Information Management software

Redpaper

Dan Weis
Paolo Bruni

DB2 for z/OS: Considerations on Small and Large Packages

Abstract

In this IBM® Redpaper, after some introductory background information on packages, we discuss the CPU utilization by packages related to two situations: when a small number of short-running SQL statements out of many SQL statements in a large package are executed and when a set of short-running SQL statements out of many different packages are executed. These two specific situations have become critical in some environments due to the performance difference after migration from DB2® V7 to V8. DB2 V8 has shown an increase in CPU associated with the major changes in functionalities. We show the measured CPU across V7, V8, and V9, highlight the improvement provided by V9, and provide some general tips on reducing CPU utilization by packages.

Notice that in this paper we refer to DB2 9 for z/OS® as DB2 V9 every time we compare it to other DB2 versions, such as V8 and V7, in the same section or diagram. This is done for the sake of consistency and ease of reading in the text and in the diagrams.

Disclaimer

Measurement data included in this presentation are obtained by the members of the DB2 performance department at the IBM Silicon Valley Laboratory.

The materials in this presentation are current as of April 2008 and are subject to enhancements at some future date, a new release of DB2, or a Programming Temporary Fix.

The information contained in this presentation has not been submitted to any formal IBM review and is distributed on an *as is* basis without any warranty either expressed or implied. The use of this information is a customer responsibility.

Background information

In this section we provide background information related to the application CPU time, the BIND PACKAGE options, the allocation and deallocation process of packages, and considerations on binding packages with RELEASE(COMMIT) versus RELEASE(DEALLOCATE). For more information, refer to the DB2 9 for z/OS Information Center available at the Web site:

http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp?topic=/com.ibm.db29.doc/db2prodhome.htm

Application CPU time

The cost in terms of class 2 (in DB2) CPU time for a DB2 application is basically the sum of the cost of each SQL statement used in the package as well as some extra CPU cost associated with the creation of the DB2 package environment. The package environment in DB2 consists of EDM storage allocation and initialization for a number of control block structures for the package that is established at the time the package is bound. There is also some CPU used at COMMIT for a package bound RELEASE(COMMIT), which generally releases resources including the package control block structures that are initialized at package allocation. The performance of the package allocation and deallocation is directly dependent on the number of SQL statements that are defined in the package. If the SQL statements that are executed in a package are very short running (transaction like), then the CPU time for package allocation/deallocation can be a very significant part of the total package CPU execution cost.

Bind package options

Generally, a BIND PACKAGE has options similar to the BIND PLAN, and often a package inherits the options from the PLAN.

The ACQUIRE and RELEASE options determine when to acquire and release resources that a program uses. ACQUIRE can do that at program allocation or when the resources are used (ALLOCATE or USE). RELEASE releases the resources either at each commit point or when the program terminates (COMMIT or DEALLOCATE). Basically, ACQUIRE and RELEASE control the duration of a table, partition, or tablespace lock held by, in most cases, static SQL statements in a plan or package. The options have no effect on page, row, LOB, or XML locks. Page and row locks are always removed at commit.

ACQUIRE was important when binding Database request modules (DBRMs) directly into plans. ACQUIRE(USE) tells DB2 to acquire the necessary resources to run an SQL statement when that statement is executed. This is the only behavior of a package. If you were dealing with a plan that had DBRMs bound into it directly, you could instead choose ALLOCATE. This acquires all the resources for all the SQL statements in all the DBRMs bound into that plan at the time the plan's thread is allocated. This gave a plan predictable performance because everything was allocated the same way all the time, but it could also be allocating resources for SQL statements that may never be executed in a particular plan execution. The result would be unnecessary extra cost and resource allocation. When binding a plan, ACQUIRE(ALLOCATE) must always be paired with RELEASE(DEALLOCATE).

The main difference is that no option exists for ACQUIRE for packages. ACQUIRE(USE) is always used by a package. At the local server the default for RELEASE is the value used by the plan that includes the package in its package list. At a remote server the default is COMMIT.

The RELEASE option can impact:

- Release of tablespace locks
- Reuse of EDM pool pages occupied by cursor tables (CT) and package tables (PT)
- Release of thread storage
- Resetting sequential detection and index look-aside information
- Destruction of procedures (IPROC, UPROC)

ACQUIRE(USE)/RELEASE(DEALLOCATE)

This combination results in the most efficient use of processing time in most cases.

- A table, partition, or tablespace used by the plan or package is locked only if it is needed while running.
- All tables or tablespaces are unlocked only when the plan terminates.
- The least restrictive lock needed to execute each SQL statement is used, with the exception that if a more restrictive lock remains from a previous statement, that lock is used without change.

RELEASE(DEALLOCATE) can increase the package or plan size, because additional items become resident in the package or plan, therefore requiring more virtual storage.

RELEASE (DEALLOCATE) also changes a few of the rules when it comes to releasing storage. For example, even if you choose to specify the DSNZPARM keyword CONTSTOR YES in an attempt to take advantage of storage contraction, RELEASE(DEALLOCATE) negates the entire process. A commit does not trigger CONTSTOR's processing. With RELEASE(DEALLOCATE), until the thread is actually deallocated, it just keeps getting larger.

RELEASE(DEALLOCATE) has no effect on packages that are run on a DB2 server through a DRDA® connection with a client system. The value also has no effect on dynamic SQL statements, which always use RELEASE(COMMIT), with one exception: When you use RELEASE(DEALLOCATE) and KEEPDYNAMIC(YES), and your subsystem is installed with YES for field CACHE DYNAMIC SQL on installation panel DSNTIP4, and the RELEASE(DEALLOCATE) option is honored for dynamic SELECT, INSERT, UPDATE, and DELETE statements.

The need to use RELEASE (DEALLOCATE) is generally decreasing. Prior to DB2 V8, it was often used in data sharing to minimize moving locks in and out the coupling facility and reducing XES contentions. With DB2 V8 data sharing's lock protocol 2 (NFM), this locking issue is no longer critical.

ACQUIRE(USE) / RELEASE(COMMIT)

This combination is the default combination and provides the greatest concurrency, but it requires more processing time if the application commits frequently.

- A table, partition, or tablespace is locked only when needed. That locking is important if the process contains many SQL statements that are rarely used or statements that are intended to access data only in certain circumstances.
- All tables and tablespaces are unlocked when:
  - In TSO

    Batch and CAF: An SQL COMMIT or ROLLBACK statement is issued, or your application process terminates.
  - In IMS™

    A CHKP or SYNC call (for single-mode transactions), a GU call to the I/O PCB, or a ROLL or ROLB call is completed.

– In CICS®

A SYNCPOINT command is issued.

– In RRS

A COMMIT or ROLLBACK statement is issued. To commit or roll back work when using RRSAF, use the SRRCMIT or SRRBACK commands, respectively. If the only recoverable resource being used by your application is within a single DB2 subsystem, then you can also use the DB2 COMMIT and ROLLBACK functions.

However, if the cursor is defined WITH HOLD, table or tablespace locks necessary to maintain the cursor position are held past the commit point.

- Table, partition, or tablespace locks are released at the next commit point unless the cursor is defined WITH HOLD.
- The least restrictive lock needed to execute each SQL statement is used, except when a more restrictive lock remains from a previous statement. In that case, that lock is used without change.

The disadvantage here is that this combination can increase the frequency of deadlocks. Because all locks are acquired in a sequence that is predictable only in an actual run, more concurrent access delays might occur.

Locks that are acquired for dynamic statements are held until one of the following events occurs:

- The application process commits.
- The application issues a PREPARE statement with the same statement identifier. Locks are released at the next commit point.
- The statement is removed from the cache because it has not been used. Locks are released at the next commit point.

KEEPDYNAMIC(NO/YES)

This determines whether DB2 keeps dynamic SQL statements after commit points.

- NO -specifies that DB2 does not keep dynamic SQL statements after commit points.
- YES specifies that DB2 keeps dynamic SQL statements after commit points.

If you specify KEEPDYNAMIC(YES), the application does not need to prepare an SQL statement after every commit point. DB2 keeps the dynamic SQL statement resulting from an explicit PREPARE (as opposed to EXECUTE IMMEDIATE[1]), until one of the following occurs:

- The application process ends.
- A rollback operation occurs.
- The application executes an explicit PREPARE statement with the same statement identifier.

If you specify KEEPDYNAMIC(YES) and the prepared statement cache is active (DSNZPARM CACHEDYN=YES), DB2 keeps a copy of the prepared statement in the cache. If the prepared statement cache is not active, DB2 keeps only the SQL statement string past a commit point. DB2 then implicitly prepares the SQL statement if the application executes an OPEN, EXECUTE, or DESCRIBE operation for that statement. If you specify KEEPDYNAMIC(YES), DDF server threads that are used to execute KEEPDYNAMIC(YES) packages will remain active. Active DDF server threads are subject to idle thread time-outs. If you specify KEEPDYNAMIC(YES), you must not specify REOPT(ALWAYS).

[1] KEEPDYNAMIC(YES) does not affect EXECUTE IMMEDIATE dynamic SQL. EXECUTE IMMEDIATE means prepare and execute once, not intending to keep the same prepared statement.

KEEPDYNAMIC(YES) and REOPT(ALWAYS) are mutually exclusive. However, you can use KEEPDYNAMIC(YES) with REOPT(ONCE).

The downside of KEEPDYNAMIC(YES) can be virtual storage usage below the bar, so you need to be cautious about setting this option.

Subsystem parameter CACHEDYN_FREELOCAL, in macro DSN6SPRM, added by PTF UK15493 in DB2 for z/OS V8, indicates whether DB2 can free local cached dynamic statements to relieve DBM1 below-the-bar storage constraint. CACHEDYN_FREELOCAL applies only when the KEEPDYNAMIC(YES) bind option is active for plans and packages. The default value is 0, which means that DB2 does not free cached dynamic statements to relieve high use of storage by dynamic SQL caching. If you specify 1, DB2 frees some cached dynamic statements to relieve high use of storage by dynamic SQL caching when the cached SQL statement pools have grown to a certain size.

Dynamic statement caching

Generally, the RELEASE option has no effect on dynamic SQL statements with one exception. When you use the bind options RELEASE(DEALLOCATE) and KEEPDYNAMIC(YES), and your subsystem is installed with YES for field CACHE DYNAMIC SQL on installation panel DSNTIP4, DB2 retains prepared SELECT, INSERT, UPDATE, and DELETE statements in memory past commit points. For this reason, DB2 can honor the RELEASE(DEALLOCATE) option for these dynamic statements. The locks are held until deallocation, or until the *commit* after the prepared statement is freed from memory, in the following situations:

- The application issues a PREPARE statement with the same statement identifier.
- The statement is removed from memory because it has not been used.
- An object that the statement is dependent on is dropped or altered, or a privilege needed by the statement is revoked.
- RUNSTATS is run against an object that the statement is dependent on.

Package allocation/deallocation process

The first time that a package statement is referenced in an application, DB2 reads the package directory into the EDM pool as part of the package skeleton (SKPT). The SKPT is the cached copy of the package in the EDM pool. The package directory contains information about how many statements are included and their size. The package directory is created when the package is bound. A package has a separate section for each statement that is defined in the package. There is also a specific package section that is associated with all of the statements in the package. This is called the package table (PT).

After the package directory is read into the EDM pool as part of the package SKPT, the PT is then read into the EDM pool also and connected to the SKPT. The PT is then copied into a different area of the EDM pool for the thread to use and is initialized. The initialization process of copying the storage, address relocation, and establishing control blocks implies some CPU cost for every statement that is defined in the package, including statements that may not be executed. Future initial references to the package will obtain the PT from the skeleton (SKPT). The size of the PT includes some fixed cost as well as some variable amount that is proportional to the number of statements defined in the package.

Next, the referenced statement section is loaded into the EDM pool, attached to the SKPT, and a copy made and initialized for the thread. Then the statement is executed.

As COMMIT occurs, in case of BIND with RELEASE(COMMIT), the statement section (if no cursor WITH HOLD option is used) is closed, cleaned up, and released from the EDM pool. If all the package statement sections are freed at COMMIT then the PT will be released, too.

In the case that RELEASE(DEALLOCATE) or dynamic SQL from explicit PREPARE and bind option KEEPDYNAMIC(YES) is in effect, the section is cleaned up for possible reuse and the package remains allocated.

COMMIT versus DEALLOCATE

We have seen that RELEASE has two values: DEALLOCATE and COMMIT. RELEASE(COMMIT) releases the above resources at commit time, while RELEASE(DEALLOCATE) releases resources at thread deallocation.

Usually, RELEASE(COMMIT) is the correct choice. In a high-performance online transaction, there is probably only a short unit of work being executed anyway, which makes RELEASE(COMMIT) a preferred choice. In other environments, however, such as batch and distributed transactions, you need to consider the implications of this choice.

In a long-running batch job, RELEASE(COMMIT) can let a utility *sneak* in when a commit occurs. This can be good or bad, depending on your planned result. To prevent interruption by a subsequent job being submitted, RELEASE(DEALLOCATE) could be used. It is usually associated with CICS protected threads. The DB2 environment should be designed for high-performance with the use of long-running, persistent CICS-DB2 threads bound with RELEASE DEALLOCATE for high-volume CICS transactions or terminal and non-terminal-driven transactions with high commit rates.

However, changing the definitions of plans or packages to use RELEASE DEALLOCATE should be done only after the high-profile CICS transactions or transactions with high commit rates have been identified and steps have been taken to redefine these transactions as protected threads. You would not accomplish much if you defined a pool thread with RELEASE (DEALLOCATE). Effective CICS-DB2 thread reuse is a prerequisite to benefiting from using RELEASE (DEALLOCATE). Similar considerations apply for IMS *wait for input* type of transactions.

The advantages of RELEASE(DEALLOCATE) include:

- It can potentially reduce XES Lock/Unlock activity for parent L-locks (IS/IX) and can be a major weapon in reducing XES global lock contention in a data-sharing environment (for versions prior to DB2 V8 NFM).

  Situations with X, S, or U table or tablespace locks mean limited concurrency for any application accessing those tables and tablespaces.
- Locking traffic can be avoided by specifying CURRENTDATA(NO) and ISOLATION CS or specifying ISOLATION UR. However, once a DB2 (or data sharing group) is migrated to V8 New Function Mode, locking protocol level 2 is enabled, so RELEASE(DEALLOCATE) is not as critical to reduce XES locking.
- DB2 will not reset index look-aside cache and will not reset dynamic prefetch (sequential detection) counters.
- Counters associated with the use of INSERT procedures (IPROC) and UPDATE procedures (UPROCs) are not reset, nor are tablespace intent locks.
- There is greater potential of reducing CPU usage per transaction for simple, high-volume transactions because resources will not be reacquired multiple times.

- When dynamic statement cache is not used, RELEASE(DEALLOCATE) has no affect on dynamic SQL.

The disadvantages of using RELEASE(DEALLOCATE) include:

- Locks will be held for an excessive time. The most sensitive to this issue could be utilities trying to access tablespaces with many locks. However, this could also interfere with DDL changes, LOCK TABLE statements, BIND/REBIND operations, and possibly GRANTs and REVOKEs. Restrictive locks might be held a long time.
- More EDM pool storage could be used because more strings are kept in the pool for a longer time. Cursor tables are not released at commit and could grow to a significant size. Monitoring of the EDM pool might be required.
- In V8, the traditional EDM pool is still below the 2 GB bar and contains the CT, PT, SKCT, and SKPT structures. Separate pools now exist for both the DBDs and dynamic statement cache. Both pools are allocated above the 2 GB bar. In V9, the EDM pool contains only CT and PT storage. The SKCT and SKPT structures are moved to a separate pool above the bar.
- When using created temporary tables, space for logical work files is not released until the thread is deallocated.
- When using dynamic statement caching (DSNZPARM CACHEDYN=YES) and specifying KEEPDYNAMIC(YES), locks are held until deallocation or the first commit point that follows the removal of the statement from the local cache. This means that locks could be held much longer than anticipated.
- For DB2 V7 installations, using dynamic statement caching can have a significant impact on the EDM pool, if the dynamic statement cache is defined in the EDM pool and not in a data space. DB2 V8 and V9 use the EDMSTMTC pool above the bar for the dynamic statement cache and avoid this.
- Application programmers have to be aware of the effects of using SQL LOCK TABLE:
  - SQL LOCK TABLE will be held across a commit.
  - Mass delete locks are held until thread termination.

  Examine the AT(COMMIT) option for STOP DATABASE, which allows a thread bound with RELEASE(DEALLOCATE) to be interrupted at the next commit point. A last resort-approach to interrupting a long-running transaction with RELEASE(DEALLOCATE) is the CANCEL THREAD command.

RELEASE(COMMIT) should be the assumed development default. Application programmers and DBAs should justify any decisions to use RELEASE(DEALLOCATE). You should always specify the RELEASE keyword on a package to ensure that you are getting what you expect, and you should specify RELEASE(COMMIT) on the plan. Specifying RELEASE(COMMIT) on the plan prevents using RELEASE(DEALLOCATE) by accident at the package level, should the RELEASE keyword not be specified in the bind package statement.

Impact of single statement execution in packages with a large number of SQL statements

The allocation and deallocation process for the package includes some initialization for every single statement in the package whether it is used or not. If a package has a large number of statements and only one statement is executed, for instance, followed by a COMMIT, there can be considerable cost due to all of the statements in the package that are not used when the package is freed at COMMIT.

RELEASE(COMMIT)

As an example, a relative comparison with various package sizes from 2 to 2002 statements per package is shown in Figure 1 across the three current DB2 versions: V7, V8, and V9.

The class 2 CPU measurement is an average for several invocations of a plan calling a single package of the various sizes 5,000 times.

The application package has only a static SELECT followed by COMMIT. This is typically the worst case, since any more expensive SQL statement (such as INSERT) will be less impacted by the package management cost.

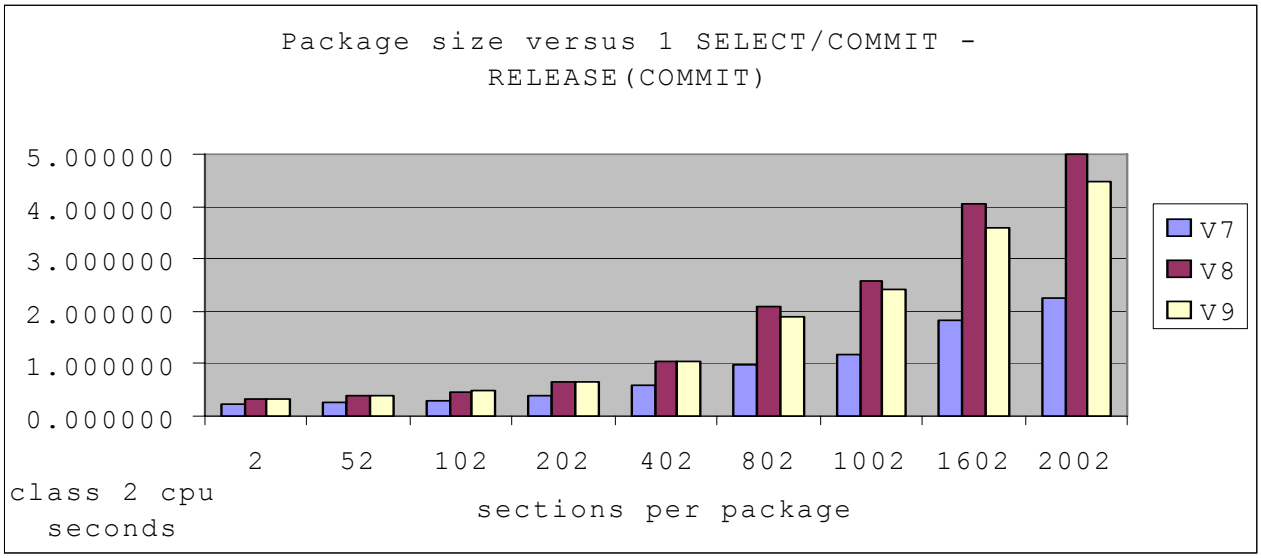

Figure 1 Influence of package size by DB2 version with RELEASE(COMMIT)

With DB2 V7, we can observe that a 202 section package takes about twice as long as a 2 section package, and a 2002 section package about 10 times longer.

With DB2 V8, the dependency on package size is more critical: the largest package of 2002 sections (SQL statements) takes about 20 times longer in class 2 CPU over the same identical process in a 2 section package. The primary reasons that V8 takes a longer CPU time than V7 is that V8 is using 64-bit instructions to 32-bit instructions in V7, as well as additional functions in V8.

With DB2 V9, this large package size performance issue with RELEASE(COMMIT) still exists, though it is slightly smaller.

Notice that this measurement is attempting to show the typical worst case of a performance impact of package size for a very short-running SQL. The percentage impact will go down as the number of SQL statements executed per package goes up or the cost of each SQL statement executed goes up.

RELEASE(DEALLOCATE)

You need to balance the CPU savings of RELEASE(DEALLOCATE) with other needs for storage, and the CPU needs to manage the EDM and thread storage. Use RELEASE(DEALLOCATE) only for the short-running, very high transaction rate applications. If you use RELEASE(DEALLOCATE), then you need to manage the threads more, or the inability to run utilities and costs for storage and storage management will be larger than the

savings. For batch applications with a very high commit rate, the better technique is to increase the time between COMMIT to a few seconds and to keep RELEASE(COMMIT), rather than using RELEASE(DEALLOCATE). In any case, monitor and manage the long-running units of recovery, deadlocks, and time outs. If you use RELEASE(DEALLOCATE), then you will need to perform more monitoring for locking and virtual storage (especially the EDM pool).

Running the same test with RELEASE(DEALLOCATE) demonstrates the cost of this large package size. See Figure 2. Note that Figure 2 is a different scale from Figure 1 on page 8 in order not to obscure the differences across versions within RELEASE(DEALLOCATE).

Figure 2 Influence of package size by DB2 Version with RELEASE(ALLOCATE)

With DB2 V7, we can observe that the CPU time for a 202 section package is about the same as a 2 section package, while it takes 3 times longer for a 2002 section package.

With DB2 V8, the dependency on package size is still critical: a 202 section package takes about 2 times longer than a 2 section package, while the largest package of 2002 sections is about 10 times the class 2 CPU over the same identical process in a 2 section package.

With DB2 V9 this large package size performance issue with RELEASE(DEALLOCATE) is resolved. DB2 V9 with RELEASE(DEALLOCATE) actually performs much better than even V7 at large package sizes because the package cost has been reduced.

Comparing with Figure 1 on page 8, the observation is that running the same test with RELEASE(DEALLOCATE) performs better than RELEASE(COMMIT). The chart demonstrates that the cost of larger package sizes is reduced when compared to RELEASE(COMMIT), but it still has an increase on CPU utilization at COMMIT directly related to the package size.

Besides using RELEASE(DEALLOCATE), the repeated package cost in each commit scope can be mitigated if the package has a open and held cursor (CURSOR WITH HOLD) at the time of the COMMIT. If a held cursor is open at COMMIT then the package is not freed even with RELEASE(COMMIT).

KEEPDYNAMIC(NO) and KEEPDYNAMIC(YES) with RELEASE(COMMIT)

For applications that include dynamic SQL, another method that may mitigate the problem of a large package size cost at allocation and deallocation is to bind it with KEEPDYNAMIC(YES) even if the application has no logic to take advantage of KEEPDYNAMIC(YES). Package sections that have an explicit EXEC SQL PREPARE in the commit scope will also not be freed at commit, as the referenced section will prevent the freeing of the package. Note that this will work for explicit PREPARE statements, but not for EXECUTE IMMEDIATE, which is not affected by KEEPDYNAMIC(YES). Note also that there can be some small cost at COMMIT for KEEPDYNAMIC(YES), so KEEPDYNAMIC(NO) may still perform best where it is not needed, such as with a package with all static SQL.

The chart in Figure 3 shows how the various sized packages are affected by a single PREPARE/OPEN/FETCH/CLOSE/COMMIT sequence with KEEPDYNAMIC(NO) and then again with KEEPDYNAMIC(YES), across DB2 V7, V8, and V9. In each of these tests the application invokes a single package 5,000 times and the class 2 CPU measurement is an average of several of these application invocations.

For this application test, all statements are in the global cache, so the PREPARE cost is *short prepares* only and most closely simulates the performance of static SQL. The application is run once with a 1 Gigabyte EDM statement cache pool so that *all* statements are in the global cache and there are 0 full prepares. Full prepares would add much extra cost to make the performance differences between small and large packages executing only a few statements less noticeable, while the objective here is to show what happens with high performance and very short-running transaction/SQL.

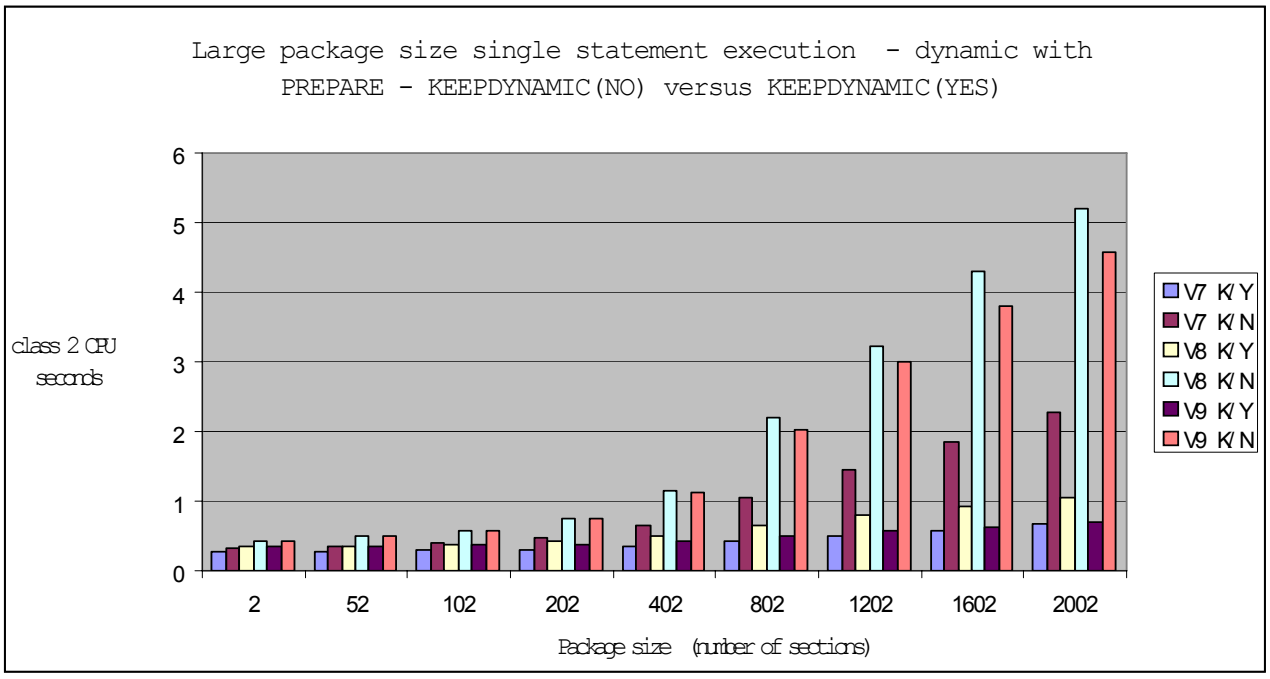

Figure 3 Influence by package size for KEEPDYNAMIC(NO) and (YES) across DB2 V7, V8, and V9

KEEPDYNAMIC(YES) has an effect similar to an open held cursor. That is, if a PREPARE is processed for a package section, then because of KEEPDYNAMIC(YES), the section (and thus the package) will not be freed at COMMIT but will remain allocated. This can help reduce the rate of increase in package cost significantly, even if the application has no logic to take advantage of KEEPDYNAMIC(YES). Even with KEEPDYNAMIC(YES) (K/Y in Figure 3), the cost for the statement execution still increases with the package size. But with

KEEPDYNAMIC(NO) the package may be freed and reallocated on a subsequent execution, which has a very high relative cost.

With respect to DB2 versions, V8 shows the highest values and most variation with the increasing number of sections. For V9, the result is very similar to V8.

Impact of statements executed in separate multiple packages versus one package

Another instance of the cost for package allocation and deallocation can be seen when the several short-running SQL statements in a single commit scope are dispersed into several packages instead of one. When statements are executed from multiple packages in a commit scope, there is package cost for each package. Combining *frequently occurring* statements into one package will reduce the package cost to a single package. In this case it may even be worthwhile to replicate a very short-running statement into multiple packages rather than have it execute from a package by itself.

Figure 4 and Figure 5 on page 12 show the relative performance of 10 very short-running SQL statements followed by a COMMIT. They are run from 10 packages versus when run from 1 package. In this application test, the commit scope is 10 SELECT, either from the same or different packages, as noted. The average class 2 CPU time is measured for a number of 500 sequences (of 10 SELECTs and 1 COMMIT).

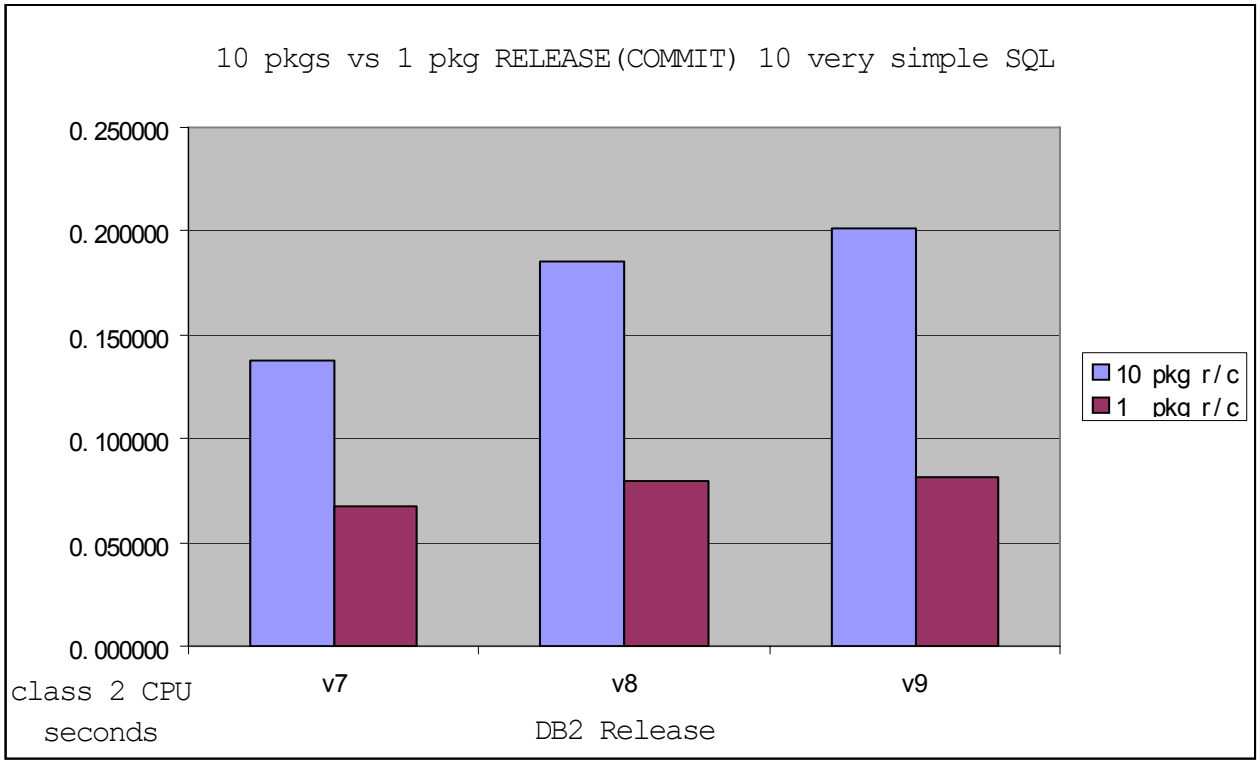

Figure 4 Spreading 10 SQL statements into 10 packages - RELEASE(COMMIT)

When statements are spread among several packages, the package allocation and deallocation cost occurs for all of them unless they have one of the exceptions previously noted. In this simple case, using 10 packages has twice the cost of 1 package for all three versions of DB2. RELEASE(COMMIT) is the big factor here.

In Figure 5 we show the measurements for this application when using static SQL bound with RELEASE(DEALLOCATE). For the RELEASE(DEALLOCATE) case there are two observations:

- Multiple packages have higher overhead that one package.
- Multiple packages have increased overhead in V8 when compared to V7.

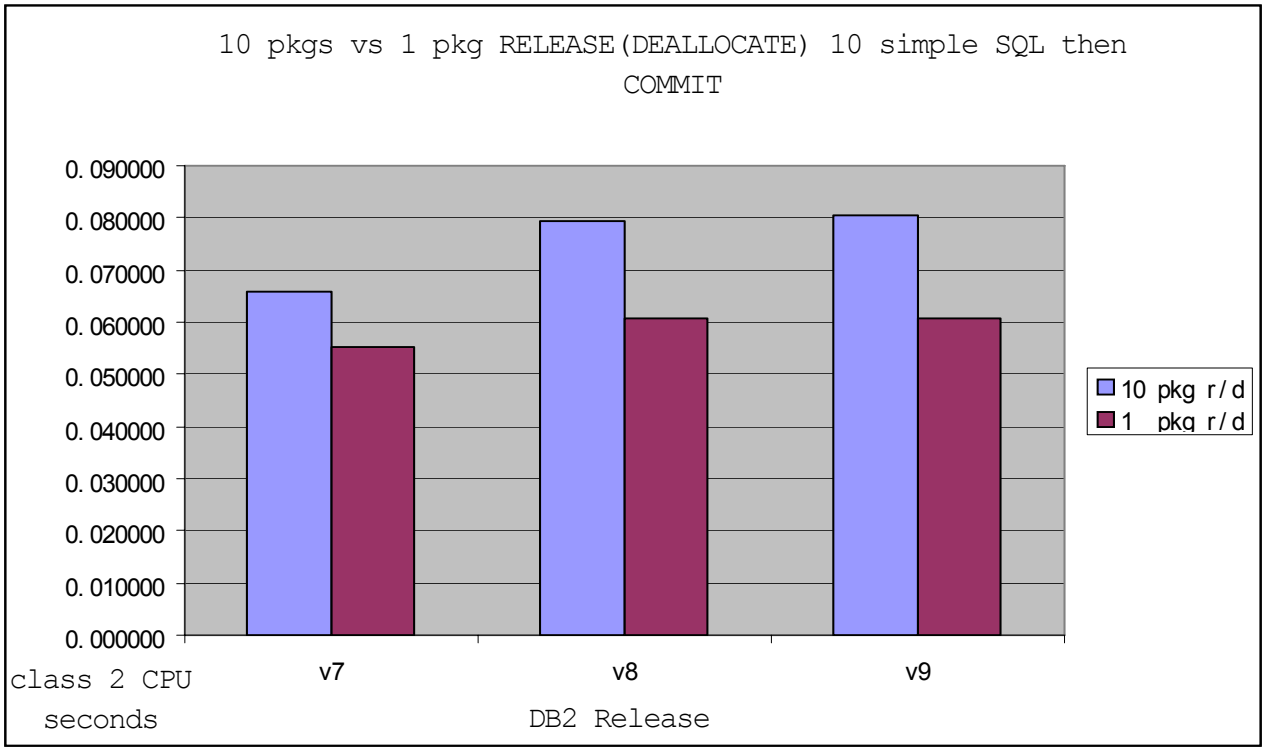

Figure 5 Spreading 10 SQL statements into 10 packages - RELEASE(DEALLOCATE)

Accounting trace classes 1, 2, 3, 7, and 8 were used to obtain package CPU.

Note: Reporting on DB2 accounting class 7 and 8 data

To monitor packages you generally need to start an accounting trace and collect data for classes 1, 2, 3, 7, and 8. Both elapsed time and CPU time will be collected for each class. You can also sporadically activate CLASS(10), introduced by V8 APAR PK28561 (UK18090), when you need to collect package details.

The accounting classes and descriptions of data collected are:

- 1 - CPU time and elapsed time in application, at plan level
- 2 - CPU time and elapsed time in DB2, at plan level
- 3 - Elapsed time due to suspensions, at plan level
- 7 - CPU time and elapsed time in DB2, at package level
- 8 - Elapsed time due to suspensions, at package level
- 10 - Detailed information about locks, buffers, and SQL statistics at the package level, additionally collected in IFCID 239.

See also informational APAR II14421.

What about DBRMs

Users may still have plans bound from DBRMs. Figure 6 shows the average class 2 CPU time for one plan invocation that executes 5,000 times a single statement in the DBRM/package followed by a COMMIT. Actually, all of the tests are each an average of 20 plan invocations (5,000 statement/commit sequences each).

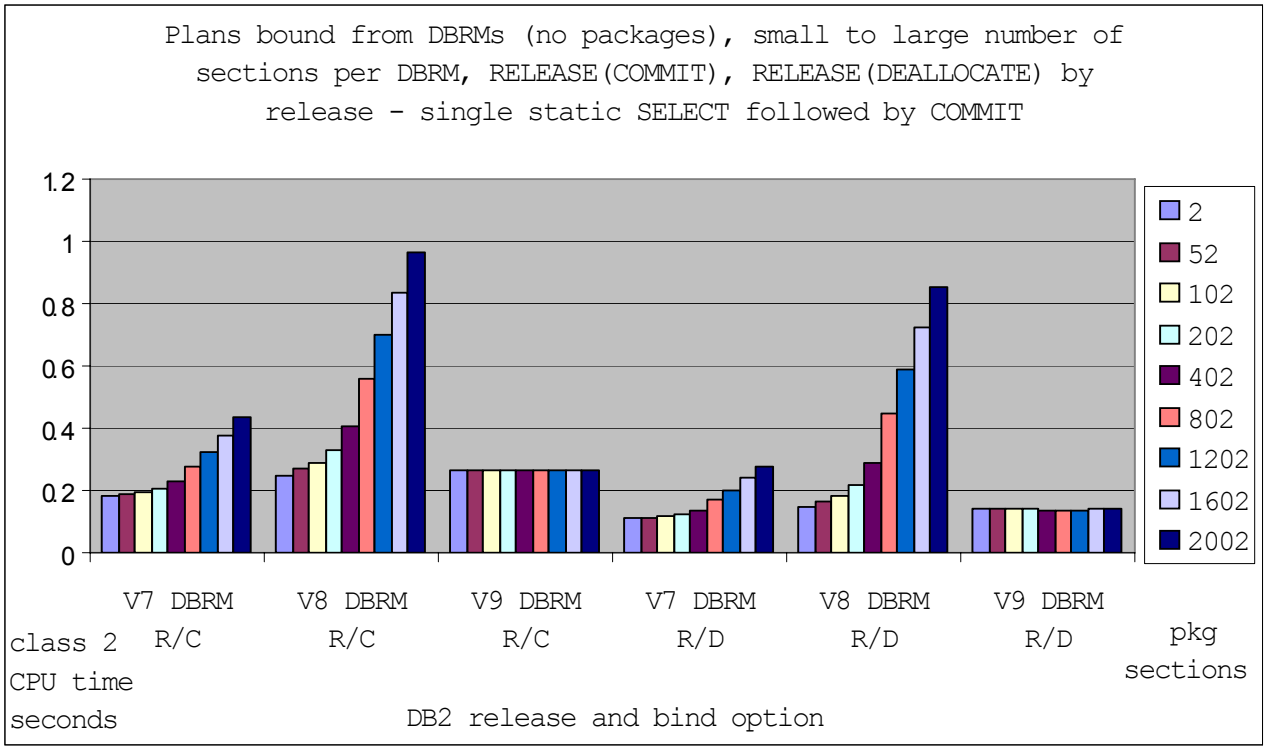

Figure 6 Influence for DBRM-bound plans for RELEASE COMMIT or DEALLOCATE across DB2 V7, V8, and V9

In Figure 6 the data is grouped by DB2 Version (V7, V8, and V9) and the RELEASE bind option. Each group contains the package sizes, smallest to largest.

V8 shows the highest CPU costs associated with package size for both RELEASE(COMMIT) and RELEASE(DEALLOCATE).

V7 has a similar shape but smaller values than V8 (about half) and higher percentage reduction with RELEASE DEALLOCATE.

V9 is about constant with package size variation for both RELEASE options, with DEALLOCATE about half of RELEASE(COMMIT).

V9 generally performs well with large DBRMS (any BIND option) and performs well with RELEASE(DEALLOCATE) with large packages, and not as well with small packages.

Conclusions on measurements

Application performance for very short-running SQL can be affected by cost of package processing. If the application has only static SQL statements and has many SQL but is only executing a few and is bound RELEASE(COMMIT), then the performance cost can be an order of magnitude larger than that of the same package without the unused SQL.

The package CPU cost may be individually very small, but if the package is executed a very large number of times, then the total CPU time for the package cost becomes expensive.

Recommendations

To determine whether this problem might arise in your environment, check the number of sections in your packages. A way to find packages with a large number of sections is to issue the query:

```
SELECT SUBSTR(NAME, 1, 8) AS NAME,
       MAX(SECTNOI) AS MAX_SECTNO
  FROM SYSIBM.SYSPACKSTMT
 GROUP BY NAME
 HAVING MAX(SECTNOI) > 200
 ORDER BY 2 DESC;
```

Precise package cost is not directly shown by the instrumentation, but using a performance trace class (2, 3) you can identify SQL time and approximately calculate the package cost as the difference between package class 7 CPU time and the sum of the class 7 CPU for the SQL statements in the package.

A good indicator, but not showing the total of the package cost, is the COMMIT CPU duration (IFCID 88-89). If the CPU cost of the SQL is relatively high then the package cost may be relatively insignificant. This issue is most pronounced only when there are very short-running (even non I/O) SQL. You can also use an accounting report to note SQL/COMMIT activity.

When a small number of short-running SQL statements, out of many SQL statements in a package, are executed, or when a set of short-running SQL statements is executed out of many different packages, one of the following tuning actions can be considered to reduce the increase in transaction CPU time (anywhere between no improvement and a 10 time improvement in the cases shown):

- Use the BIND option RELEASE(DEALLOCATE), instead of RELEASE(COMMIT), assuming that virtual storage usage is not constrained.

  Additional significant improvement is available in DB2 9 for z/OS, such that the performance of small and large sized packages using RELEASE(DEALLOCATE) is much closer.

- Use CURSOR WITH HOLD.

  If the cursor is held open across the commit. If the cursor is closed, the statement section will be released with RELEASE(COMMIT).[2]

- Use BIND with KEEPDYNAMIC(YES) and explicit PREPARE (MAXKEEPD=0) with RELEASE(COMMIT), if dynamic SQL.

  For the use of dynamic statement caching see *DB2 for z/OS and OS/390 : Squeezing the Most Out of Dynamic SQL*, SG24-6418.

- Redesign the application to reduce the number of SQL statements in a package to contain only those typically executed in a given package execution. Also try to keep those SQL statements typically executed in a transaction in one package.

[2] The biggest part of the package overhead at COMMIT/reallocation is actually establishing the PT. Just keeping one section allocated prevents the PT from being released. So one could theoretically design a 50 statement package to have 1 statement have an open/held cursor on SYSIBM.SYSDUMMY1 for the duration so the statements could use RELEASE(COMMIT) behavior for all the other sections but avoid the huge (in V8 and V9) cost of reallocating and initializing the PT.

The team that wrote this IBM Redpaper

This paper was produced by specialists working at the Silicon Valley Lab, San Jose California.

Dan Weis is a member of the DB2 for z/OS performance department in IBM Silicon Valley Laboratory. For most of his 39 years with IBM, Dan has worked on DB2 for z/OS, first in service support, then on performance.

Paolo Bruni is an ITSO Project Leader responsible for IBM Redbooks® publications on DB2 for z/OS and DB2 tools, and is located in the IBM Silicon Valley Laboratory. For most of his 39 years with IBM, in Italy, the UK, and the USA, Paolo has worked on database systems and specialized in performance.

Acknowledgements

Many thanks are owed to the reviewers of this Redpaper: Tammie Dang, Roger Miller, and Akira Shibamiya.

References

Refer to the following publications for more information:

- *DB2 for z/OS and OS/390 Version 7 Performance Topics*, SG24-6129
- *DB2 9 for z/OS Performance Topics*, SG24-7473
- *DB2 for z/OS and OS/390 : Squeezing the Most Out of Dynamic SQL*, SG24-6418
- *DB2 Version 9.1 for z/OS Application Programming and SQL Guide*, SC18-9841-01
- *DB2 Version 9.1 for z/OS SQL Reference*, SC18-9854-03
- *DB2 Version 9.1 for z/OS Command Reference*, SC18-9844-01

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

This document REDP-4424-00 was created or updated on July 14, 2008.

Send us your comments in one of the following ways:
- Use the online **Contact us** review Redbooks form found at:
  ibm.com/redbooks
- Send your comments in an email to:
  redbooks@us.ibm.com
- Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400 U.S.A.

Trademarks