

SECTION 3: CHAPTERS 7-10

CA Performance Handbook

for DB2 for z/OS

About the Contributors from Yevich, Lawson and Associates Inc.

DAN LUKSETICH is a senior DB2 DBA. He works as a DBA, application architect, presenter, author, and teacher. Dan has over 17 years working with DB2 as a DB2 DBA, application architect, system programmer and COBOL and BAL programmer — working on major implementations on z/OS, AIX, and Linux environments.

His experience includes DB2 application design and architecture, database administration, complex SQL and SQL tuning, performance audits, replication, disaster recovery, stored procedures, UDFs, and triggers.

SUSAN LAWSON is an internationally recognized consultant and lecturer with a strong background in database and system administration. She works with clients to help development, implement and tune some of the world's largest and most complex DB2 databases and applications. She also performs performance audits to help reduce costs through proper performance tuning.

She is an IBM Gold Consultant for DB2 and z/Series. She has authored the IBM 'DB2 for z/OS V8 DBA Certification Guide', 'DB2 for z/OS V7 Application Programming Certification Guide' and 'DB2 for z/OS V9 DBA Certification Guide' — 2007. She also co-authored several books including 'DB2 High Performance Design and Tuning' and 'DB2 Answers' and is a frequent speaker at user group and industry events. (Visit DB2Expert.com)

About CA

CA (NYSE: CA), one of the world's largest information technology (IT) management software companies, unifies and simplifies the management of enterprise-wide IT for greater business results. Our vision, tools and expertise help customers manage risk, improve service, manage costs and align their IT investments with their business needs. CA Database Management encompasses this vision with an integrated and comprehensive solution for database design and modeling, database performance management, database administration, and database backup and recovery across multiple database systems and platforms.

Table of Contents

About This Handbook

Originally known as “The DB2 Performance Tool Handbook” by PLATINUM Technologies (PLATINUM was acquired by Computer Associates in 1999), this important update provides information to consider as you approach database performance management, descriptions of common performance and tuning issues during design, development, and implementation of a DB2 for z/OS application and specific techniques for performance tuning and monitoring.

- **Chapter 1:** Provides a overview information on database performance tuning
- **Chapter 2:** Provides descriptions of SQL access paths
- **Chapter 3:** Describes SQL tuning techniques
- **Chapter 4:** Explains how to design and tune tables and indexes for performance
- **Chapter 5:** Describes data access methods
- **Chapter 6:** Describes how to properly monitor and isolate performance problem
- **Chapter 7:** Describes DB2 application performance features
- **Chapter 8:** Explains how to adjust subsystem parameters, and configure subsystem resources for performance
- **Chapter 9:** Describes tuning approaches when working with packaged applications
- **Chapter 10:** Offers tuning tips drawn from real world experience

Who Should Read This Handbook

The primary audiences for this handbook are physical and logical database administrators. Since performance management has many stakeholders, other audiences who will benefit from this information include application developers, data modelers and data center managers. For performance initiatives to be successful, DBAs, developers, and managers must cooperate and contribute to this effort. This is because there is no right and wrong when it comes to performance tuning. There are only trade-offs as people make decisions about database design and performance. Are you designing for ease of use, minimized programming effort, flexibility, availability, or performance? As these choices are made, there are different organizational costs and benefits whether they are measured in the amount of time and effort by the personnel involved in development and maintenance, response time for end users, or CPU metrics.

This handbook assumes a good working knowledge of DB2 and SQL, and is designed to help you build good performance into the application, database, and the DB2 subsystem. It provides techniques to help you monitor DB2 for performance, and to identify and tune production performance problems.

Page intentionally left blank

Application Design and Tuning for Performance

While the majority of performance improvements will be realized via proper application design, or by tuning the application, nowhere else does “it depends” matter most than when dealing with applications. This is because the performance design and tuning techniques you apply will vary depending upon the nature, needs, and characteristics of your application. This chapter will describe some general recommendations for improving performance in applications.

Issuing Only the Necessary Statements

The best performing statement is the statement that never executes. The first thing that you have to ask yourself is “is this statement necessary?” or “have I already read that data before?” One of the major issues with applications today is the quantity of SQL statements issued per transaction. The most expensive thing you can do is to leave your allied address space (DB2 distributed address space for remote queries) to go to the DB2 address spaces for a SQL request. You need to avoid this whenever possible.

Caching Data

If you are accessing code tables for validating data, editing data, translating values, or populating drop-down boxes then those code tables should be locally cached for better performance. This is particularly true for any code tables that rarely change (if the codes are frequently changing then perhaps it’s not a code table). For a batch COBOL job for example read the code tables you’ll use into in-core tables. For larger code tables you can employ a binary or quick search algorithm to quickly look up values. For CICS applications set up the code in VSAM files, and use them as CICS data tables. This is much faster than using DB2 to look up the value for every CICS transaction. The VSAM files can be refreshed on regular intervals via a batch process, or on an as-needed basis. If you are using a remote application, or Windows based clients you should read the code tables once when the application starts, and cache the values locally to populate various on screen fields, or to validate the data input on a screen.

If other situations, and especially for object or service oriented applications you should always check if an object has already been read in before reading it again. This will avoid blindly and constantly rereading the data every time a method is invoked to retrieve the data. If you are concerned that an object is not “fresh” and that you may be updating old data then you can employ a concept call optimistic locking. With optimistic locking you don’t have to constantly reread the object (the table or query to support the object). Instead you read it once, and when you go to update the object you can check the update timestamp to see if someone else has updated the object before you. This technique is described further in the locking section of this chapter.

Utilizing DB2 for System Generated Keys

One of the ways you can reduce the quantity of SQL issued, as well as a major source of locking contention in your application is to use DB2 system generated key values to satisfy whatever requirements you may have for system generated keys.

Traditionally people have used a “next-key” table, which contained one row of one column with a numeric data type, to generate keys. This typically involved reading the column, incrementing and updating the column, and then using the new value as a key to another table. These next-key tables typically wound up being a huge application bottleneck.

There are several ways to generate key values inside DB2, two of which are identity columns (DB2 V7, DB2 V8, DB2 9) and sequence objects (DB2 V8, DB2 9). Identity columns are attached to tables, and sequence objects are independent of tables. If you are on DB2 V7 then your only choice of the two is the identity column. However, there are some limitations to the changes you can make to identity columns in DB2 V7 so you are best placing them in their own separate table of one row and one column, and using them as a next key generator table.

The high performance solution for key generation in DB2 is the sequence object. Make sure that when you use sequence objects (or identity columns) that you utilize the CACHE and ORDER settings according to your high performance needs. These settings will impact the number of values that are cached in advance of a request, as well as whether or not the order the values are returned is important. The settings for a high level of performance in a data sharing group would be, for example, CACHE 50 NO ORDER.

When using sequence objects or identity columns (or even default values, triggers, ROWIDs), the GENERATE_UNIQUE function, the RAND function, and more) you can reduce the number of SQL statements your application issues by utilizing a SELECT from a result table (DB2 V8, DB2 9). In this case that result table will be the result of an insert. In the following example we use a sequence object to generate a unique key, and then return that unique key back to the application:

```
SELECT ACCT_ID
FROM FINAL TABLE
  (INSERT INTO UID1.ACCOUNT (ACCT_ID,NAME, TYPE, BALANCE)
  VALUES
  (NEXT VALUE FOR ACCT_SEQ,'Master Card','Credit',50000))
```

Check the tips chapter for a cool tip on using the same sequence object for batch and online key assignment!

Avoiding Programmatic Joins

If you are using a modern generic application design where there is an access module for each DB2 table, or perhaps you are using an object oriented or service oriented design, then you are using a form of programmatic joins. In almost every single situation in which a SQL join can be coded instead of a programmatic join, then the SQL join will outperform the programmatic join.

There is a certain set of trade-offs that have to be considered here:

- Flexible program and table access design versus improved performance for SQL joins
- The extra programming time for multiple SQL statements to perform single table access when required or joins when required versus the diminished performance of programmatic joins
- The extra programming required to add control breaks in the fetch loop versus the diminished performance of programmatic joins

DB2 does provide a feature known as an INSTEAD OF trigger (DB2 9) that allows somewhat of a mapping between the object world and multiple tables in a database. You can create a view that joins two tables that are commonly accessed together. Then the object based application can treat the view as a table. Since the view is on a join, it is a read-only view. However, you can define an INSTEAD OF trigger on that view to allow INSERTs, UPDATEs, and DELETEs against the view. The INSTEAD OF trigger can be coded to perform the necessary changes to the tables of the join using the transition variables from the view.

In our tests of a simple two table SQL join versus the equivalent programmatic join (FETCH loop within a FETCH loop in a COBOL program), the two table SQL join used 30% less CPU than the programmatic join.

Multi-Row Operations

In an effort to reduce the quantity of SQL statements the application is issuing DB2 provides for some multi-row operations. This includes:

- Multi-row fetch (DB2 V8, DB2 9)
- Multi-row insert (DB2 V8, DB2 9)
- MERGE statement (DB2 9)

These are in addition to the possibility of doing a mass INSERT, UPDATE, or DELETE operation.

MULTI-ROW FETCH Multi-row fetching gives us the opportunity to return multiple rows (up to 32,767) in a single API call with a potential CPU performance improvement somewhere around 50%. It works for static or dynamic SQL, and scrollable or non-scrollable cursors. There is also support for positioned UPDATEs and DELETEs. The sample programs DSNTEP4 (which is DSNTEP2 with multi-row fetch) and DSNTIAUL also can exploit multi-row fetch.

There are two reasons to take advantage of the multi-row fetch capability:

1. To reduce the number of statements issued between your program address space and DB2.
2. To reduce the number of statements issued between DB2 and the DDF address space.

The first way to take advantage of multi-row fetch is to program for it in your application code. The second way to take advantage of multi-row fetch is in our distributed applications that are using block fetching. Once in compatibility mode in DB2 V8 the blocks used for block fetching are built using the multi-row capability without any code change on our part. This results in great savings for our distributed SQLJ applications. In one situation the observed benefit of this feature was when a remote SQLJ application migrated from DB2 V7 to DB2 V8 it did not have a CPU increase.

Coding for a multi-row fetch is quite simple. The basic changes include:

- Adding the phrase “WITH ROWSET POSITIONING” to a cursor declaration
- Adding the phrases “NEXT ROWSET” and “FOR n ROWS” to the FETCH statement
- Changing the host variables to host variable arrays (for COBOL this is as simple as adding an OCCURS clause)
- Placing a loop within your fetch loop to process the rows

These changes are quite simple, and can have a profound impact on performance. In our tests of a sequential batch program the use of 50 row fetch (the point of diminishing return for our test) of 39 million rows of a table reduced CPU consumption by 60% over single-row fetch. In a completely random test where we expected on average 20 rows per random key, our 20 row fetch used 25% less CPU than the single-row fetch. Keep in mind, however, that multi-row fetch is a CPU saver, and not necessarily an elapsed time saver.

When using multi-row fetch, the GET DIAGNOSTICS statement is not necessary, and should be avoided due to high CPU overhead. Instead use the SQLCODE field of the SQLCA to determine whether your fetch was successful (SQLCODE 000), if the fetch failed (negative SQLCODE), or if you hit end of file (SQLCODE 100). If you received an SQLCODE 100 then you can check the SQLERRD3 field of the SQLCA to determine the number of rows to process.

MULTI-ROW INSERT As with multi-row fetch reading multiple rows per FETCH statement, a multi-row insert can insert multiple rows into a table in a single INSERT statement. The INSERT statement simply needs to contain the “FOR n ROWS” clause, and the host variables referenced in the VALUES clause need to be host variable arrays. IBM states that multi-row inserts can result in as much as a 25% CPU savings over single row inserts. In addition, multi-row inserts can have a dramatic impact on the performance of remote applications in that the number of statements issued across a network can be significantly reduced.

The multi-row insert can be coded as ATOMIC, meaning that if one insert fails then the entire statement fails, or it can be coded as NOT ATOMIC ON SQLERROR CONTINUE, which means that any one failure of any of the inserts will only impact that one insert of the set.

As with the multi-row fetch, the GET DIAGNOSTICS statement is not initially necessary, and should be avoided for performance reasons unless it needed. In the case of a failed non-atomic multi-row insert you'll get a SQLCODE of -253 if one or more of the inserts failed. Only then should you use GET DIAGNOSTICS to determine which one failed. Remember, if you get a SQLCODE of zero then all the inserts were a success, and there is no need for additional analysis.

MERGE STATEMENT Many times applications are interfacing with other applications. In these situations an application may receive a large quantity of data that applies to multiple rows of a table. Typically the application would in this case perform a blind update. That is, the application would simply attempt to update the rows of data in the table, and if any update failed because a row was not found, then the application would insert the data instead. In other situations, the application may read all of the existing data, compare that data to the new incoming data, and then programmatically insert or update the table with the new data.

DB2 9 supports this type of processing via the MERGE statement. The MERGE statement updates a target (table or view, or the underlying tables of a fullselect) using the specified input data. Rows in the target that match the input data are updated as specified, and rows that do not exist in the target are inserted. MERGE can utilize a table or an array of variables as input.

Since the MERGE operates against multiple rows, it can be coded as ATOMIC or NOT ATOMIC. The NOT ATOMIC option will allow rows that have been successfully updated or inserted to remain if others have failed. The GET DIAGNOSTICS statement should be used along with NOT ATOMIC to determine which updates or inserts have failed.

The following example shows a MERGE of rows on the employee sample table:

```
MERGE INTO EMP AS EXISING_TBL
USING (VALUES (:EMPNO, :SALARY, :COMM, :BONUS)
       FOR :ROW-CNT ROWS) AS INPUT_TBL(EMPNO, SALARY, COMM, BONUS)
ON INPUT_TBL.EMPNO = EXISTING_TBL.EMPNO
WHEN MATCHED THEN
    UPDATE SET SALARY = INPUT_TBL.SALARY
            ,COMM = INPUT_TBL.COMM
            ,BONUS = INPUT_TBL.BONUS
WHEN NOT MATCHED THEN
    INSERT (EMPNO, SALARY, COMM, BONUS)
VALUES (INPUT_TBL.EMPNO, INPUT_TBL.SALARY, INPUT_TBL.COMM,
        INPUT_TBL.BONUS)
```

As with the multi-row insert operation the use of GET DIAGNOSTICS should be limited.

Placing Data Intensive Business Logic in the Database

There are many advanced features of DB2 UDB for z/OS that let you take advantage of the power of the mainframe server.

- Advanced and Complex SQL Statements
- User-Defined Functions (UDFs)
- Stored Procedures
- Triggers and Constraints

These advanced features allow applications to be written quickly by pushing some of the logic of the application into the database server. Most of the time advanced functionality can be incorporated into the database using these features at a much lower development cost than coding the feature into the application itself. A feature such as database enforced referential integrity (RI) is a perfect example of something that is quite easy to implement in the database, but would take significantly longer time to code in a program.

These advanced database features also allow application logic to be placed as part of the database engine itself, making this logic more easily reusable enterprise wide. Reusing existing logic will mean faster time to market for new applications that need that logic, and having the logic centrally located makes it easier to manage than client code. Also, in many cases having data intensive logic located on the database server will result in improved performance as that logic can process the data at the server, and only return a result to the client.

Using advanced SQL for performance was addressed in Chapter 3 of this guide, and so let's address the other features here.

USER-DEFINED FUNCTIONS Functions are a useful way of extending the programming power of the database engine. Functions allow us to push additional logic into our SQL statements. User-Defined scalar functions work on individual values of a parameter list, and return a single value result. A table function can return an actual table to a SQL statement for further processing (just like any other table). User-defined functions (UDF) provide a major breakthrough in database programming technology. UDFs actually allow developers and DBAs to extend the capabilities of the database. This allows for more processing to be pushed into the database engine, which in turns allows these types of processes to become more centralized and controllable. Virtually any type of processing can be placed in a UDF, including legacy application programs. This can be used to create some absolutely amazing results, as well as push legacy processing into SQL statements. Once your processing is inside SQL statements you can put those SQL statements anywhere. So that anywhere you can run your SQL statements (say, from a web browser) you can run your programs! So, just like complex SQL statements, UDFs place more logic into the highly portable SQL statements.

Also just like complex SQL, UDFs can be a performance advantage or disadvantage. If the UDFs process large amounts of data, and return a result to the SQL statement, there may be a performance advantage over the equivalent client application code. However, if a UDF is used to process data only then it can be a performance disadvantage, especially if the UDF is invoked many times or embedded in a table expression, as data type casting (for SQL scalar UDFs compared to the equivalent expression coded directly in the SQL statement) and task switch overhead (external UDFs run in a stored procedure address space) are expensive (DB2 V8 relieves some of this overhead for table functions). Converting a legacy program into a UDF in about a day's time, invoking that program from a SQL statement, and then placing that SQL statement where it can be accessed via a client process may just be worth that expense!

Simply put, if the UDF results in the application program issuing fewer SQL statements, or getting access to a legacy process then chances are that the UDF is the right decision.

STORED PROCEDURES Stored procedures are becoming more prevalent on the mainframe, and can be part of a valuable implementation strategy. Stored procedures can be a performance benefit for distributed applications, or a performance problem. In every good implementation there are trade-offs. Most of the trade-offs involve sacrificing performance for things like flexibility, reusability, security, and time to delivery. It is possible to minimize the impact of distributed application performance with the proper use of stored procedures.

Since stored procedures can be used to encapsulate business logic in a central location on the mainframe, they offer a great advantage as a source of secured, reusable code. By using a stored procedure the client will only need to have authority to execute the stored procedures and will not need authority to the DB2 tables that are accessed from the stored procedures. A properly implemented stored procedure can help improve availability. Stored procedures can be stopped, queuing all requestors. A change can be implemented while access is prevented, and the procedures restarted once the change has been made. If business logic and SQL access is encapsulated within stored procedures there is less dependency on client or application server code for business processes. That is, the client takes care of things like display logic and edits, and the stored procedure contains the business logic. This simplifies the change process, and makes the code more reusable. In addition, like UDFs stored procedures can be used to access legacy data stores, and quickly web enable our legacy processes.

The major advantage to stored procedures is when they are implemented in a client/server application that must issue several remote SQL statements. The network overhead involved in sending multiple SQL commands and receiving result sets is quite significant, therefore proper use of stored procedures to accept a request, process that request with encapsulated SQL statements and business logic, and return a result will lessen the traffic across the network and reduce the application overhead. If a stored procedure is coded in this manner then it can be a significant performance improvement. Conversely, if the stored procedures contain only a few or one SQL statement the advantages of security, availability, and reusability can be realized, but performance will be worse than the equivalent single statement executions from the client due to task switch overhead.

DB2 9 offers a significant performance improvement for stored procedures with the introduction of native SQL procedures. These unfenced SQL procedures will execute as run time structures rather than be converted into external C program procedures (as with DB2 V7 and DB2 V8). Running these native SQL procedures will eliminate the task switch overhead of executing in the stored procedure address space. This represents a significant performance improvement for SQL procedures that contain little program logic, and few SQL statements.

TRIGGER AND CONSTRAINTS Triggers and constraints can be used to move application logic into the database. The greatest advantage to triggers and constraints is that they are generally data intensive operations, and these types of operations are better performers when placed close to the data. These features consist of:

- Triggers
- Database Enforced Referential Integrity (RI)
- Table Check Constraints

A trigger is a database object that contains some application logic in the form of SQL statements that are invoked when data in a DB2 table is changed. These triggers are installed into the database, and are then dependent upon the table on which they are defined. SQL DELETE, UPDATE, and INSERT statements can activate triggers. They can be used to replicate data, enforce certain business rules, and to fabricate data. Database enforced RI can be used to ensure that relationships from tables are maintained automatically. Child table data cannot be created unless a parent row exists, and rules can be implemented to tell DB2 to restrict or cascade deletes to a parent when child data exists. Table check constraints are used to ensure values of specific table columns, and are invoked during LOAD, insert, and update operations.

Triggers and constraints ease the programming burden because the logic, in the form of SQL is much easier to code than the equivalent application programming logic. This helps make the application programs smaller and easier to manage. In addition, since the triggers and constraints are connected to DB2 tables, then are centrally located rules and universally enforced. This helps to ensure data integrity across many application processes. Triggers can also be used to automatically invoke UDFs and stored procedures, which can introduce some automatic and centrally controlled intense application logic.

There are wonderful advantages to using triggers and constraints. They most certainly provide for better data integrity, faster application delivery time, and centrally located reusable code. Since the logic in triggers and constraints is usually data intensive their use typically outperforms the equivalent application logic simply due to the fact that no data has to be returned to the application when these automated processes fire. There is one trade-off for performance, however. When triggers, RI, or check constraints are used in place of application edits they can be a serious performance disadvantage. This is especially true if several edits on a data entry screen are verified at the server. It could be as bad as one trip to the server and back per edit. This would seriously increase message traffic between the client and the server. For this reason, data edits are best performed at the client when possible.

It is important to understand that when you are working with triggers you need to respect the triggers when performing schema migrations, or changes to the triggering tables. The triggers will, in some situations, have to be recreated in the same sequence they were originally created. In certain situations trigger execution sequence may be important, and if there are multiple triggers of the same type against a table then they will be executed in the order they were defined.

Organizing Your Input

One way to insure the fastest level of performance for large scale batch operations is to make sure that the input data to the process is organized in a meaningful manner. That is, the data is sorted according to the cluster of the primary table. Better yet, all of the tables accessed by the large batch process should have the same cluster as the input. This could mean pre-sorting data into the proper order prior to processing. If you code a generic transaction processor that handles both online and batch processing then you could be asking for trouble. If online is truly random then you can organize the tables and batch input files for the highest level of batch processing, and it should have little or no impact on the online transactions.

Remember, a locally executing batch process that is processing the input data in the same sequence as the cluster of your tables, and bound with `RELEASE(DEALLOCATE)` will utilize several performance enhancers, especially dynamic prefetch and index lookaside, to significantly improve the performance of these batch processes.

Searching

Search queries, as well as driving cursors (large queries that provide the input data to a batch process), can be expensive queries. Here there is, once again, a trade off between the amount of program code you are willing to write and the performance of your application.

If you code a generic search query, you will get generic performance. In the following example the `SELECT` statement basically supports a direct read, a range read, and a restart read in one statement. In order to enable this type of generic access a generic predicate has to be coded. In most cases this means that for every SQL statement issued more data will be read than is needed. This is due to the fact that DB2 has a limited ability to match on these types of predicates. In the following statement the predicate supports a direct read, sequential read, and restart read for at least one part of a three part compound key:

```

WHERE COL1 = WS-COL1-MIN
AND ( ( COL2 >= WS-COL2-MIN
      AND COL2 <= WS-COL2-MAX
      AND COL3 >= WS-COL3-MIN
      AND COL3 <= WS-COL3-MAX)
OR
      ( COL2 > WS-COL2-MIN
      AND COL2 <= WS-COL2-MAX ) )
OR ( COL1 > WS-COL1-MIN
    AND COL1 <= WS-COL1-MAX )

```

These predicates are very flexible, however they are not the best performing. The predicate in this example most likely results in a non-matching index scan even though an index on COL1, COL2, COL3 is available. This means that the entire index will have to be searched each time the query is executed. This is not a bad access path for a batch cursor that is reading an entire table in a particular order. For any other query, however, it is a detriment. This is especially true for online queries that are actually providing three columns of data (all min and max values are equal). For larger tables the CPU and elapsed time consumed can be significant.

The very best solution is to have separate predicates for various numbers of key columns provided. This will allow DB2 to have matching index access for each combination of key parts provided.

FIRST KEY ACCESS:

```

WHERE COL1 = WS-COL1

```

TWO KEYS PROVIDED:

```

WHERE COL1 = WS-COL1
AND COL2 = WS-COL2

```

THREE KEYS PROVIDED:

```

WHERE COL1 = WS-COL1
AND COL2 = WS-COL2
AND COL2 = WS-COL2

```

This will dramatically increase the number of SQL statements coded within the program, but will also dramatically increase the statement performance.

If the additional statements are not desired then there is another choice for the generic predicates. This would involve adding a redundant Boolean term predicate. These Boolean term predicates will enable DB2 to match on one column of the index. Therefore, for this WHERE clause:

```

WHERE COL1 = WS-COL1-MIN
AND ( ( COL2 >= WS-COL2-MIN
      AND COL2 <= WS-COL2-MAX
      AND COL3 >= WS-COL3-MIN
      AND COL3 <= WS-COL3-MAX)
OR
      ( COL2 > WS-COL2-MIN
      AND COL2 <= WS-COL2-MAX ) )
OR ( COL1 > WS-COL1-MIN
    AND COL1 <= WS-COL1-MAX )

```

An additional redundant predicate can be added:

```
WHERE (COL1 = WS-COL1-MIN
AND (( COL2 >= WS-COL2-MIN
AND COL2 <= WS-COL2-MAX
AND COL3 >= WS-COL3-MIN
AND COL3 <= WS-COL3-MAX)
OR
( COL2 > WS-COL2-MIN
AND COL2 <= WS-COL2-MAX ))
OR ( COL1 > WS-COL1-MIN
AND COL1 <= WS-COL1-MAX ))
AND ( COL1 => WS-COL1-MIN
AND COL1 <= WS-COL1-MAX )
```

The addition of this redundant predicate does not affect the result of the query, but allows DB2 to match on the COL1 column.

Name searching can be a challenge. Once again we are faced with multiple queries to solve multiple conditions, or one large generic query to solve any request. In many cases it pays to study the common input fields for a search, and then code specific queries that match those columns only, and are supported by an index. Then, the generic query can support the less frequently searched on fields.

We have choices for coding our search queries. Let's say that we need to search for two variations of a name to try and find someone in our database. The following query can be coded to achieve that (in this case a name reversal):

```
SELECT PERSON_ID
FROM   PERSON_TBL
WHERE  (LASTNAME = 'RADY' AND
        FIRST_NAME = 'BOB') OR
        (LASTNAME = 'BOB' AND
        FIRST_NAME = 'RADY');
```

This query gets the job done, but uses multi-index access at best. Another way you could code the query is as follows:

```
SELECT PERSON_ID
FROM   PERSON_TBL
WHERE  LASTNAME = 'RADY'
AND    FIRST_NAME = 'BOB'
UNION ALL
SELECT PERSON_ID
FROM   PERSON_TBL
WHERE  LASTNAME = 'BOB'
AND    FIRST_NAME = 'RADY'
```

This query gets better index access, but will probe the table twice. The next query uses a common table expression (DB2 V8, DB2 9) to build a search list, and then divides that table into the person table:

```
WITH PRSN_SEARCH(LASTNAME, FIRST_NAME) AS
  (SELECT 'RADY', 'BOB' FROM SYSIBM.SYSDUMMY1
   UNION ALL
   SELECT 'BOB', 'RADY' FROM SYSIBM.SYSDUMMY1)
SELECT PERSON_ID
FROM   PERSON_TBL A, PRSN_SEARCH B
WHERE  A.LASTNAME = B.LASTNAME
AND    A.FIRST_NAME = B.FIRST_NAME
```

This query gets good index matching and perhaps reduced probes. Finally, the next query utilizes a during join predicate to probe on the first condition and only apply the second condition if the first finds nothing. That is, it will only execute the second search if the first finds nothing and completely avoid the second probe into the table. Keep in mind that this query may not produce the same results as the previous queries due to the optionality of the search:

```
SELECT COALESCE(A.PERSON_ID, B.PERSON_ID)
FROM   SYSIBM.SYSDUMMY1
LEFT OUTER JOIN
      PERSON_TBL A
ON IBMREQD = 'Y'
AND (A.LASTNAME = 'RADY' OR A.LASTNAME IS NULL)
AND (A.FIRST_NAME = 'BOB' OR A.FIRST_NAME IS NULL)
LEFT OUTER JOIN
  (SELECT PERSON_ID
   FROM   PERSON_TBL
   WHERE  LASTNAME = 'BOB' AND FIRSTNAME = 'RADY') AS B
ON A.EMPNO IS NULL
```

Which search query is the best for your situation? Test and find out! Just keep in mind that when performance is a concern there are many choices.

Existence Checking

What is the best for existence checking within a query? Is it a join, a correlated subquery, or a non-correlated subquery? Of course it depends on your situation, but these types of existence checks are always better resolved in a SQL statement than with separate queries in your program. Here are the general guidelines:

Non-correlated subqueries, such as this:

```
SELECT SNAME
FROM S
WHERE S# IN
  (SELECT S# FROM SP
   WHERE P# = 'P2')
```

Are generally good when there is no available index for inner select but there is on outer table column (indexable). Or when there is no index on either inner or outer columns. We also like non-correlated subqueries when there is relatively a small amount of data provided by the subquery. Keep in mind that DB2 can transform a non-correlated subquery to a join.

Correlated subqueries, such as this:

```
SELECT SNAME
FROM S
WHERE EXISTS
  (SELECT * FROM SP
   WHERE SP.P# = 'P2'
   AND SP.S# = S.S#)
```

Are generally good when there is a supporting index available on inner select and there is a cost benefit in reducing repeated executions to inner table and distinct sort for join. They may be a benefit as well if the inner query could return a large amount of data if coded as a non-correlated subquery as long as there is a supporting index for the inner query. Also, the correlated subquery can outperform the equivalent non-correlated subquery if an index on the outer table is not used (DB2 chose a different index based upon other predicates) and one exists in support of the inner table.

Joins, such as this:

```
SELECT DISTINCT SNAME
FROM   S, SP
WHERE  S.S# = SP.S#
AND    SP.P# = 'P2'
```

May be best if supporting indexes are available and most rows hook up in the join. Also, if the join results in no extra rows returned then the DISTINCT can also be avoided. Joins can provide DB2 the opportunity to pick the best table access sequence, as well as apply predicate transitive closure.

Which existence check method is best for your situation? We don't know, but you have choices and should try them out! It should also be noted that as of DB2 9 it is possible to code and ORDER BY and FETCH first in a subquery, which can provide even more options for existence checking in subqueries!

For singleton existence checks you can code FETCH FIRST and ORDER BY clauses in a singleton select. This could provide the best existence checking performance in a stand alone query:

```
SELECT 1 INTO :hv-check
FROM   TABLE
WHERE  COL1 = :hv1
FETCH  FIRST 1 ROW ONLY
```

Avoiding Read-Only Cursor Ambiguity and Locking

Lock avoidance was introduced in DB2 to reduce the overhead of always locking everything. DB2 will check to be sure that a lock is probably necessary for data integrity before acquiring a lock. Lock avoidance is critical for performance. Its effectiveness is controlled by application commits. Lock avoidance is also generally used with both isolation levels of cursor stability (CS) and repeatable read (RR) for referential integrity constraint checks. For a plan or package bound with RR, a page lock is required for the dependent page if a dependent row is found. This will be held in order to guarantee repeatability of the error on checks for updating primary keys for when deleting is restricted.

We need to bind our programs with CURRENTDATA(NO) and ISOLATION(CS) in DB2 V7 in order to allow for lock avoidance. In DB2 V8 and DB2 9 CURRENTDATA(YES) can also avoid locks. Although DB2 will consider ambiguous cursors as read-only, it is always best to code your read-only cursors with the clause FOR FETCH ONLY or FOR READ ONLY.

Lock avoidance also needs frequent commits so that other processes do not have to acquire locks on updated pages, and this also allows for page reorganization to occur to clear the “possibly uncommitted” (PUNC) bit flags in a page. Frequent commits allow the commit log sequence number (CLSN) on a page to be updated more often since it is dependent on the begin unit of recovery in the log, the oldest begin unit of recovery being required.

The best way to avoid taking locks in your read-only cursors is to read uncommitted. Use the WITH UR clause in your statements to avoid taking or waiting on locks. Keep in mind however that using WITH UR can result in the reading of uncommitted, or dirty, data that may eventually be rolled back. If you are using WITH UR in an application that will update the data, then an optimistic locking strategy is your best performing option.

Optimistic Locking

With high demands for full database availability, as well as high transaction rates and levels of concurrency, reducing database locks is always desired. With this in mind, many applications are employing a technique called “optimistic locking” to achieve these higher levels of availability and concurrency. This technique traditionally involves reading data with an uncommitted read or with cursor stability. Update timestamps are maintained in all of the data tables. This update timestamp is read along with all the other data in a row. When a direct update is subsequently performed on the row that was selected, the timestamp is used to verify that no other application or user has changed the data between the point of the read and the update. This places additional responsibility on the application to use the timestamp on all updates, but the result is a higher level of DB2 performance and concurrency.

Here is a hypothetical example of optimistic locking. First the application reads the data from a table with the intention of subsequently updating:

```
SELECT UPDATE_TS, DATA1
FROM TABLE1
WHERE KEY1 = :WS-KEY1
WITH UR
```

Here the data is has been changed and the update takes place.

```
UPDATE TABLE1
SET DATA1 = :WS-DATA1, UPDATE_TS = :WS-NEW-UPDATE-TS
WHERE KEY1 = :WS-KEY1
AND UPDATE_TS = :WS-UPDATE-TS
```

If the data has changed then the update will get a SQLCODE of 100, and restart logic will have to be employed for the update. This requires that all applications respect the optimistic locking strategy and update timestamp when updating the same table.

As of DB2 9, IBM has introduced built-in support for optimistic locking via the ROW CHANGE TIMESTAMP. When a table is created or altered, a special column can be created as a row change timestamp. These timestamp columns will be automatically updated by DB2 whenever a row of a table is updated. This built-in support for optimistic locking takes some of the responsibility (that of updating the timestamp) out of the hands of the various applications that might be updating the data.

Here is how the previous example would look when using the ROW CHANGE TIMESTAMP for optimistic locking:

```
SELECT ROW CHANGE TIMESTAMP FOR TABLE1, DATA1
FROM TABLE1
WHERE KEY1 = :WS-KEY1
WITH UR
```

Here the data has been changed and update takes place.

```
UPDATE TABLE1
SET DATA1 = :WS-DATA1
WHERE KEY1 = :WS-KEY1
AND ROW CHANGE TIMSTAMP FOR TABLE1 = :WS-UPDATE-TS
```

Commit Strategies and Heuristic Control Tables

Heuristic control tables are implemented to allow great flexibility in controlling concurrency and restartability. Our processing has become more complex, our tables have become larger, and our requirements are now 5 9's availability (99.999%). In order to manage our environments and its objects, we need to have dynamic control of everything. How many different control tables, and what indicator columns we put in them will vary depending on the objects and the processing requirements.

Heuristic control/restart tables have rows unique to each application process to assist in controlling the commit scope using the number of database updates or time between commits as their primary focus. There can also be an indicator in the table to tell an application that it is time to stop at the next commit point. These tables are accessed every time an application starts a unit-of-recovery (unit-of-work), which would be at process initiation or at a commit point. The normal process is for an application to read the control table at the very beginning of the process to get the dynamic parameters and commit time to be used. The table is then used to store information about the frequency of commits as well as any other dynamic information that is pertinent to the application process, such as the unavailability of some particular resource for a period of time. Once the program is running, it both updates and reads from the control table at commit time. Information about the status of all processing at the time of the commit is generally stored in the table so that a restart can occur at that point if required.

Values in these tables can be changed either through SQL in a program or by a production control specialist to be able to dynamically account for the differences in processes through time. For example, you would probably want to change the commit scope of a job that is running during the on-line day vs. when it is running during the evening. You can also set indicators to tell an application to gracefully shut down, run different queries with different access paths due to a resource being taken down, or go to sleep for a period of time.

Tuning Subsystems

There are several areas in the DB2 subsystem that you can examine for performance improvements. These areas are components of DB2 that aid in application processing. This chapter describes those DB2 components and presents some tuning tips for them.

Buffer Pools

Buffer pools are areas of virtual storage that temporarily store pages of table spaces or indexes. When a program accesses a row of a table DB2 places the page containing that row in a buffer. When a program changes a row of a table DB2 must write the data in the buffer back to disk (eventually) normally either at a DB2 system checkpoint or a write threshold. The write thresholds are either a vertical threshold at the page set level or a horizontal threshold at the buffer pool level.

The way buffer pools work is fairly simple by design, but it is tuning these simple operations that can make all the difference in the world to the performance of our applications. The data manager issues GETPAGE requests to the buffer manager who hopefully can satisfy the request from the buffer pool instead of having to retrieve the page from disk. We often trade CPU for I/O in order to manage our buffer pools efficiently. Buffer pools are maintained by subsystem, but individual buffer pool design and use should be by object granularity and in some cases also by application.

DB2 buffer pool management by design allows the following: ability to ALTER and DISPLAY buffer pool information dynamically without requiring a bounce of the DB2 subsystem. This improves availability by allowing us to dynamically create new buffer pools when necessary and to also dynamically modify or delete buffer pools. We may find we need to do ALTERs of buffer pools a couple times during the day because of varying workload characteristics. We will discuss this when we look at tuning the buffer pool thresholds. Initial buffer pool definitions set at installation/migration but are often hard to configure at this time because the application process against the objects is usually not detailed at installation. But regardless of what is set at installation we can use ALTER any time after the install to add/delete new buffer pools, resize the buffer pools or change any of the thresholds. The buffer pool definitions are stored in BSDS (Boot Strap Dataset) and we can move objects between buffer pools via an ALTER INDEX/TABLESPACE and a subsequent START/STOP command of the object.

Pages

There are three types of pages in virtual pools:

- Available pages: pages on an available queue (LRU, FIFO, MRU) for stealing
- In-Use pages: pages currently in use by a process that are not available for stealing. In Use counts do not indicate the size of the buffer pool, but this count can help determine residency for initial sizing
- Updated pages: these pages are not 'in-use', not available for stealing, and are considered 'dirty pages' in buffer pool waiting to be externalized

There are four page sizes and several bufferpools to support each size:

BP0 - BP49	4K pages
BP8K0 - BP8K9	8K pages
BP16K0 - BP16K9	16K pages
BP32K0 - BP32K9	32K pages

Work file table space pages are only 4K or 32K. There is a DSNZPARM called DSVCI that allows the control interval to match to the actual page size.

Our asynchronous page writes per I/O will change with each page size accordingly.

4K Pages	32 Writes per I/O
8K Pages	16 Writes per I/O
16K Pages	8 Writes per I/O
32K Pages	4 Writes per I/O

With these new page sizes we can achieve better hit ratios and have less I/O because we can fit more rows on a page. For instance if we have a 2200 byte row (maybe for a data warehouse), a 4K page would only be able to hold 1 row, but if an 8K page was used 3 rows could fit on a page, 1 more than if 4K pages were used and one less lock also if required. However, we do not want to use these new page sizes as a band-aid for what may be a poor design. You may want to consider decreasing the row size based upon usage to get more rows per page.

Virtual Buffer Pools

We can have buffer pools up to 80 virtual buffer pools. This allows for up to 50 4K page buffer pools (BP0 - BP49), up to 10 32K page buffer pools (BP32K - BP32K9), up to 10 8K page buffer pools and up to 10 16K page buffer pools. The size of the buffer pools is limited by the physical memory available on your system, with a maximum size for all buffer pools of 1TB. It does not take any additional resources to search a large pool versus a small pool. If you exceed the available memory in the system, then the system will begin swapping pages from physical memory to disk, which can have severe performance impacts.

Buffer Pool Queue Management

Pages used in the buffer pools are processed in two categories: Random (pages read one at a time) or Sequential (pages read via prefetch). These pages are queued separately: LRU — Random Least Recently Used queue or SLRU — Sequential Least Recently Used Queue (Prefetch will only steal from this queue). The percentage of each queue in a buffer pool is controlled via the VPSEQT parameter (Sequential Steal Threshold). This becomes a hard threshold to adjust, and often requires two settings — for example, one setting for batch processing and a different setting for on-line processing. The way we process our data between our batch and on-line processes often differs. Batch is usually more sequentially processed, whereas on-line is processed more randomly.

DB2 breaks up these queues into multiple LRU chains. This way there is less overhead for queue management because the latch that is taken at the head of the queue (actually on the hash control block which keeps the order of the pages on the queue) will be latched less because the queues are smaller. Multiple subpools are created for a large virtual buffer pool and the threshold is controlled by DB2, not to exceed 4000 VBP buffers in each subpool. The LRU queue is managed within each of the subpools in order to reduce the buffer pool latch contention when the degree of concurrency is high. Stealing of these buffers occurs in a round-robin fashion through the subpools.

FIFO — First-in, first-out can also be used instead of the default of LRU. With this method the oldest pages are moved out regardless. This decreases the cost of doing a GETPAGE operation and reduces internal latch contention for high concurrency. This would only be used where there is little or no I/O and where table space or index is resident in the buffer pool. We will have separate buffer pools with LRU and FIFO objects and this can be set via the ALTER BUFFERPOOL command with a new PGSTEAL option of FIFO. LRU is the PGSTEAL option default.

I/O Requests and Externalization

Synchronous reads are physical pages that are read in one page per I/O. Synchronous writes are pages written one page per I/O. We want to keep synchronous read and writes to only what is truly necessary, meaning small in occurrence and number. If not, we may begin to see buffer pool stress (maybe too many checkpoints). DB2 will begin to use synchronous writes if the IWTH threshold (Immediate Write) is reached (more on this threshold later in this chapter) or if 2 system checkpoints pass without a page being written that has been updated and not yet committed.

Asynchronous reads are several pages read per I/O for such prefetch operations such as sequential prefetch, dynamic prefetch or list prefetch. Asynchronous writes are several pages per I/O for such operations as deferred writes.

Pages are externalized to disk when the following occurs:

- DWQT threshold reached
- VDWQT threshold reached
- Dataset is physically closed or switched from R/W to R/O
- DB2 takes a checkpoint (LOGLOAD or CHKFREQ is reached)
- QUIESCE (WRITE YES) utility is executed
- If page is at the top of LRU chain and another update is required of the same page by another process

We want to control page externalization via our DWQT and VDWQT thresholds for best performance and avoid surges in I/O. We do not want page externalization to be controlled by DB2 system checkpoints because too many pages would be written to disk at one time causing I/O queuing delays, increased response time and I/O spikes. During a checkpoint all updated pages in the buffer pools are externalized to disk and the checkpoint recorded in the log (except for the work files).

Checkpoints and Page Externalization

DB2 checkpoints are controlled through the DSNZPARM — CHKFREQ. The CHKFREQ parameter is the number of minutes between checkpoints for a value of 1 to 60, or the number of log records written between DB2 checkpoints for a value of 200 to 16,000,000. The default value is 500,000. Often we may need different settings for this parameter depending on our workload. For example we may want it higher during our batch processing. However, this is a hard parameter to set often because it requires a bounce of the DB2 subsystem in order to take effect. Recognizing the importance of the ability to change this parameter based on workloads. The SET LOG CHKTIME command allows us to dynamically set the CHKFREQ parameter. There have been other options added to the -SET LOG command to be able to SUSPEND and RESUME logging for a DB2 subsystem. SUSPEND causes a system checkpoint to be taken in a non-data sharing environment. By obtaining the log-write latch any further log records are prevented from being created and any unwritten log buffers will be written to disk. Also, the BSDS will be updated with the high-written RBA. All further database updates are prevented until update activity is resumed by issuing a -SET LOG command to RESUME logging, or until a -STOP DB2 command is issued. These are single-subsystem only commands so they will have to be entered for each member when running in a data sharing environment.

In very general terms during an on-line processing, DB2 should checkpoint about every 5 to 10 minutes, or some other value based on investigative analysis of the impact on restart time after a failure. There are two real concerns for how often we take checkpoints:

- The cost and disruption of the checkpoints
- The restart time for the subsystem after a crash

Many times the costs and disruption of DB2 checkpoints are overstated. While a DB2 checkpoint is a tiny hiccup, it does not prevent processing from proceeding. Having a CHKFREQ setting that is too high along with large buffer pools and high thresholds, such as the defaults, can cause enough I/O to make the checkpoint disruptive. In trying to control checkpoints, some users increased the CHKFREQ value and made the checkpoints less frequent, but in effect made them much more disruptive. The situation is corrected by reducing the amount written and increasing the checkpoint frequency which yields much better performance and availability. It is not only possible, but does occur at some installations, that a checkpoint every minute did not impact performance or availability. The write efficiency at DB2 checkpoints is the key factor needed to be observed to see if CHKFREQ can be reduced. If the write thresholds (DWQT/VDQWT) are doing their job, then there is less work to perform at each checkpoint. Also using the write thresholds to cause I/O to be performed in a level, non-disruptive fashion is also helpful for the non-volatile storage in storage controllers.

However, even if we have our write thresholds (DWQT/VDQWT) set properly, as well as our checkpoints, we could still see an unwanted write problem. This could occur if we do not have our log datasets properly sized. If the active log data sets are too small then active log switches will occur often. When an active log switch takes place a checkpoint is taken automatically. Therefore, our logs could be driving excessive check point processing resulting in constant writes. This would prevent us from achieving a high ratio of pages written per I/O because the deferred write queue would not be allowed to fill as it should.

Sizing

Buffer pool sizes are determined by the VPSIZE parameter. This parameter determines the number of pages to be used for the virtual pool. DB2 can handle large bufferpools efficiently, as long as enough real memory is available. If insufficient real storage exists to back the bufferpool storage requested, then paging can occur. Paging can occur when the bufferpool size exceeds the available real memory on the z/OS image. DB2 limits the total amount of storage allocated for bufferpools to approximately twice the amount of real storage (but less is recommended). There is a maximum of 1TB total for all bufferpools (provided the real storage is available).

In order to size bufferpools it is helpful to know the residency rate of the pages for the object(s) in the bufferpool.

Sequential vs. Random Processing

The VPSEQT (Virtual Pool Sequential Steal Threshold) is the percentage of the virtual buffer pool that can be used for sequentially accessed pages. This is to prevent sequential data from using all the buffer pool and keep some space available for random processing. The value is 0 to 100% with a default of 80%. That would indicate that 80% of the buffer pool is to be set aside for sequential processing, and 20% for random processing. This parameter needs to be set according to how your objects in that buffer pool are processed.

One tuning option often used is altering the VPSEQT to 0 to set the pool up for just random use. When the VPSEQT is altered to 0, the SLRU will no longer be valid and the buffer pool is now totally random. Since only the LRU will be used, all pages on the SLRU have to be freed. This will also disable prefetch operations in this buffer pool and this is beneficial for certain strategies. However, there are problems with this strategy for certain buffer pools and this will be addressed later.

Writes

The DWQT (Deferred Write Threshold), also known as the Horizontal Deferred Write Threshold, is the percentage threshold that determines when DB2 starts turning on write engines to begin deferred writes (32 pages/Async I/O). The value can be from 0 to 90%. When the threshold is reached, write engines (up to 600 write engines as of this publication) begin writing pages out to disk. Running out of write engines can occur if the write thresholds are not set to keep a constant flow of updated pages being written to disk. This can occur and if it is uncommon then it is okay, but if this occurs daily then there is a tuning opportunity. DB2 turns on these write engines, basically one vertical pageset, queue at a time, until a 10% reverse threshold is met. When DB2 runs out of write engines it can be detected in the statistics reports in the WRITE ENGINES NOT AVAILABLE indicator on Statistics report.

When setting the DWQT threshold a high value is useful to help improve hit ratio for updated pages, but will increase I/O time when deferred write engines begin. We would use a low value to reduce I/O length for deferred write engines, but this will increase the number of deferred writes. This threshold should be set based on the referencing of the data by the applications.

If we choose to set the DWQT to zero so that all objects defined to the buffer pool are scheduled to be written immediately to disk then DB2 actually uses its own internal calculations for exactly how many changed pages can exist in the buffer pool before it is written to disk.

32 pages are still written per I/O, but it will take 40 dirty pages (updated pages) to trigger the threshold so that the highly re-referenced updated pages, such as space map pages, remain in the buffer pool.

When implementing LOBs (Large Objects), a separate buffer pool should be used and this buffer pool should not be shared (backed by a group buffer pool in a data sharing environment). The DWQT should be set to 0 so that for LOBS with LOG NO, force-at-commit processing occurs and the updates continually flow to disk instead of surges of writes. For LOBs defined with LOG YES, DB2 could use deferred writes and avoid massive surges at checkpoint.

The DWQT threshold works at a buffer pool level for controlling writes of pages to the buffer pools, but for a more efficient write process you will want to control writes at the pageset/partition level. This can be controlled via the VDWQT (Vertical Deferred Write Threshold). The percentage threshold that determines when DB2 starts turning on write engines and begins the deferred writes for a given data set. This helps to keep a particular pageset/partition from monopolizing the entire buffer pool with its updated pages. The value is 0 to 90% with a default of 10%. The VDWQT should always be less than the DWQT.

A good rule of thumb for setting the VDWQT is that if less than 10 pages are written per I/O, set it to 0. You may also want to set it to 0 to trickle write the data out to disk. It is normally best to keep this value low in order to prevent heavily updated pagesets from dominating the section of the deferred write area. Either a percentage of pages or actual number of pages, from 0 to 9999, can be specified for the VDWQT. You must set the percentage to 0 to use the number specified. Set to 0,0 and system uses $\text{MIN}(32,1\%)$ for good for trickle I/O.

If we choose to set the VDWQT to zero, 32 pages are still written per I/O, but it will take 40 dirty pages (updated pages) to trigger the threshold so that the highly re-referenced updated pages, such as space map pages, remain in the buffer pool.

It is a good idea to set the VDWQT using a number rather than a percentage because if someone increases the buffer pool that means that now more pages for a particular pageset can occupy the buffer pool and this may not always be optimal or what you want.

When looking at any performance report, showing the amount of activity for the VDWQT and the DWQT, you would want to see the VDWQT being triggered most of the time (VERTIC.DEFER.WRITE THRESHOLD), and the DWQT extremely less (HORIZ.DEFER.WRITE THRESHOLD). There can be no general ratios since that would depend on the both the activity and the number of objects in the buffer pools. The bottom line is that we would want to be controlling I/O by the VDWQT, with the DWQT watching for and controlling activity across the entire pool and in general writing out rapidly queuing up pages. This will also assist in limiting the amount of I/O that checkpoint would have to perform.

Parallelism

THE VPPSEQT Virtual Pool Parallel Sequential Threshold is the percentage of VPSEQT setting that can be used for parallel operations. The value is 0 to 100% with a default of 50%. If this is set to 0 then parallelism is disabled for objects in that particular buffer pool. This can be useful in buffer pools that cannot support parallel operations. The VPXPSEQT — Virtual Pool Sysplex Parallel Sequential Threshold is a percentage of the VPPSEQT to use for inbound queries. It also defaults to 50% and if it is set to 0, Sysplex Query Parallelism is disabled when originating from the member the pool is allocated to. In affinity data sharing environments this is normally set to 0 to prevent inbound resource consumption of work files and bufferpools.

Stealing Method

The VPSTEAL threshold allows us to choose a queuing method for the buffer pools. The default is LRU (Least Recently Used), but FIFO (First In- First Out) is also an option. This option turns off the overhead for maintaining the queue and may be useful for objects that can completely fit in the bufferpool or if the hit ratio is less than 1%.

Page Fixing

You can use the PGFIX keyword with the ALTER BUFFERPOOL command to fix a buffer pool in real storage for an extended period of time. The PGFIX keyword has the following options:

- **PGFIX(YES)** The buffer pool is fixed in real storage for the long term. Page buffers are fixed when they are first used and remain fixed.
- **PGFIX(NO)** The buffer pool is not fixed in real storage for the long term. Page buffers are fixed and unfixed in real storage, allowing for paging to disk. PGFIX(NO) is the default option.

The recommendation is to use PGFIX(YES) for buffer pools with a high I/O rate, that is, a high number of pages read or written. For buffer pools with zero I/O, such as some read-only data or some indexes with a nearly 100% hit ratio, PGFIX(YES) is not recommended. In these cases, PGFIX(YES) does not provide a performance advantage.

Internal Thresholds

The following thresholds are a percent of unavailable pages to total pages, where unavailable means either updated or in use by a process.

SPTH

THE SPTH Sequential Prefetch Threshold is checked before a prefetch operation is scheduled and during buffer allocation for a previously scheduled prefetch. If the SPTH threshold is exceeded prefetch will either not be scheduled or will be canceled. PREFETCH DISABLED — NO BUFFER (Indicator on Statistics Report) will be incremented every time a virtual buffer pool reaches 90% of active unavailable pages, disabling sequential prefetch. This value should always be zero. If this value is not 0, then it is a clear indication that you are probably experiencing degradation in performance due to all prefetch being disabled. To eliminate this you may want to increase the size of the buffer pool (VPSIZE). Another option may be to have more frequent commits in the application programs to free pages in the buffer pool, as this will put the pages on the write queues.

DMTH

THE DMTH Data Manager Threshold (also referred to as Buffer Critical Threshold) occurs when 95% of all buffer pages are unavailable (in use). The Buffer Manager will request all threads to release any possible pages immediately. This occurs by a setting GETPAGE/ RELPAGE processing by row instead of page. After a GETPAGE and single row is processed then a RELPAGE is issued. This will cause CPU to become high for objects in that buffer pool and I/O sensitive transaction can suffer. This can occur if the buffer pool is too small. You can observe when this occurs by seeing a non-zero value in the DM THRESHOLD REACHED indicator on a statistics reports. This is checked every time a page is read or updated. If this threshold is not reached then DB2 will access the page in the virtual pool once for each page (no matter how many rows used). If this threshold has been reached then DB2 will access the page in the virtual pool once for every ROW on the page that is retrieved or updated. This can lead to serious performance degradation.

IWTH

The Immediate Write Threshold (IWTH) is reached when 97.5% of buffers are unavailable (in use). If this threshold is reached then synchronous writes begin and this presents a performance problem. For example if there are 100 rows in page and if there are 100 updates then 100 synchronous writes will occur, one by one for each row. Synchronous writes are not concurrent with SQL, but serial, so the application will be waiting while the write occurs (including 100 log writes which must occur first). This causes large increases in I/O time. It is not recorded explicitly in a statistic reports, but DB2 will appear to be hung and you will see synchronous writes begin to occur when this threshold is reached. Be careful with some various monitors that send exception messages to the console when synchronous writes occur and refers to it as IWTH reached — not all synchronous writes are caused by this threshold being reached. This is simply being reported incorrectly. See the following note.

Note: Be aware that if looking at some performance reports, the IWTH counter can also be incremented when dirty pages are on the write queue have been re-referenced, which has caused a synchronous I/O before the page could be used by the new process. This threshold counter can also be incremented if more than two checkpoints occur before an updated page is written since this will cause a synchronous I/O to write out the page.

Virtual Pool Design Strategies

Separate buffer pools should be used based upon their type of usage by the applications (such as buffer pools for objects that are randomly accessed vs. those that are sequentially accessed). Each one of these buffer pools will have its own unique settings and the type of processing may even differ between the batch cycle and the on-line day. These are very generic breakouts just for this example. Actual definitions would be much finer tuned, less generic.

MORE DETAILED EXAMPLE OF BUFFER POOL OBJECT BREAKOUTS

BP0	Catalog and directory — DB2 only use
BP1	Work files (Sort)
BP2	Code and reference tables — heavily accessed
BP3	Small tables, heavily updated — trans tables, work tables
BP4	Basic tables
BP5	Basic indexes
BP6	Special for large clustered, range scanned table
BP7	Special for Master Table full index (Random searched table)
BP8	Special for an entire database for a special application
BP9	Derived tables and “saved” tables for ad-hoc
BP10	Staging tables (edit tables for short lived data)
BP11	Staging indexes (edit tables for short lived data)
BP12	Vendor tool/utility object

Tuning with the -DISPLAY Buffer Pool Command

In several cases the buffer pools can be tuned effectively using the DISPLAY BUFFERPOOL command. When a tool is not available for tuning, the following steps can be used to help tune buffer pools.

1. Use command and view statistics
2. Make changes (i.e. thresholds, size, object placement)
3. Use command again during processing and view statistics
4. Measure statistics

The output contains valuable information such as prefetch information (Sequential, List, Dynamic Requests) Pages Read, Prefetch I/O and Disablement (No buffer, No engine). The incremental detail display shifts the time frame every time a new display is performed.

RID Pool

The RID (Row Identifier) pool is used for storing and sorting RIDs for operations such as:

- List Prefetch
- Multiple Index Access
- Hybrid Joins
- Enforcing unique keys while updating multiple rows

The RID pool is looked at by the optimizer for prefetch and RID use. The full use of RID POOL is possible for any single user at run time. Run time can result in a table space scan being performed if not enough space is available in the RID. For example, if you want to retrieve 10,000 rows from 100,000,000 row table and there is no RID pool available, then a scan of 100,000,000 rows would occur, at any time and without external notification. The optimizer assumes physical I/O will be less with a large pool.

Sizing

The default size of the RID pool is currently 8 MB with a maximum size of 10000 MB, and is controlled by the MAXRBLK installation parameter. The RID pool could be set to 0, and this would disable the types of operations that use the RID pool, and DB2 would not choose access paths that the RID pool supports. The RID pool is created at start up time, but no space is allocated until RID storage is actually needed. It is then allocated in 32KB blocks as needed, until the maximum size you specified on installation panel DSNTIPC is reached. There are a few guidelines for setting the RID pool size. You should have as large a RID pool as required as it is a benefit for processing and can lead to performance degradation if it is too small. A good guideline for sizing the RID pool is as follows:

```
Number of concurrent RID processing activities
X average number of RIDs x 2 x 5 bytes per RID
```

Statistics to Monitor

There are three statistics to monitor for RID pool problems:

RIDS OVER THE RDS LIMIT This is the number of times list prefetch is turned off because the RID list built for a single set of index entries is greater than 25% of number of rows in the table. If this is the case, DB2 determines that instead of using list prefetch to satisfy a query it would be more efficient to perform a table space scan, which may or may not be good depending on the size of the table accessed. Increasing the size of the RID pool will not help in this case. This is an application issue for access paths and needs to be evaluated for queries using list prefetch.

There is one very critical issue regarding this type of failure. The 25% threshold is actually stored in the package/plan at bind time, therefore it may no longer match the real 25% value, and in fact could be far less. It is important to know what packages/plans are using list prefetch, and on what tables. If the underlying tables are growing, then rebinding the packages/plans that are dependent on it should be rebound after a RUNSTATS utility has updated the statistics. Key correlation statistics and better information about skewed distribution of data can also help to gather better statistics for access path selection and may help avoid this problem.

RIDS OVER THE DM LIMIT This occurs when over 28 million RIDS were required to satisfy a query. Currently there is a 28 million RID limit in DB2. The consequences of hitting this limit can be fallback to a table space scan. In order to control this, you have a couple of options:

- Fix the index by doing something creative
- Add an additional index better suited for filtering
- Force list prefetch off and use another index
- Rewrite the query
- Maybe it just requires a table space scan

INSUFFICIENT POOL SIZE This indicates that the RID pool is too small.

The SORT Pool

Sorts are performed in 2 phases:

- Initialization
 - DB2 builds ordered sets of 'runs' from the given input
- Merge
 - DB2 will merge the 'runs' together

DB2 allocates at startup a sort pool in the private area of the DBM1 address space. DB2 uses a special sorting technique called a tournament sort. During the sorting processes it is not uncommon for this algorithm to produce logical work files called runs, which are intermediate sets of ordered data. If the sort pool is large enough then the sort completes in that area. More often than not the sort cannot complete in the sort pool and the runs are moved into the work file database, especially if there are many rows to sort. These runs are later merged to complete the sort. When work file database is used for holding the pages that make up the sort runs, you could experience performance degradation if the pages get externalized to the physical work files since they will have to be read back in later in order to complete the sort.

Size

The sort pool size defaults to 2MB unless specified. It can range in size from 240KB to 128MB and is set with an installation DSNZPARM. The larger the Sort Pool (Sort Work Area) is, the fewer sort runs are produced. If the sort pool is large enough then the buffer pools and sort work files may not be used. If buffer pools and work file database are not used then the better performance will be due to less I/O. We want to size sort pool and work file database large because we do not want sorts to have pages being written to disk.

The EDM Pool

The EDM pool (Environmental Descriptor Manager) is made up of three components, each of which is in its own separate storage, and each contains many items including the following:

- EDM Pool
 - CTs - cursor tables (copies of the SKCTs)
 - PTs - package tables (copies of the SKPTs)
 - Authorization cache block for each plan
 - Except those with CACHESIZE set to 0
- EDM Skeleton Pool
 - SKCTs - skeleton cursor tables
 - SKPTs - skeleton package tables
- EDM DBD cache
 - DBDs - database descriptors
- EDM Statement Cache
 - Skeletons of dynamic SQL for CACHE DYNAMIC SQL

Sizing

If the pool is too small, then you will see increased I/O activity in the following DB2 table spaces, which support the DB2 directory:

```
DSNDB01.DBD01
DSNDB01.SPT01
DSNDB01.SCT02
```

Our main goal for the EDM pool is to limit the I/O against the directory and catalog. If the pool is too small, then you will also see increased response times due to the loading of the SKCTs, SKPTs, and DBDs, and re-preparing the dynamic SQL statements because they could not remain cached. By correctly sizing the EDM pools you can avoid unnecessary I/Os from accumulating for a transaction. If a SKCT, SKPT or DBD has to be reloaded into the EDM pool this is additional I/O. This can happen if the pool pages are stolen because the EDM pool is too small. Pages in the pool are maintained on an LRU queue, and the least recently used pages get stolen if required. A DB2 performance monitor statistics report can be used to track the statistics concerning the use of the EDM pools.

If a new application is migrating to the environment it may be helpful to look in SYSIBM.SYSPACKAGES to give you an idea of the number of packages that may have to exist in the EDM pool and this can help determine the size.

Efficiency

We can measure the following ratios to help us determine if our EDM pool is efficient. Think of these as EDM pool hit ratios:

- CT requests versus CTs not in EDM pool
- PT requests versus PTs not in EDM pool
- DBD requests versus DBDs not in EDM pool

What you want is a value of 5 for each of the above (1 out of 5). An 80% hit ratio is what you are aiming for.

Dynamic SQL Caching

If we are going to use dynamic SQL caching we are going to have to pay attention to our EDM statement cache pool size. Cached statements are not backed by disk and if its pages are stolen, and the statement is reused, it will have to be prepared again. Static plans and packages can be flushed from EDM by LRU but are backed by disk and can be retrieved when used again. There are statistics to help monitor cache use and trace fields show effectiveness of cache, and can be seen on the Statistics Long Report. In addition, the dynamic statement cache can be “snapped” via the EXPLAIN STMTCACHE ALL Explain statement. The results of this statement are placed in the DSN_STATEMENT_CACHE_TABLE described earlier in this chapter.

In addition to the global dynamic statement caching in a subsystem, an application can also cache statements at the thread level via the KEEP DYNAMIC(YES) bind parameter in combination with not re-preparing the statements. In these situations the statements are cached at the thread level in thread storage as well as at the global level. As long as there is not a shortage of virtual storage the local application thread level cache is the most efficient storage for the prepared statements. The MAXKEEPD subsystem parameter can be used to limit the amount of thread storage consumed by applications caching dynamic SQL at the thread level.

Logging

Every system has some component that will eventually become the final bottleneck. Logging is not to be overlooked when trying to get transactions through the systems in a high performance environment. Logging can be tuned and refined, but the synchronous I/O associated with logging and commits will always be there.

Log Reads

When DB2 needs to read from the log it is important that the reads perform well because reads are normally performed during recovery, restarts and rollbacks — processes that you do not want taking forever. An input buffer will have to be dedicated for every process requesting a log read. DB2 will first look for the record in the log output buffer. If it finds the record there it can apply it directly from the output buffer. If it is not in the output buffer then DB2 will look for it in the active log data set and then the archive log data set. When it is found the record is moved to the input buffer so it can be read by the requesting process. You can monitor the successes of reads from the output buffers and active logs in the statistics report. These reads are the better performers. If the record has to be read in from the archive log, the processing time will be extended. For this reason it is important to have large output buffers and active logs.

Log Writes

Applications move log records to the log output buffer using two methods — no wait or force. The 'no wait' method moves the log record to the output buffer and returns control to the application, however if there are no output buffers available the application will wait. If this happens it can be observed in the statistics report when the UNAVAILABLE ACTIVE LOG BUFF has a non-zero value. This means that DB2 had to wait to externalize log records due to the fact that there were no available output log buffers. Successful moves without a wait are recorded in the statistics report under NO WAIT requests.

A force occurs at commit time and the application will wait during this process, which is considered a synchronous write.

Log records are then written from the output buffers to the active log datasets on disk either synchronously or asynchronously. To know how often this happens you can look at the WRITE OUTPUT LOG BUFFERS in the statistics report.

In order to improve the performance of the log writes there are a few options. First we can increase the number of output buffers available for writing active log datasets, which is performed by changing an installation parameter (OUTBUFF). You would want to increase this if you are seeing that there are unavailable buffers. Providing a large buffer will improve performance for log reads and writes.

Locking and Contention

DB2 uses locking to ensure consistent data based on user requirements and to avoid losing data updates. You must balance the need for concurrency with the need for performance. If at all possible, you want to minimize:

SUSPENSION An application process is suspended when it requests a lock that is already held by another application process and cannot be shared. The suspended process temporarily stops running.

TIME OUT An application process is said to time out when it is terminated because it has been suspended for longer than a preset interval. DB2 terminates the process and returns a -911 or -913 SQL code.

DEADLOCK A deadlock occurs when two or more application processes hold locks on resources that the others need and without which they cannot proceed.

To monitor DB2 locking problems:

- Use the `-DISPLAY DATABASE` command to find out what locks are held or waiting at any moment
- Use `EXPLAIN` to monitor the locks required by a particular SQL statement, or all the SQL in a particular plan or package
- Activate a DB2 statistics class 3 trace with IFCID 172. This report outlines all the resources and agents involved in a deadlock and the significant locking parameters, such as lock state and duration, related to their requests
- Activate a DB2 statistics class 3 trace with IFCID 196. This report shows detailed information on timed-out requests, including the holder(s) of the unavailable resources

The way DB2 issues locks is complex. It depends on the type of processing being done, the `LOCKSIZE` parameter specified when the table was created, the isolation level of the plan or package being executed, and the method of data access.

Thread Management

The thread management parameters on DB2 install panel `DSNTIPE` control how many threads can be connected to DB2, and determine main storage size needed. Improper allocation of the parameters on this panel directly affect main storage usage. If the allocation is too high, storage is wasted. If the allocation is too low, performance degradation occurs because users are waiting for available threads.

You can use a DB2 performance trace record with IFCID 0073 to retrieve information about how often thread create requests wait for available threads. Starting a performance trace can involve substantial overhead, so be sure to qualify the trace with specific IFCIDS, and other qualifiers, to limit the data collected.

The `MAX USERS` field on `DSNTIPE` specifies the maximum number of allied threads that can be allocated concurrently. These threads include TSO users, batch jobs, IMS, CICS, and tasks using the call attachment facility. The maximum number of threads that can be accessing data concurrently is the sum of this value and the `MAX REMOTE ACTIVE` specification. When the number of users trying to access DB2 exceeds your maximum, plan allocation requests are queued.

The DB2 Catalog and Directory

Make sure the DB2 catalog and directory are on separate volumes. Even better, make sure the volumes are completely dedicated to the catalog and directory for best performance. If you have indexes on the DB2 catalog, place them on a separate volume as well.

You can reorg the catalog and directory. You should periodically run RUNSTATs on the catalog and analyze the appropriate statistics that let you know when you need to reorg.

Also, consider isolating the DB2 catalog and directory into their own buffer pool.

Understanding and Tuning Your Packaged Applications

Many organizations today do not retain the manpower resources to build and maintain their own custom applications. In many situations, therefore, these organizations are relying on “off the shelf” packaged software solutions to help run their applications and automate some of their business activities such as accounting, billing, customer management, inventory management, and human resources. Most of the time these packaged application are referred to as enterprise resource planning (ERP) applications. This chapter will offer some tips as to how to manage and tune your DB2 database for a higher level of performance with these ERP applications.

Database Utilization and Packaged Applications

You should keep in mind that these ERP applications are intentionally generic by design. They are also typically database agnostic, which means that they are not written to a specific database vendor or platform, but rather they are written with a generic set of standardized SQL statements that can work on a variety of database management systems and platforms. So, you should assume that your ERP application is taking advantage of few if any of the DB2 SQL performance features, or specific database performance features, table or index designs.

These packaged applications are also typically written using an object oriented design, and are created in such a way that they can be customized by the organization implementing the software. This provides a great flexibility in that many of the tables in the ERP applications' database can be used in a variety of different ways, and for a variety of purposes. With this great flexibility also comes the potential for reduced database performance. OO design may lead to an increase in SQL statements issued, and the flexibility of the implementation could have an impact on table access sequence, random data access, and mismatching of predicates to indexes.

Finally, when you purchase an application you very typically have no access to the source code, or the SQL statements issued. Sometimes you can get a vendor to change a SQL statement based upon information you provided about a performance problem, but this is typically not the case. So, with that in mind the majority of your focus should be on first getting the database organized in a manner that best accommodates the SQL statements, and then tune the subsystem to provide the highest level of throughput. Of course, even though you can't change the SQL statements you should first be looking at those statements for potential performance problems that can be improved by changing the database or subsystem.

Finding and Fixing SQL Performance Problems

The first thing that needs to be done when working with a packaged application is to determine where the performance problems exist. It is most likely that you will not be able to change the SQL statements, as this will require getting the vendor involved and having them make a change that will improve a query for you, and not negatively impact performance for other customers. Even though you won't be able to change the SQL statements, you can change subsystem parameters, memory, and even change the tables and indexes for performance. Before you begin your tuning effort you have to decide if you are tuning in order to improve the response time for end users, or if you are tuning to save CPU dollars.

The best way to find the programs and SQL statements that have the potential for elapsed and CPU time savings is to utilize the technique described in Chapter 6 called "Overall Application Performance Monitoring". This would be the technique for packaged applications that are using static embedded SQL statements. However, if your packaged application is utilizing dynamic SQL then this technique will be less effective. In that case you are better off utilizing one or more of the techniques outlined in Chapter 3 "Recommendations for Distributed Dynamic SQL".

Whether or not the application is using static or dynamic SQL it's best to capture all of the SQL statements being issued, and cataloging them in some sort of document or perhaps even in a DB2 table. This can be achieved by querying SYSIBM.SYSSTMT table by plan for statements in a plan, or the SYSIBM.SYSPACKSTMT table by package for statements in a package. You can query these tables using the plan or package names corresponding to the application. If the application is utilizing dynamic SQL then you can capture the SQL statements by utilizing EXPLAIN STMTCACHE ALL (DB2 V8 or DB2 9), or by running a trace (DB2 V7, DB2 V8, DB2 9). Of course if you run a trace you can expect to have to parse through a significant amount of data. Keep in mind that by capturing these dynamic statements you are only seeing the statements that have been executing during the time you've monitored. For example, using EXPLAIN STMTCACHE ALL statement will only capture the statements that are residing in the dynamic statement cache at the time the statement is executed.

Once you've determined the packages and/or statements that are consuming the most resources, or have the highest elapsed time, then you can begin your tuning effort. This will involve subsystem and/or database tuning. You need to make a decision at this point as subsystem tuning really has no impact on the application and database design. However, if you make changes to the database, then you need to consider the fact that future product upgrades or releases can undo the changes you have done.

Subsystem Tuning for Packaged Applications

The easiest way to tune packaged applications is by performing subsystem tuning. This could involve simply changing some subsystem parameters, or perhaps increasing memory. The impact of the changes can be immediate, and in situations in which the subsystem is poorly configured the impact can be dramatic. Please refer to Chapter 8 on subsystem tuning for the full details of tuning such things as:

- **Buffer Pools** Buffer tuning can be a major contributor to performance with packaged applications. Consider first organizing your buffers according to the general recommendations in Chapter 8 of this guide. Then you can take a closer look at the access to individual page sets within the pools. You can use the `-DISPLAY BUFFERPOOL` command to get information about which page sets are accessed randomly versus sequentially. You can use this information to further separate objects based upon their access patterns. You can identify the code tables, and get them all in their own buffer pool with enough memory to make them completely memory resident. You could also identify the most commonly accessed page sets and separate them into their own buffer pool. Once you've made these changes you can increase the amount of memory in the most highly accessed buffer pools (especially for indexes), and monitor the buffer hit ratio for improvement. Continue to increase the memory until you get a diminishing return on the improved hit ratio or you run out of available real memory.
- **Dynamic Statement Cache** Is the application using dynamic SQL? If so, then you most certainly should make sure that the dynamic statement cache is enabled, and given enough memory to avoid binding on PREPARE. You can monitor the statement cache hit ratio by looking at a statistics report. Statement prepare time can be a significant contributor to overall response time, and you'll want to minimize that. If the statement prepare time is a big factor in query performance you could consider adjusting some DB2 installation parameters that control the bind time. These parameters, `MAX_OPT_STOR`, `MAX_OPT_CPU`, `MAX_OPT_ELAP`, and `MXQBCE` (DB2 V8, DB2 9) are what is known as "hidden" installation parameters and will help control statement prepare costs, but most certainly should be adjusted only under the advice of IBM.
- **RID Pool** Make sure to look at your statistics report to see how much the RID pool is utilized, and if there are RID failures due to lack of storage. If so, you should increase the size of the RID pool.
- **DB2 Catalog** If it isn't already, the DB2 System Catalog should be in its own buffer pool. If the application is using dynamic SQL then this pool should be large enough to avoid I/O for statement binds.
- **Workfile Database and Sort Pool** Your packaged application is probably doing a lot of sorting. If this is the case then you should make sure that your sort pool is properly sized to deal with lots of smaller sorts. In addition, you should have several workfile table spaces created in order to avoid resource contention between concurrent processes.

- **Memory and DSMAX** These packaged applications typically have a lot of objects. You can monitor to see which objects are most commonly accessed, and make those objects CLOSE NO. Make sure your DSMAX installation parameter is large enough to avoid any closing of datasets, and if you have enough storage make all of the objects CLOSE NO. You can also make sure your output log buffer is large enough to avoid shortages if the application is changing data quite often. You can consider using the KEEP DYNAMIC bind option for the packages, but keep in mind that this can increase thread storage, and you'll need to balance the number of threads with the amount of virtual storage consumed by the DBM1 address space. Make sure your buffer pools are page fixed.

Please refer to Chapter 8 of this guide for more subsystem tuning options. You can also refer to the IBM redbook entitled "DB2 UDB for z/OS V8: Through the Looking Glass and What SAP Found There" for additional tips on performance.

Table and Index Design Options for High Performance

Chapter 4 of this guide should be reviewed for general table and index design for performance. However, there are certain areas that you can focus on for your packaged applications. The first thing you have to do is find the portions of the application that have the biggest impact on performance. You can use the techniques outlined in Chapters 3 and 6 of this guide to do that. Finding the packages and statement that present themselves as performance problems can then lead you to the most commonly accessed tables. Taking a look at how these tables are accessed by the programs (DISPLAY BUFFERPOOL command, accounting report, statistics report, performance trace, EXPLAIN) can give you clues as to potential changes for performance. Are there a lot of random pages accessed? Are matching columns of an index scan not matching on all available columns? Are there more columns in a WHERE clause than in the index? Does it look like there is repetitive table access? Are the transaction queries using list prefetch? If you are finding issues like these you can take action. In general, there are two major things you can change:

- **Indexes** These packaged applications are generic in design, but your use of them will be customized for your needs. This means that the generic indexes should be adjusted to fit your needs. The easiest thing to do is add indexes in support of poor performing queries. However, you should be aware of the impact that adding indexes will have on inserts, deletes, possibly updates, and some utilities. That's why having a catalog of all the statements is important. Other things you can do is to change the order of index columns to better match your most common queries in order to increase matchcols, as well as adding columns to indexes to increase matchcols. You can also change indexes that contain variable length columns to use the NOT PADDED (DB2 V8, DB2 9) option to improve performance of those indexes.
- **Clustering** Since the packaged application can't anticipate your most common access paths you could have a problem with clustering. Understanding how the application accesses the tables is important, but you can look at joins that occur commonly and consider changing the clustering of those tables so that they match. This can have a dramatic impact on the performance of these joins, but you have to make sure that no other processes are negatively impacted. If tables are partitioned for a certain reason you could consider clustering within each partition (DB2 V8, DB2 9) to keep the separation of data by partition, but improve the access within each partition. Look for page sets that have a very high percentage of randomly accessed pages in the buffer pool as a potential opportunity to change clustering.

Of course you need to always be aware of the fact that any change you make to the database can be undone on the next release of the software. Therefore, you should document every change completely!

Monitoring and Maintaining Your Objects

The continual health of your objects is important. Keep a list of commonly accessed objects, especially those that are changed often, and begin a policy of frequent monitoring of these objects. This involves regularly running the RUNSTATS utility on these frequently changing indexes and table spaces. Performance can degrade quickly as these objects become disorganized. Make sure to set up a regular strategy of RUNSTATS, REORGS, and REBINDs (for static SQL) to maintain performance. Understanding the quantity of inserts to a table space can also guide you to adjusting the free space. Proper PCTFREE for table space can help avoid expensive exhaustive searches for inserts. Proper PCTFREE is very important for indexes where inserts are common. You don't want these applications to be splitting pages so set the PCTFREE high, especially if there is little or no sequential scanning of the index.

Real Time Statistics

You can utilize DB2's ability to collect statistics in real time to help you monitor the activity against your packaged application objects. Real time statistics allows DB2 to collect statistics on table spaces and index spaces and then periodically write this information to two user-defined tables (as of DB2 9 these tables are an integral part of the system catalog). The statistics can be used by user written queries/programs, a DB2 supplied stored procedure or Control Center, to make decisions for object maintenance.

STATISTICS COLLECTION DB2 is always collecting statistics for database objects. The statistics are kept in virtual storage and are calculated and updated asynchronously upon externalization. In order to externalize them the environment must be properly set up. A new set of DB2 objects must be created in order to allow for DB2 to write out the statistics.

SDSNSAMP(DSNTESS) contains the information necessary to set up these objects.

There are two tables (with appropriate indexes) that must be created to hold the statistics:

- SYSIBM.TABLESPACESTATS
- SYSIBM.INDEXSPACESTATS

These tables are kept in a database named DSNRTSDB, which must be started in order to externalize the statistics that are being held in virtual storage. DB2 will then populate the tables with one row per table space or index space, or one row per partition. For tables that are shared in a data-sharing environment, each member will write its own statistics to the RTS tables.

Some of the important statistics that are collected for table spaces include: total number of rows, number of active pages, and time of last COPY, REORG, or RUNSTATS execution. Some statistics that may help determine when a REORG is needed include: space allocated, extents, number of inserts, updates, or deletes (singleton or mass) since the last REORG or LOAD REPLACE, number of unclustered inserts, number of disorganized LOBs, number of overflow records created since last REORG. There are also statistics to help for determining when RUNSTATS should be executed. These include: number of inserts/updates/deletes (singleton and mass) since the last RUNSTATS execution. Statistics collected to help with COPY determination include: distinct updated pages and changes since the last COPY execution and the RBA/LRSN of first update since last COPY.

There are also statistics gathered on indexes. Basic index statistics include: total number of entries (unique or duplicate), number of levels, number of active pages, space allocated and extents. Statistics that help to determine when a REORG is needed include: time when the last REBUILD, REORG or LOAD REPLACE occurred. There are also statistics regarding the number of updates/deletes (real or pseudo, singleton or mass)/inserts (random and those that were after the highest key) since the last REORG or REBUILD. These statistics are of course very helpful for determining how our data physically looks after certain process (i.e. batch inserts) have occurred so we can take appropriate actions if necessary.

EXTERNALIZING AND USING REAL TIME STATISTICS There are different events that can trigger the externalization of the statistics. DSNZPARM STATSINST (default 30 minutes) is used to control the externalization of the statistics at a subsystem level.

There are several processes that will have an effect on the real time statistics. Those processes include: SQL, Utilities and the dropping/creating of objects.

Once externalized, queries can then be written against the tables. For example, a query against the TABLESPACESTATS table can be written to identify when a table space needs to be copied due to the fact that greater than 30 percent of the pages have changed since the last image copy was taken.

```
SELECT NAME
FROM SYSIBM.SYSTABLESPACESTATS
WHERE DBNAME = 'DB1' and
((COPYUPDATEDPAGES*100)/NACTIVE) > 30
```

This table can be used to compare the last RUNSTATS timestamp to the timestamp of the last REORG on the same object to determine when RUNSTATS is needed. If the date of the last REORG is more recent than the last RUNSTATS, then it may be time to execute RUNSTATS.

```

SELECT NAME
FROM SYSIBM.SYSTABLESPACESTATS
WHERE DBNAME = 'DB1' and
      (JULIAN_DAY (REORGLASTTIME) > JULIAN_DAY (STATSLASTTIME) )

```

This last example may be useful if you want to monitor the number of records that were inserted since the last REORG or LOAD REPLACE that are not well-clustered with respect to the clustering index. Ideally, 'well-clustered' means the record was inserted into a page that was within 16 pages of the ideal candidate page (determined by the clustering index). The SYSTABLESPACESTATS table value REORGUNCLUSTINS can be used to determine whether you need to run REORG after a series of inserts.

```

SELECT NAME
FROM SYSIBM.SYSTABLESPACESTATS
WHERE DBNAME = 'DB1' and
      ((REORGUNCLUSTINS*100) / TOTALROWS) > 10

```

There is also a DB2 supplied stored procedure to help with this process, and possibly even work toward automating the whole determination/utility execution process. This stored procedure, DSNACCOR, is a sample procedure which will query the RTS tables and determine which objects need to be reorganized, image copied, updated with current statistics, have taken too many extents, and those which may be in a restricted status. DSNACCOR creates and uses its own declared temporary tables and must run in a WLM address space. The output of the stored procedure provides recommendations by using a predetermined set of criteria in formulas that use the RTS and user input for their calculations. DSNACCOR can make recommendations for everything (COPY, REORG, RUNSTATS, EXTENTS, RESTRICT) or for one or more of your choice and for specific object types (table spaces and/or indexes).

Page intentionally left blank

Tuning Tips

Back against a wall? Not sure why DB2 is behaving in a certain way, or do you need an answer to a performance problem. It's always good to have a few ideas or tricks up your sleeve. Here are some DB2 hints and tips in no particular order.

Code DISTINCT Only When Needed

The use of DISTINCT can cause excessive sorting, which can cause queries to become very expensive. Also, the use of DISTINCT in a table expression forces materialization. Only a unique index can help to avoid sorting for a DISTINCT.

We often see DISTINCT coded when it is not necessary and sometimes it comes from a lack of understanding of the data and/or the process. Sometimes code generators create a DISTINCT after every SELECT clause, no matter where the SELECT is in a statement. Also we have seen some programmers coding DISTINCTs just as a safeguard. When considering the usage of DISTINCT, the question to first ask is 'Are duplicates even possible?' If the answer is no, then remove the DISTINCT and avoid the potential sort.

If duplicates are possible and not desired then use DISTINCT wisely. Try to use a unique index to help avoid a sort. Consider Using DISTINCT as early as possible in complex queries in order to get rid of the duplicates as early as possible. Also, avoid using DISTINCT more than once in a query. A GROUP BY all columns can utilize non-unique indexes possibly avoid a sort. Coding a GROUP BY all columns can be more efficient than using DISTINCT, but should be carefully documented in your program and/or statement.

Don't Let Java in Websphere Default the Isolation Level

ISOLATION(RS) is the default for Websphere applications. RS (Read Stability) is basically RR (Repeatable Read) with inserts allowed. RS tells DB2 that when you read a page you intend to read it or update it again later, and you want no one else to update that page until you commit, causing DB2 to take share locks on pages and hold those locks. This is rarely ever necessary, and can definitely cause increased contention.

Using Read Stability can lead to other known problems, such as the inability to get lock avoidance, and potential share lock escalations because DB2 may escalate from an S lock on the page to an S lock on the tablespace. Another problem that has been experienced is that searched updates can deadlock under isolation RS. When an isolation level of RS is used, the search operation for a searched update will read the page with an S lock. When it finds data to update then it will change the S lock to an X lock. This could be exaggerated by a self-referencing correlated subquery in the update (e.g. updating the most recent history or audit row). This is because in that situation DB2 will read the data with an S lock, put the results in a workfile with the RIDs, and then go back to do the update in RID sequence. As the transaction volume increases these deadlocks are more likely to occur. These problems have been experienced with applications that are running under Websphere and allowing it to determine the isolation level, which defaults to RS.

So how do you control this situation? Well some possible solutions include the following.

1. Change the application server connect to use an isolation level of CS rather than RS. This can be done by setting a JDBC database connection property. This is the best option, but often the most difficult to get implemented.
2. Rebind the isolation RS package being used, with an isolation level of CS. This is a dirty solution because there may be applications that require RS, or when the DB2 client software is upgraded it may result in a new RS package (or a rebind of the package), and we'll have to track that and perform a special rebind with every upgrade on every client.
3. Have the application change the statement to add WITH CS.
4. There is a zparm RRULOCK=YES option which acquires U, versus S locks for update/delete with ISO(RS) or ISO(RR). This could help to avoid deadlocks for some update statements.

Through researching this issue we have found no concrete reason why this is the default in Websphere, but for DB2 applications this is wasteful and should be changed.

Locking a Table in Exclusive Mode Will NOT Always Prevent Applications from Reading It

From some recent testing both in V7 and V8 we found that issuing the LOCK TABLE <table name> IN EXCLUSIVE MODE would not prevent another application from reading the table either with UR or CS.

In V7, this was the case if the tablespace was defined with LOCKPART(YES), and the V8 test results were the same because now LOCKPART(YES) is the default for the tablespace.

Once an update was issued then the table was not readable.

This is working as designed and the reason for this tip is to make sure that applications are aware that just issuing this statement does NOT immediately make a table unreadable.

Put Some Data in Your Empty Control Table

Do you have a table that is used for application control? Is this table heavily accessed, but usually has no data in it? We often use tables to control application behavior with regards to availability or special processing. Many times these control tables are empty, and will only contain data when we want the application to behave in an unusual way. In testing with these tables we discovered that you do not get index lookaside on an empty index. This can add a significant number of getpage operations for a very busy application. If this is the case then you should see if the application can tolerate some fake data in the table. If so you'll then get index lookaside and save a little CPU.

Use Recursive SQL to Assign Multiple Sequence Numbers in Batch

In some situations you may have a process that adds data to a database, but this process is utilized by both online and batch applications. If you are using sequence objects to assign system generated key values this works quite well for online, but can get expensive in batch. Instead of creating an object with a large increment value why not keep an increment of 1 for online, and grab a "batch" of sequence numbers in batch. You can do this by using recursive SQL. Simply write a recursive common table expression that generates a number of rows equivalent to the number of sequence values you want in a batch, and then as you select from the common table expression you can get the sequence values. Using the appropriate settings for block fetching in distributed remote, or by using multi-row fetch in local batch, you can significantly reduce the cost of getting these sequence values for a batch process. Here's what a SQL statement would look like for 1000 sequence values:

```
WITH GET_THOUSAND (C) AS
  (SELECT 1
   FROM SYSIBM.SYSDUMMY1
   UNION ALL
   SELECT C+1
   FROM   GET_THOUSAND
   WHERE  C < 1000)
SELECT NEXT VALUE FOR SEQ1
FROM   GET_THOUSAND;
```

Code Correlated Nested Table Expression versus Left Joins to Views

In some situations DB2 may decide to materialize a view when you are left joining to that view. If this is happening it is not likely that you'll be able to get the performance desired, especially for transaction environments when you transactions are processing little or not data. Consider the following view and query:

```
CREATE VIEW HIST_VIEW (ACCT_ID, HIST_DTE, ACCT_BAL) AS
  (SELECT ACCT_ID, HIST_DTE, MAX(ACCT_BAL)
   FROM   ACCT_HIST
   GROUP BY ACCT_ID, HIST_DTE);

SELECT A.ACCT_ID, A.CUST_NME, B.HIST_DTE, B.ACCT_BAL
FROM   ACCOUNT A
LEFT OUTER JOIN
      HIST_VIEW B
ON A.ACCT_ID = B.ACCT_ID
WHERE A.ACCT_ID = 1110087;
```

Consider instead the following query which will strongly encourage DB2 to use the index on the ACCT_ID column of the ACCT_HIST table:

```
SELECT A.ACCT_ID, A.CUST_NME, B.HIST_DTE, B.ACCT_BAL
FROM   ACCOUNT A
LEFT OUTER JOIN
      TABLE(SELECT ACCT_ID, HIST_DTE, MAX(ACCT_BAL)
            FROM   ACCT_HIST X
            WHERE  X.ACCT_ID = A.ACCT_ID
            GROUP BY ACCT_ID, HIST_DTE) AS B
ON A.ACCT_ID = B.ACCT_ID
WHERE A.ACCT_ID = 1110087;
```

This recommendation is not limited to aggregate queries in views, but can apply to many situations. Give it a try!

Use GET DIAGNOSTICS Only When Necessary for Multi-Row Operations

Multi-row operations can be a huge CPU saver, as well as an elapsed time saver for sequential applications. However, the GET DIAGNOSTICS statement is one of the most expensive statements you can use in an application at about three times the cost of the other statements. So, you don't want to issue a GET DIAGNOSTICS after every multi-row statement. Instead use the SQLCA first. If you get a non-negative SQLCODE from a multi-row operation then you can use the GET DIAGNOSTICS statement to identify the details of the failure. For multi-row fetch you can use the SQLERRD3 field to determine how many rows were retrieved when you get a SQLCODE 100, so you never need to use a GET DIAGNOSTICS for multi-row fetch.

Database Design Tips for High Concurrency

In order to design high concurrency in a database there are a few tips to follow. Most of the recommendations must be considered during design before the tables and table spaces are physically implemented because to change after the data is in use would be difficult.

- Use segmented or universal table spaces, not simple
 - Will keep rows of different tables on different pages, so page locks only lock rows for a single table
- Use LOCKSIZE parameters appropriately
 - Keep the amount of data locked at a minimum unless the application requires exclusive access
- Consider spacing out rows for small tables with heavy concurrent access by using MAXROWS =1
 - Row level lock could also be used to help with this but the overhead is greater, especially in a data sharing environment
- Use partitioning where possible
 - Can reduce contention and increase parallel activity for batch processes
 - Can reduce overhead in data sharing by allowing for the use of affinity routing to different partitions
 - Locks can be taken on individual partitions
- Use Data Partitioned Secondary Indexes (DPSI)
 - Promotes partition independence and less contention for utilities
- Consider using LOCKMAX 0 to turn off lock escalation
 - In some high volume environments this may be necessary. NUMLKTS will need to be increased and the applications must commit frequently

- Use volatile tables
 - Reduces contention because an index will also be used to access the data by different applications that always access the data in the same order
- Have an adequate number databases
 - Reduce DBD locking if DDL, DCL and utility execution is high for objects in the same database
- Use sequence objects
 - Will provide for better number generation without the overhead of using a single control table

Database Design Tips for Data Sharing

In DB2 data sharing the key to achieving high performance and throughput is to minimize the amount of actual sharing across members and to design databases/applications to take advantage of DB2 and data sharing. The best performing databases in data sharing are those that are properly designed. The following are some tips for properly designing database for data sharing use:

- Proper Partitioning Strategies
 - Advantages
 - Processing by key range
 - Reducing or avoiding physical contention
 - Reducing or avoiding logical contention
 - Parallel processes are more effective
- Appropriate clustering and partitioning allows for better cloning of applications across members
 - Access is predetermined and sequential
 - Key ranges can be divided among members
 - Can better spread workload
 - Reduce contention and I/O
 - Can run utilities in parallel across members

Tips for Avoiding Locks

Lock avoidance has been around since Version 3, but are you getting it? Lock avoidance can be a key component to high performance because to take an actual lock is about 400 CPU instructions and 540 bytes of memory, to avoid a lock a latch is taken by the buffer manager and the cost is significantly less. The process of lock avoidance compares the page log RBA (last time a change was made on the page) to the CLSN(Commit Log Sequence Number) on the log (last time all changes to the page set were committed) and then also checks the PUNC (Possibly Uncommitted) bit on the page, and if it finally determines there are no outstanding changes on the page, then the a latch is taken, instead of a lock. However, there are some prerequisites to getting lock avoidance.

- Up to V8 you need to use CURRENTDATA(NO) on the bind. As of V8/V9 CURRENTDATA(YES) will allow the process of detection to begin
- You need to be bound ISOLATION(CS)
- Have a read-only cursor
- Commit often
 - This is the most important point because all processes have to commit there changes in order for the CLSN to get set. If the CLSN never gets set then this process can never work and locks will always have to be taken. Hint: you can use the URCHKTH DSNZPARM to find applications that are not committing

CA, one of the world's largest information technology (IT) management software companies, unifies and simplifies complex IT management across the enterprise for greater business results. With our Enterprise IT Management vision, solutions and expertise, we help customers effectively govern, manage and secure IT.

HB05ESMDBM01E MP319340807