a commit boundary and not at every record update/add/delete operation. Because of this, the database journal writes are scheduled asynchronously. When a commit boundary is reached, the database functions ensure that all pending database file I/O is complete before continuing.

Lab tests show that using commitment control and journaling yields performance almost equal to not using database journaling. If you use journaling but not commitment control, a job can be *three to four times slower* than when you don't use journaling at all.

"But this means I have to change my code!" you say. True, but the cost of the changes are minimal compared to the performance benefit. In the CL program that calls the batch program, specify the files that use commitment control and open them. Start a commit cycle in the CL program before calling the batch program. In the application program(s), change the file description to specify that commitment control is in use. Once the program returns to the CL program, end the commit cycle to force any pending file I/O to complete.

SMAPP

The System Managed Access Path Protection (SMAPP) function provides recovery capability for users who want it but don't want to manage a full recovery process. You can tell SMAPP your database recovery time objective (or use the SMAPP default value shown on the EDTRCYAP (Edit Recovery Access Path) command menu), and SMAPP will ensure that the appropriate database access paths are journaled to meet it.

When you turn SMAPP on, it periodically calculates the database recovery time for running applications. If the recovery-objective value is exceeded, SMAPP adds additional access paths to the access paths it's currently journaling. If you collect Performance Monitor or Collection Services data, you can see the estimated recovery times and the number of system-managed (journaled) access paths in the Journaling section of the Performance Tool's Component Report.

Where do implicit Journal Records go?

Two types of application-throughput slowdown can occur when you use SMAPP. In the first case, the access paths are journaled into the system ASP (which may be RAID-protected) rather than into a separate ASP (which is mirrored). This happens if the access path's underlying physical file is *not* being journaled. By default, SMAPP puts the journal receiver into the system ASP. However, if the underlying physical file is being explicitly journaled, SMAPP's access path journal records will go to the same journal receiver being used by the physical file journaling.

Note that you control whether the physical file is journaled and where the journal receiver is. If you want to use SMAPP to automatically turn access-path journaling on and off, and you want to optimize access-path journal write performance, then consider journaling the physical file to a journal receiver located in a separate, mirrored ASP. The other option is to take control from SMAPP and explicitly journal the larger (and most costly to rebuild) access paths to the journal receiver in the mirrored journal ASP.

Purging Data Base pages

The other type of slowdown is related to the purging of large numbers of changed database physical and logical file records from main storage. System functions monitor the number of changed pages in memory, and the system starts writing them to disk when a certain threshold is reached. Some customer benchmark tests have shown batch throughput degradation as high as 60 to 70 percent when this happens.

Whether you encounter this situation depends on several factors. For example, if you're running for a long time in a very large storage pool, you're more likely to reach the threshold than if you're running in a smaller, restricted pool. You won't accumulate as many changed pages because the storage allocation functions are "stealing" the pages for memory requests for other data. Also, if the jobs don't run for a long time but instead finish and close the files, the changed pages no longer linger in main storage; they're written when the file is closed.

When you're using SMAPP, journaling the underlying physical file, and using a separate ASP for the journal receivers, you should also employ the CHGJRN RcvSizOpt(*RmvIntEnt) parameter to optimize performance. This parameter basically puts a "fork in the road" and routes the onerous (unnecessary for database recovery) SMAPP journal entries to a separate set of disk drives (still in the separate journal ASP). The system journaling functions split the use of the journal disk drives and set aside a separate set of disk arms for the journal receivers that will receive the SMAPP entries.

Hardware Options

Two of the most significant hardware performance enhancements in the history of the S/38 and AS/400 are the *n*-way processors and write-cached disk I/O processors (IOPs).

If you want optimum disk-write performance and you're using the older disk IOPs that have 4 MB of write cache, spread your journal receiver disk arms over as many of these IOPs as you can (no more than five arms per IOP). If you're using the 26 MB write cache disk IOPs, you can put more arms under the control of one IOP and achieve good results.

Spreading the receivers over multiple arms and using the 26 MB write cache provides a large amount of memory for the write-intensive journal receiver operations. Using the IOP's write cache to the fullest lets the job or task that issues a journal write see a very short disk-write response time (one or two milliseconds) before continuing processing.

Note that the output data record may stay in the IOP's write cache for several milliseconds (sometimes up to 200) before it's written. When this occurs, it's usually because there's other data in the cache that arrived earlier. Because each IOP write cache has a separately powered backup memory, you don't have a potential integrity problem.

Update Blocking — QJOSPEED PRPQ

So far, much of this discussion has focused on output-only write-add processing. There's another class of jobs that, for one reason or another, can't be changed (e.g., you don't have the source code, the proper people, the money, the time). For those of you who aren't already using journaling, Rochester has produced a chargeable PRPQ that allows *output blocking of journal data for update processing*. For more information, see the sidebar "Batch Journal Caching for OS/400".

FMTDTA Command (a.k.a. Sort)

Many application people I meet aren't familiar with the Format Data command, but they've usually heard of the function it performs. It's called Sort. In some application scenarios, you can achieve dramatic processing-time improvements by changing the physical sequence of the data to put it in the order of the most commonly used key(s). You can do this either by sorting the file

or reorganizing it into the sequence in which it is most often processed and then processing the file sequentially. Most application writers don't do this and consequently there is a lot of room for improvement, especially when you have large files in which every record has to be processed. sorting data ahead of time and processing it sequentially

Some users say that they're already using sequential processing, but upon investigating the files used in a sample job, Further investigation shows that only their logical files are being processed sequentially. The logical files read a record with key value *n*, then the record with key value (*n* + 1), and so on. Meanwhile, the data in the physical file records is scattered all over the ASP. Performance analysis of these jobs usually reveals a ratio of one physical read for every logical read (i.e, practically no blocking).

To see if this situation applies to your system, look at the application file processing. If the physical file can be sorted into the processing key sequence and read sequentially (using blocking), and you can use a separate file description for output, you may be able to further improve performance.

Applications generated by most of the 4GLs don't process physical files; for the most part they work exclusively with logical files and don't bother to sort the real data into the best sequence for processing performance. If the code works, that's the extent of it. Truly optimizing database performance isn't a consideration with 4GL-built code. By applying some of the techniques in this article, however, you may be able to improve their batch throughput. Beware, though, that if you change your 4GL generated code, there may be no guarantee of processing integrity.

That's a lot of information for you to absorb and understand! But if you're goal is to drive your processor to the maximum with your nightly, monthly, or quarterly long-running batch work, applying these methods to your system will help.

Sidebar: Batch Journal Caching for OS/400

by Larry Youngren, IBM Rochester

Batch Journal Caching for AS/400 is provided by database PRPQ 5799-BJC, which has been available since July 2000 for OS/400 V4 (and V5).

Batch Journal Caching can provide a significant performance improvement for batch environments that use journaling. It changes the handling of disk writes to achieve the maximum performance for journaled database operations. By caching journal writes in main memory, Batch Journal Caching can greatly reduce the impact of journaling on batch runtime by eliminating the delay while each journal entry is written to disk. This PRPQ is an ideal solution for customers with batch workloads who use journaling as part of a high-availability solution for replicating database changes to a backup system.

Though the Batch Journal Caching PRPQ is intended primarily for batch jobs, some interactive applications may also benefit. Applications that perform numerous database update/add/delete operations should see the greatest improvement. Applications that use commitment control will probably see less improvement because commitment control already performs some journal caching.

With normal, non-cached journaling in a batch environment, each database record that the batch job updates, adds, or deletes causes a new journal entry to be constructed in main memory. The batch job then waits for each new journal entry to be written to disk to assure recovery. This results in a lot of synchronous disk writes.

The Batch Journal Caching PRPQ lets you selectively enable a new variation of journal caching by letting most database operations avoid waiting for the synchronous write of journal entries to disk. The main memory cache can house 128 K, so a lot of journaled entries can be cached before a synchronous disk write ensues.

The Batch Journal Caching PRPQ can help you avoid the problems and costs associated with making application changes (such as adding commitment control) to improve performance in batch environments. It may be an ideal solution for a targeted backup system in a high-availability environment where you employ a never-ending apply job that replays database operations against a replica of your original database files.

For optimal journal performance, you should consider many factors other than the use of the Batch Journal Caching PRPQ, including the number and type of disk units and disk controllers, the amount of write cache, the placement of journal receivers in user ASPs, and application changes.

Technical Caveats

To use the Batch Journal Caching PRPQ, you must be running either OS/400 V4R4 with PTFs MF24293, MF24626, and SF63186 or V4R5 with PTFs MF24863, MF24866, MF24870, and SF63192. Note that this type of journaling is different than traditional journaling and can affect the recovery time for the associated database files. Because journal entries are temporarily cached in main memory, a few recent database changes that are still cached and not yet written to disk may not reach disk in the rare event of a system failure of such severity that the contents of main memory aren't preserved.

Further, this type of journaling may be unsuitable for interactive applications where

· single-system recovery (rather than dual-system replication) is the primary reason for using journaling

· it's unacceptable to lose even one recent database change in the rare event of a system failure that fails to preserve the contents of main memory.

The Batch Journal Caching PRPQ is intended primarily for situations where you're using journaling to enable database replication to a second system (e.g., for high-availability or business intelligence applications) under a heavy workload such as batch processing or heavy interactive work with applications that don't employ commitment control.

You can also selectively enable the PRPQ to optimize performance when running nightly batch workloads and then disable it each morning to optimize recoverability when running interactive applications. This approach can speed up some nightly batch jobs without sacrificing robust recovery for daytime interactive work.