

January 2014

The DB2 for z/OS Performance Handbook



Daniel L. Luksetich
DB2 Consultant

Notices

Copyright ©2014 by CA. All rights reserved. All trademarks, trade names, service marks and logos referenced herein belong to their respective companies.

The information in this publication could include typographical errors or technical inaccuracies, and the authors assume no responsibility for its accuracy or completeness. The statements and opinions expressed in this book are those of the authors and are not necessarily those of CA, Inc. ("CA"). CA may make modifications to any CA product, software program, method or procedure described in this publication at any time without notice.

Any reference in this publication to third-party products and websites is provided for convenience only and shall not serve as the authors' endorsement of such products or websites. Your use of such products, websites, any information regarding such products or any materials provided with such products or on such websites shall be at your own risk.

To the extent permitted by applicable law, the content of this book is provided "AS IS" without warranty of any kind, including, without limitation, any implied warranties of merchantability, fitness for a particular purpose, or non-infringement. In no event will the authors or CA be liable for any loss or damage, direct or indirect, arising from or related to the use of this book, including, without limitation, lost profits, lost investment, business interruption, goodwill or lost data, even if expressly advised in advance of the possibility of such damages. Neither the content of this book nor any software product referenced herein serves as a substitute for your compliance with any laws (including but not limited to any act, statute, regulation, rule, directive, standard, policy, administrative order, executive order, and so on (collectively, "Laws") referenced herein or otherwise. You should consult with competent legal counsel regarding any such Laws.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by an information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

Table of Contents

Notices	2
About CA Technologies	6
Introduction	7
Section 1: The Basics of Performance and Performance Tuning	9
System Tuning versus Application Tuning	
Application Design for Performance and SQL Tuning	
Database Design for Performance	
Summary	
Section 2: SQL Access Paths	12
The Critical Importance of Appropriate Catalog Statistics	
DB2 Engine Overview	
Predicate Types	
Query Transformation and Access Path Selection	
Single Table Access Paths	
Multi-Table Access Paths	
Nested Table Expression Access	
Query Parallelism	
Sequential Detection and Index Lookaside	
Section 3: Coding SQL for Performance	42
The Most Efficient SQL Statement	
The Basics of Coding SQL Statements for Performance	
Subqueries and Joins	
Table Expressions	
SQL Features and Performance	
Dynamic SQL and Performance	

Table of Contents

Section 4: Utilizing DB2 EXPLAIN

67

EXPLAIN Processing

Interpreting EXPLAIN Results

The PLAN_TABLE

Externalizing the Dynamic Statement Cache via EXPLAIN

Section 5: Controlling SQL Access Paths

74

Parallelism

Maintaining Access Paths

Optimization Hints

Influencing the Optimizer

Section 6: Table and Index Design for High Performance

80

Selecting a Table Space Type and Features

Table Design Features and Performance

Index Choices

Keys and Clustering

Design Recommendations for High Performance Insert

Minimizing Change

Denormalization

Section 7: Memory, Subsystem Parameters and Performance

109

Buffers

Logging

DASD and I/O

Big Impact Subsystem Parameters

zIIP Engines

Table of Contents

Section 8: Monitoring Subsystem and Application Performance 122

DB2 Traces

Understanding and Reporting Trace Output

DB2 Performance Database

Displaying Buffer Statistics

Section 9: Testing Performance Designs and Decisions 129

Establishing a Test Database

Generating Data and Statements

Simple Performance Benchmarks

Section 10: Top Tuning Tips 140

Start with the Given Design

Deploy Indexes Slowly

Develop a Table Access Sequence

Update at the End

Always Test Performance Changes

Put Some Data in That Table

Get a Sponsor

Take Advantage of SQL Features

Use One Authid per Application

Stick a Comment in that Statement

Use Query Numbers

Take Advantage of Cache

Grab Those Resources

Understand the Trade Offs

Section 11: About the Author 142

About CA Technologies

CA Technologies (NASDAQ: CA) provides IT management solutions that help customers manage and secure complex IT environments to support agile business services. Organizations leverage CA Technologies software and SaaS solutions to accelerate innovation, transform infrastructure and secure data and identities, from the data center to the cloud.

Solutions from CA Technologies are the critical foundation for helping customers gain deep insight into and exceptional control over their complex, mixed IT environments. Our solutions enable customers to support changing business models at the right time, the right place, and from the best sources, and flexibly deliver IT services in the manner that best serves business needs.

Our solutions and services are the foundational elements for the deep insight and exceptional control necessary to make the promise of flexible IT and agile business a reality.

To learn firsthand how CA Technologies helps our customers succeed, please go to ca.com/success.

Introduction

IBM has made a significant effort to improve performance with every release of the product, as well as introduce features that automate performance tuning. The things we worried about just a few years ago seem like distant memories and we need to switch focus somewhat from micro-managing things like specific access paths, buffers and DASD response times, and to focus more on proper table and index design, reduction in SQL calls, proper SQL coding techniques and monitoring application performance. It's not that system monitoring and tuning are not important, they are. However, there have been considerable advances in hardware performance and subsystem performance such that they should no longer be the primary focus of the DBA. Given the proliferation of application development automation, the majority of DBA time today should be focused on making sure generic designs do not negatively impact the demands of high performance processes. I'm not against development tools; they are the norm now. However, we do need to balance time to delivery with performance, especially considering high volumes of data and high transaction rates.

About The DB2 Performance Content in This Book

There is a DB2 for z/OS Managing Performance Guide, and here are DB2 for z/OS performance redbooks. These valuable and often under utilized references are extremely important tools in your day to day performance tuning efforts. This book is not a repeat or condensation of that material, nor is it a replacement for that material. The intention of this book is to complement the printed performance information that is part of the DB2 documentation and to give the reader some insight on particular situations where certain features are of benefit or detriment. Therefore, it is the reader's responsibility to cross-reference and research each feature and technique discussed in this book. Also, remember that nothing is free. With any deployment of any technique or feature there is going to be a cost to pay. I'll attempt to balance those choices in this book.

This book is heavily focused on application, database, and SQL design and tuning for performance. Systems tuning and performance will be secondary to application and database tuning. It's not that system tuning is not important, and you should take system tuning seriously. I will address some of the big buttons in this book.

This is a performance book and not a reference guide. What this means is that this book should be read from start to finish, and not used to look up various features to learn something about performance. There are many features that are addressed in multiple sections of this book, as well as settings, techniques, and features that have a relationship to each other and are not necessarily addressed in one place in the book. You can use this book as a reference, but it is strongly recommended that you read it entirely first!

This book is very much slanted toward transaction processing as opposed to data warehousing and business analytics. DB2 for z/OS is an excellent data warehouse engine. Recent advancements both in query performance and the introduction of the IBM DB2 Analytics Accelerator (IDAA) make it a great and economically feasible database for data warehouses. I'm especially an advocate of running both transaction processing and decision support against the same production database. However, in my personal experience most enterprises are utilizing DB2 for z/OS for transaction processing and not

data warehousing. So, the majority of design and tuning techniques presented here are specifically in support of transaction processing systems. Many of the concepts can be applied to a data warehouse, especially the information included in sections 2, 3, and 4.

There is a balance to everything. This is a performance book. It is much more about performance than flexibility and availability. The concepts of performance, flexibility, and availability are generally opposed to each other, especially when implementation and operation costs are of concern. It is your responsibility to strike a balance in your implementation that is appropriate for your enterprise and situation.

The information provided in this book is relevant through DB2 11 for z/OS. In most feature descriptions I make no distinction between versions when discussing various features. Be sure to check the DB2 manuals for the version you are working with to confirm the availability of specific features.

Section 1:

The Basics of Performance and Performance Tuning

We're constantly striving for high performance, our managers are constantly demanding high performance, and IBM is quite dedicated to delivering a higher level of performance with each release of DB2. With these things in mind we first need to focus on the concept of high performance, and what it means within the context of processing for each individual organization and/or situation.

System Tuning versus Application Tuning

There has always been a division in the eyes of some people with regards to performance tuning. Many DBAs spend a significant amount of time monitoring and tuning DB2 configuration parameters in an attempt to achieve a higher level of performance for the applications using DB2. Certain parameters, such as buffer sizes, can make a huge difference in performance if they are terribly undersized to begin with, but once most parameters are set within a reasonable threshold any changes typically make little to no impact on performance. I believe in the 80/20 rule when it comes to performance tuning. That is, 20% of the focus should be on systems and 80% should be on database and applications¹. When it comes to application performance tuning people should be looking at either SQL or database design. Hopefully, both together!

Application Design for Performance and SQL Tuning

There are two major challenges when it comes to SQL tuning. The first and most important is the quantity of SQL statements issued, and the second is the relative performance of each SQL statement.

Ever hear the saying that the most efficient SQL statement is the one that is never executed? Well, it's true. The biggest challenge in today's fast paced application development environment is keeping the number of SQL statements executed per transaction to a minimum. In many cases, the development utilizes a framework and service-oriented architecture. Many times database tables are mapped one to one with objects in the data services application layer. Repeated calls to a specific service can likewise result in repeated calls to populate an object from a database table. A greedy design where service calls are made repeatedly within a unit of work can result in an extreme degradation in performance. Many times the database is to blame, but database monitors show extreme quantities of reasonably performing SQL statements. There is very little system or database tuning that can be done in these situations. This is why when designing a new database application a focus of the design needs to be a more conscious approach to design for performance. This more performance-conscious approach should take two forms; the first focuses on calling the database only when necessary, and the second involves combining multiple calls into a single call. The first is conceptually easy; if a data object is already populated and there is no need to refresh it then the database call should be avoided. This may require some additional tests within the application code, but the results can be very effective. The second may be a bit

more dramatic, and it is best to do this only when performance is more of a concern versus speed of delivery and the flexibility of the code. I've had great success in designing applications where the focus on multi-table SQL statements was concentrated on the high-performance component of the application, while normal development was in place for the majority of the application logic. For these high performance components you simply need to look at which tables are being accessed during a unit of work, which columns from the tables are required (if at all), and the relationships between the tables. If application SQL complexity is too much for the application developers, or if there is no interest in placing the complexity within the application code, then stored procedures, triggers, and user-defined functions provide a nice alternative for bundling multiple SQL calls into one call. It's almost impossible to tune a simple statement against a single table with one predicate against a primary key column, but with multiple table access more tuning opportunities exist, and in general the more complex SQL statement will almost always outperform many simple statements providing the same functionality. This is especially true across a network. For example, I have tested programmatic joins versus the equivalent SQL joins for two tables and realized a 30% performance gain for the SQL join.

More complex SQL statements offer more opportunities for the tuning of these individual statements, but before you set off looking into complex SQL tuning you need to step back and understand the overall performance of your application. This is where active monitoring of your production DB2 system is important. Hopefully you have a tool that can analyze the appropriate DB2 trace output to produce meaningful performance reports. There are also DB2 components, such as the dynamic statement cache, that can be utilized to produce performance metrics. Whatever the case may be, you should be looking at the performance of applications at the application level, user level, and then statement level². It is important that if you are going to be tuning SQL you should tune the one that consumes the most resources, or the one that exceeds a specified SLA, first. Regular monitoring should be a part of application and SQL performance tuning, especially when changes are being implemented. The ability to show a real world result is crucial to getting management support for tuning.

When tuning SQL statements I typically utilize two techniques to determine the potential savings to be realized from a database or SQL change; the EXPLAIN³ facility and benchmarking. EXPLAIN will take as input a SQL statement and externalize the predicted access path within the database engine. There are several EXPLAIN tables that are populated when a statement is EXPLAINed, so it is best to utilize a tool to analyze the access path. You can, if you are adventurous, write your own query against the EXPLAIN tables, and there are various examples available out on the internet. In addition to providing access path information EXPLAIN also provides statement cost information. This is internal cost information that DB2 uses to determine the access path, and it is also information that you can use to compare the predicted cost of two different SQL statements that are logically equivalent. While comparing the statement cost in EXPLAIN is a good start, it is not always the definitive guide to better SQL performance. So, I generally use cost calculations as one part of the predicted savings of a SQL tuning change. I also look at the access path and perform benchmark testing. The benchmark

testing is usually performed using REXX or SPUFI⁴ on the mainframe. If a test database is properly configured and database statistics updated to⁵ match production statistics, then you can get a pretty good estimate of the potential performance savings by running a benchmark test and capturing the performance using a monitoring tool or the statement cache. I always back any proposed SQL performance change with both EXPLAIN and benchmark test results.

Database Design for Performance

There are a myriad of tuning possibilities when it comes to the database. Before jumping onto any of the touted performance features of DB2 it is critical to first understand the problem that needs to be solved. If you want to partition a table, are you actually hoping that the partitioning results in a performance improvement or is it for data management or flexibility? Clustering of data is critical, but clustering has to reflect potential sequential access for single table, and especially multi-table access. Generic table keys are OK, but clustering is meaningless unless parent keys are used for clustering child table data when processing follows a key-based sequential pattern or when multiple tables are being joined. Natural keys may be preferred, unless compound keys become exceedingly large. There are a number of choices of page sizes, and this can be especially important for indexes where page splitting may be a concern. Tables can also be designed specifically for high volume inserts with such features as “append only” and “member cluster”.

There is no reason to sit in meeting rooms for hours or days trying to come up with a database design that you “think” will perform properly. I personally let the database itself influence the design. This involves setting up test database designs and test conditions that will mock the predicted application and database design. This usually involves generated data and statements that are run through SPUFI, REXX programs, db2batch, or shell scripts, all while performance information is gathered from monitors and compared. Sound complicated? It really isn’t, and most proof-of-concept tests I’ve been involved with have not lasted more than a week or two. The effort is minimal, but the rewards can be dramatic.

Summary

While this has been a high level review of DB2 performance tuning, I hope it serves as an inspiration to look for specific solutions to your performance issues. Don’t be afraid to test things and perform some benchmarks.

Section 2:

SQL Access Paths

The good news is that the DB2 engine is becoming more and more sophisticated with each new release. This means that almost all of the time your SQL statements will execute with the highest level of performance and stability of performance. The bad news is that in the infrequent situations where tuning of SQL statements is required, you're going to need a knowledge of statistics and access paths, all of which is becoming significantly more complicated. Nonetheless, an understanding of catalog statistics and the major access paths is a requirement for SQL performance tuning. The DB2 Administration Guide, DB2 for z/OS Managing Performance Guide, and Utilities Guide are all required reading to gain this knowledge. This section of the book will attempt to consolidate this information, as well as offer some advice on when various statistics and access paths are important for your applications.

The Critical Importance of Appropriate Catalog Statistics

The DB2 engine is cost based and access path selection is based upon statistics that are stored in the DB2 system catalog. Accurate statistics are a necessity if you want to get the best level of performance for your SQL statements. There are many different types and levels of statistics that can be gathered, and there are trade-offs to consider. The more statistics you gather, the more information will become available to the optimizer for calculating access paths. However, not all of these statistics are necessary for every type of query and data. So, you shouldn't just gather all available statistics for your tables and indexes since the quantity of statistics is going to influence the cost of binding or prepare SQL statements. You are going to need an understanding of any special data or application requirements before gathering anything beyond the basic statistics.

I strongly recommend that when you do collect statistics you do so for the complete set of objects related to a table space. That is, you should not collect index statistics separate from table, column, and table space statistics. This can lead to conflicts, which can distort optimizer calculations and impact access path selection. I almost never gather statistics as part of the execution of other utilities, and never gather index statistics separate from other statistics.

Base Line Statistics

The RUNSTATS utility is used to gather statistics based upon the data stored in your tables as well as the key values and entries in the indexes referencing those tables. The RUNSTATS utility will collect statistics based upon parameters set for table space, tables, columns, and indexes. The following is an example of an extremely common RUNSTATS utility control statement that gathers what I consider to be the minimal base line statistics for a table space, associated table, and associated indexes.

```
RUNSTATS TABLE SPACE DB1.TS1 TABLE(ALL) INDEX(ALL)
```

These base line statistics are important for transaction processing systems where the vast majority of the SQL statements are static embedded statements that use host variables in predicates or dynamic SQL statements where the vast majority of the SQL statements use parameter markers in predicates. Also, if the majority of statements in your application are very simple, single table accesses then there probably is not much need for any type of statistics beyond this.

I am typically only utilizing base line statistics until I have a special situation that may require additional statistics.

Frequency Distribution Statistics

Some performance problems can be attributed to skewed distribution of data. The optimizer assumes that the data is uniformly distributed. When this is not the case, the result can be poor join sequences, increased synchronous I/O, and longer application response times. These outcomes can occur if the optimizer chooses an incorrect access path because it has incorrectly estimated the number of matching rows for a qualified predicate. For a predicate such as this:

```
WHERE LASTNAME = 'RADY'
```

DB2, without frequency distribution statistics, will use a formula of $1/\text{COLCARD}$ (1 divided by the column cardinality) in order to determine the percentage of time “RADY” would be found in the table. If the cardinality was 1,000, then the percentage of qualifying rows would be 0.1%. If the values of the LASTNAME column were evenly distributed then this would be an accurate estimate. However, let’s assume that we have a high degree of nepotism at our company and we have about 200 Rady’s working for us. Then, the actual percentage of qualifying rows would be 20%. In this case, the default filtering calculation will be incorrect when compared to the actual data values. Since DB2 uses the filter factors it calculates to determine whether or not to use an index, as well as the table access sequence for multi-table statements, such a difference in filtering estimates can result in a less than efficient access path for the specific query and value. In a situation such as this, frequency distribution statistics should be considered for the table.

The RUNSTATS utility can collect any number of most and least frequently occurring values for a column or set of columns along with the frequency of occurrence for those values. Please keep in mind that the frequency distribution statistics are only meaningful, for the most part, if your queries are using embedded literals or runtime re-optimization.

If you suspect that you have a situation where you might benefit from frequency distribution statistics, you can run some queries against the table with the suspected skewed data to make sure before you collect the statistics. In the case of our LASTNAME column, such a query would look like this:

```
SELECT LASTNAME, COUNT(*)  
FROM EMP  
GROUP BY LASTNAME
```

If the column counts in the result set show a significant level of variation, and your SQL statements against this table will qualify in that predicates with literal values are coded against the column, then you could benefit from gathering frequency distribution statistics.

Column Correlation Statistics

DB2 calculates a filter factor based upon formulas it uses to calculate the percentage of rows that will qualify for a given predicate. When there are multiple predicates coded against a table, DB2 will determine column level filter factors for each column. It will then use filter factors coded for multiple predicates against a given table to calculate index and table level filter factors. These will ultimately

contribute to decisions as to whether or not to use a certain index and/or the table access sequence for a given SQL statement. Filter factors in these situations can get distorted if the columns being referenced in a compound predicate are highly correlated. To understand column correlation think of data values as they change in a table. If the value for one column changes, what is the likelihood that another column will change? If this is the case then those columns may be correlated and therefore predicates coded against both columns may inaccurately influence the filter factor. A classic example of correlated columns is city and state. If you look at the United States the chances are that if the value for the city column is “Chicago” there’s a very good chance that the value for state will be “Illinois”, and if the value for the city column is “Baltimore” then there’s a very good chance that the value for state will be “Maryland”. Assuming the following predicate:

```
WHERE CITY = ? AND STATE = ?
```

DB2 is going to calculate multiple filter factors. For the predicate against city, assuming 10,000 different cities in our table, will be $1/10000$ or 0.0001 , and for the predicate against the state column, the filter factor will be $1/50$ or 0.02 . DB2 is going to then calculate the combined filter factor for the compound predicate $0.0001 * 0.02 = 0.000002$. If the columns are correlated, then the filter factor is exaggerated by the percentage of the lesser occurring value, becoming in this case 2% of what it should be. This can have a significant impact when it comes to index selection and table access sequence, especially in more complex queries.

The RUNSTATS utility can collect statistics for combinations of columns. So, the same statistics that are available for single columns are also available for combinations of columns. When these statistics are available, the DB2 optimizer will consider the correlation statistics versus the single column statistics for compound predicates. This will result in a more accurate filter factor for these compound predicates and thus result in more accurate index selection and table access sequence.

Before you collect column correlation statistics you need to make sure that they’ll actually benefit your applications. There are two things that need to be done to confirm this need. First, you need to determine if there are compound predicates coded against columns that can potentially be correlated. Second, you need to determine if the columns in question are actually correlated. To accomplish the first will be a matter of scanning the SQL statements in the application, and to accomplish the second you can run some queries against the suspected tables and columns. In our example of the table with the CITY and STATE columns your analysis query would look something like this:

```
SELECT VALUE1, VALUE2,
       CASE WHEN VALUE1 > VALUE2 THEN 'COLUMNS CORRELATED'
       ELSE 'COLUMNS NOT CORRELATED' END
FROM (
  SELECT COUNT (DISTINCT CITY) * COUNT (DISTINCT STATE)
        , (SELECT COUNT(*)
           FROM (SELECT DISTINCT CITY, STATE
                 FROM TAB1) AS T2)
  FROM TAB1) AS CORRELATION_COMPARE(VALUE1, VALUE2)
```

The query is comparing a multiplication of the counts of distinct values to the distinct combination of the two values. I truly don't believe that it's absolutely black and white, and so if the two values are close to equal then they are not correlated. However, if the first value is significantly greater than the second value then it is likely the two columns are correlated. Also, if the table is relatively small then it's more difficult to determine column correlation. The following is an example of a RUNSTATS utility to collect correlation statistics if the query above indicated that the correlation statistics were required.

```
RUNSTATS TABLESPACE DB1.TS1  
TABLE (DANL.TAB1)  
COLGROUP (CITY, STATE)
```

Histogram Statistics

Histogram statistics enable DB2 to improve access path selection by estimating predicate selectivity from value-distribution statistics that are collected over the entire range of values in a data set. This information aids filtering estimation when certain data ranges are heavily populated and others are sparsely populated.

DB2 chooses the best access path for a query based on predicate selectivity estimation, which in turn relies heavily on data distribution statistics. Histogram statistics summarize data distribution on an interval scale by dividing the entire range of possible values within a data set into a number of intervals.

DB2 creates equal-depth histogram statistics, meaning that it divides the whole range of values into intervals that each contain about the same percentage of the total number of rows. The following columns in a histogram statistics table define an interval:

- QUANTILENO. An ordinary sequence number that identifies the interval.
- HIGHVALUE. A value that serves as the upper bound for the interval.
- LOWVALUE. A value that serves as the lower bound for the interval.

Histogram statistics are most useful if your queries against the skewed data include range, LIKE, and BETWEEN predicates. Histogram statistics could also be useful for equality predicates when a portion of the input values are expected not to match the data. As with frequency distribution statistics, the histogram statistics are most useful when your predicates contain literal values or you are using runtime reoptimization.

Sampling

Executing a RUNSTATS utility can be an expensive endeavor, especially for extremely large tables. There is an option to use sampling when running a RUNSTATS utility. The sampling isn't necessarily going to reduce the elapsed time of the utility, but will reduce the CPU time consumed, which is important if you are paying per CPU hour. I have found that the statistics produced via sampling are fairly accurate when you are sampling between 10 and 100 percent of the rows in the table. Do not specify less than 10 percent for a sample as experience has shown that the statistics produced can be highly inaccurate. There are two choices for sampling : row level and page level. Although I have never tried page level sampling, it may be a promising alternative to row level from an elapsed time perspective.

When to Gather Statistics

There are three basic thoughts when it comes to gathering statistics.

- Gather statistics on a regular basis. This is quite a common practice, and it could be a good thing or a bad thing. In my opinion this is a bit reckless. Since statistics can cause access path changes, the regular refreshing of statistics creates a certain level of unpredictability to access path changes. Some shops install a regular policy of the “Three R’s”, REORG, RUNSTATS, and REBIND. This is only a good practice for enterprises where DBA support is limited and the DBAs do not have a good relationship with development and know nothing about the applications that are using the data. In this case you are putting 100% of your faith in the DB2 engine; do not expect to do proactive SQL analysis and tuning and have a tuning strategy that involves simply fighting fires. If this is not your situation and it is your policy then I strongly recommend that you explore one of the other choices for gathering statistics. This may very well be a better policy for data warehouses.
- Gather statistics once and forget about it. This technique is most appropriate when the database and application is quite well known by the DBAs and plenty of proactive SQL performance analysis has taken place and is understood. This is also a good policy for transaction processing systems with extremely high data volume and extremely high transaction rates. In these situations you should have done your homework, and the access path should be understood and predictable. In these situations there is no need to gather statistics beyond what has been gathered to get the desired access paths. Understand that in this situation you are trading the CPU costs for running regular RUNSTATS and the unpredictability of production access path selection with the cost of DBA time and effort to fully understand the database and application. It does require that the application developers and DBAs have a strong relationship and that as changes are made to the application, database or data, the developers and DBAs are communicating, testing, and documenting the impact. RUNSTATS will have to be executed when changes are implemented. This is the technique I use in the vast majority of high volume, high transaction rate OLTP implementations.
- Gather statistics when something changes. This technique is pretty much a balance between the first two techniques. It requires an initial gathering of representative statistics and then a policy in place whereas if data, database, or processes change, the DBA is informed and a RUNSTATS is executed. This is a very common recommendation and a smarter thing to do than the option of running regular RUNSTATS, but only when you have the DBA resources available to deal with the coordination of change.

In addition to gathering statistics, the RUNSTATS utility can also reset the statistics. This is important when there is significant change to database, data, or processes, or if you have previously deployed a flavor of statistics that is no longer desired. This is also great for test environments where you may want to try an assortment of different statistics when testing an application for performance. See the DB2 11 Utility Guide for details.

Real-time Statistics and Object Reorganization

Catalog statistics are critical to performance and reflect the condition of the data at the moment they are gathered. The REORG utility is useful for organizing the data in tables and the keys in an index, and hopefully the organization of the data in your tables and the catalog statistics about those tables are in harmony. It is really important to gather catalog statistics when objects are properly organized in situations where you have a good policy of keeping objects properly organized and are aware of expected application performance. If no good policy exists, then the “Three R’s” might just be the way to go.

DB2 gathers and reports on object organization and activity in the real-time statistics catalog tables, SYSIBM.SYSTABLESPACESTATS and SYSIBM.SYSINDEXSPACESTATS. The gathering of these statistics is automatic and the population of the data in the tables controlled by the STATSINT subsystem parameter. You can also force the externalization of these statistics for an object with the following command:

```
ACCESS DB(<database name>) SP(<pageset name>) MODE(STATS)
```

For high volume transaction processing systems, I typically do the “one and done” approach to collecting catalog statistics, and then employ an intelligent policy of monitoring my production objects for organization, executing REORGs on only the objects that require reorganization⁶. I utilize the real-time statistics tables to tell me when objects need to be reorganized, and only reorganize those objects when needed. The chances are that if you do this type of analysis you’ll be really surprised at the reduction in REORGs from a regular REORG policy. It is quite likely that real-time statistics will expose the need for index REORGs far more often than table space REORGs.

IBM supplies a stored procedure called DSNACCOX⁷ that can interpret the data in the real-time statistics. Since the installation and use of this stored procedure is optional, it may not be available at your installation. For this reason I have developed a set of queries against the real-time statistics tables that I execute on a regular basis. These queries tell me which tables and indexes to reorganize. Below are examples of basic rules that run against the statistics gathered since your last REORG. These are not dissimilar from the default rules, but these rules can easily be adjusted to your preferences.

Table space:

- Indirect references greater than 5%
- Inserts greater than 25%
- Deletes greater than 25%
- Unclustered inserts greater than 10% (be mindful of intentional unclustered inserts) The SQL seems to check for 5% and not 10% ?????
- Disorganized LOBs greater than 50%
- Mass deletes
- More than 30 extents
- Table is unclustered or has poor clustering

⁶ CA Database Analyzer® for DB2 for z/OS

⁷ CA Database Analyzer® provides a superset of the THRESHOLDS embedded in DSNACCOX.

```

SELECT TBSP.DBNAME,
       TBSP.NAME,
       TB.NAME,
       TBSP.PARTITION,
       TBSP.NACTIVE,
       TBSP.NPAGES,
       TBSP.TOTALROWS,
       CASE WHEN ((TBSP.REORGNEARINDREF+TBSP
REORGFARINDREF)*100)/TBSP.TOTALROWS)>5
           THEN ((TBSP.REORGNEARINDREF+TBSP
REORGFARINDREF)*100)/TBSP.TOTALROWS)
           ELSE 0 END AS INDREF,
       CASE WHEN ((TBSP.REORGINSERTS*100)/TBSP.TOTALROWS)>25
           THEN ((TBSP.REORGINSERTS*100)/TBSP.TOTALROWS)
           ELSE 0 END AS INSERTS,
       CASE WHEN ((TBSP.REORGDELETES*100)/TBSP.TOTALROWS)>25
           THEN ((TBSP.REORGDELETES*100)/TBSP.TOTALROWS)
           ELSE 0 END AS DELETES,
       CASE WHEN ((TBSP.REORGUNCLUSTINS*100)/TBSP
TOTALROWS)>10
           THEN ((TBSP.REORGUNCLUSTINS*100)/TBSP.TOTALROWS)
           ELSE 0 END AS UNCLINS,
       CASE WHEN ((TBSP.REORGDISORGLOB*100)/TBSP.TOTALROWS)>50
           THEN ((TBSP.REORGDISORGLOB*100)/TBSP.TOTALROWS)
           ELSE 0 END AS DLOB,
       CASE WHEN TBSP.REORGMASDELETE>0
           THEN TBSP.REORGMASDELETE
           ELSE 0 END AS MDEL,
       CASE WHEN TBSP.EXTENTS>30
           THEN TBSP.EXTENTS
           ELSE 0 END AS EXT,
       CASE WHEN IX.CLUSTERED = 'N'
           THEN IX.CLUSTERED
           ELSE ' ' END AS CLUS,
       CASE WHEN (IX.CLUSTERRATIOF*100) < 95
           THEN INT((IX.CLUSTERRATIOF*100))

```

```

        ELSE 0 END AS CLUSR,
        TBSP.EXTENTS,
        TBSP.LOADRLASTTIME,
        TBSP.REORGLASTTIME,
        TBSP.REORGINSERTS,
        TBSP.REORGDELETES,
        TBSP.REORGUPDATES,
        TBSP.REORGUNCLUSTINS,
        TBSP.REORGDISORGLOB,
        TBSP.REORGMASSDELETE,
        TBSP.REORGNEARINDREF,
        TBSP.REORGFARINDREF,
        TBSP.INSTANCE,
        TBSP.SPACE,
        TBSP.DATASIZE
FROM    SYSIBM.SYSTABLESPACESTATS TBSP
INNER JOIN
        SYSIBM.SYSTABLES TB
ON      TBSP.DBNAME = TB.DBNAME
AND     TBSP.NAME = TB.TSNAME
LEFT OUTER JOIN
        SYSIBM.SYSINDEXES IX
ON      TB.CREATOR = IX.TBCREATOR
AND     TB.NAME = IX.TBNAME
AND     IX.CLUSTERING = 'Y'
WHERE   TBSP.DBNAME IN
        ('<dbname>', '<dbname>')
AND     TB.CREATOR = '<creator>'
AND     TBSP.NACTIVE > 100
AND     TBSP.TOTALROWS > 0
AND     (
        ( (( (TBSP.REORGNEARINDREF+TBSP.REORGFARINDREF)*100)/
TBSP.TOTALROWS)>5)
        OR
        ( (( (TBSP.REORGINSERTS*100)/TBSP.TOTALROWS)>25)

```

```
OR
  (( (TBSP.REORGDELETES*100)/TBSP.TOTALROWS)>25)
OR
  (( (TBSP.REORGUNCLUSTINS*100)/TBSP.TOTALROWS)>10)
OR
  (( (TBSP.REORGDISORGLOB*100)/TBSP.TOTALROWS)>50)
OR
  (TBSP.REORGMASDELETE>0)
OR
  (TBSP.EXTENTS>30)
OR
  (IX.CLUSTERED = 'N')
OR
  ((IX.CLUSTERRATIOF*100) < 95)
)
ORDER BY TBSP.DBNAME,
        TBSP.NAME,
        TBSP.PARTITION
WITH UR
```

Index:

- Far away leaf pages greater than 10%
- Extents greater than 30%
- Inserts plus deletes greater than 20%
- Pseudodeletes greater than 10%
- Appended inserts greater than 10% (be aware of potential intended appended inserts)
- Any mass deletes
- Any change in number of levels

```

SELECT SUBSTR(INDX.TBCREATOR,1,10) AS TBCREATOR,
       SUBSTR(INDX.TBNAME,1,20) AS TBNAME,
       SUBSTR(RTSI.CREATOR,1,10) AS CREATOR,
       SUBSTR(RTSI.NAME,1,20) AS NAME,
       RTSI.PARTITION,
       SUBSTR(RTSI.DBNAME,1,10) AS
DBNAME,NACTIVE,TOTALENTRIES,
       CASE WHEN (((1+RTSI.REORGLAFFAR)*100)-99)/RTSI
NACTIVE > 10)
           THEN (((1+RTSI.REORGLAFFAR)*100)-99)/RTSI
NACTIVE)
           ELSE 0 END AS LAFFAR,
       CASE WHEN (RTSI.EXTENTS > 30)
           THEN (RTSI.EXTENTS)
           ELSE 0 END AS EXTENTS,
       CASE WHEN (((1+RTSI.REORGINSERTS+RTSI
REORGDELETES)*100)-99)/RTSI.TOTALENTRIES > 20)
           THEN (((1+RTSI.REORGINSERTS+RTSI
REORGDELETES)*100)-99)/RTSI.TOTALENTRIES)
           ELSE 0 END AS INS_DEL,
       CASE WHEN (((1+RTSI.REORGPSEUDODELETES)*100)-99)/RTSI
TOTALENTRIES > 10)
           THEN (((1+RTSI.REORGPSEUDODELETES)*100)-99)/RTSI
TOTALENTRIES)
           ELSE 0 END AS PDEL,
       CASE WHEN (((1+RTSI.REORGAPPENDINSERT)*100)-99)/RTSI
TOTALENTRIES > 10)
           THEN (((1+RTSI.REORGAPPENDINSERT)*100)-99)/RTSI
TOTALENTRIES)
           ELSE 0 END AS AINS,
       CASE WHEN (RTSI.REORGMASDELETE > 0)
           THEN (RTSI.REORGMASDELETE)
           ELSE 0 END AS MDEL,
       CASE WHEN (RTSI.REORGNUMLEVELS <> 0)
           THEN (RTSI.REORGNUMLEVELS)
           ELSE 0 END AS LEVELS,
       RTSI.REBUILDLASTTIME,
       RTSI.REORGLASTTIME,
       RTSI.REORGINSERTS,
       RTSI.REORGDELETES,
       RTSI.REORGAPPENDINSERT,

```

```

RTSI.REORGPSEUDODELETES,
RTSI.REORGMASDELETE,
RTSI.REORGLAFNEAR,
RTSI.REORGLAFFAR,
RTSI.REORGNUMLEVELS
FROM   SYSIBM.SYSINDEXSPACESTATS RTSI
INNER JOIN
        SYSIBM.SYSINDEXES INDX
ON      RTSI.CREATOR = INDX.CREATOR
AND     RTSI.NAME = INDX.NAME
WHERE   RTSI.CREATOR = '<creator>'
AND     RTSI.NACTIVE > 10
AND     RTSI.TOTALENTRIES > 0
AND     (
        (((1+RTSI.REORGLAFFAR)*100)-99)/RTSI.NACTIVE > 10)
OR
        (RTSI.EXTENTS > 30)
OR
        (((1+RTSI.REORGININSERTS+RTSI.REORGDELETES)*100)-99)
RTSI.TOTALENTRIES > 20)
OR
        (((1+RTSI.REORGPSEUDODELETES)*100)-99)/RTSI
TOTALENTRIES > 10)
OR
        (((1+RTSI.REORGAPPENDINSERT)*100)-99)/RTSI
TOTALENTRIES > 10)
OR
        (RTSI.REORGMASDELETE > 0)
OR
        (RTSI.REORGNUMLEVELS <> 0)
)
ORDER BY INDX.TBCREATOR,
        INDX.TBNAME,
        RTSI.PARTITION
WITH UR

```

Please be aware of the fact that the output from these queries, as well as the stored procedure, is dependent upon populated and updated statistics in the RTS tables. In certain situations these statistics may not be populated. It is important to check the real-time statistics for the objects of interest to make sure they are being populated before running these queries. Otherwise you may miss important REORGs. The index query does not take into account indexes that will be REORGed as part of a tablespace REORG resulting from the table space query. You can easily add an exclusion to the index query for tables not in the table space query.

DB2 Can Report on Missing Statistics

As stated earlier, it is extremely important to collect accurate statistics on your object, and collect consistent statistics as well. If there are holes in statistics, DB2 will collect and report on the missing or conflicting statistics as part of the statement compile process (PREPARE or BIND) or as part of the EXPLAIN process. This missing information is documented in the SYSIBM.SYSSTATFEEDBACK system catalog table. The population of this table is controlled by the STATFDBK_SCOPE subsystem parameter.

There are basically two things you can do if you discover data in the SYSIBM.SYSSTATFEEDBACK: run a RUNSTATS utility to report on the statistics and/or update the statistics; or use a tool that will analyze the information in SYSIBM.SYSSTATFEEDBACK and make recommendations.

Missing statistics will also be reported during EXPLAIN or BIND/REBIND/PREPARE.

For more information about this, please see Section 4 of this book.

Invalidating the Dynamic Statement Cache

Executing RUNSTATS will invalidate statements in the dynamic statement cache, which will then go through a long prepare and statement compilation at next execution. There may be situations in which it is desirable to refresh the monitoring statistics captured in the dynamic statement cache (see Section 8 for details about these statistics) without actually updating the system catalog statistics for an object. You can do this with the following format of the RUNSTATS utility.

```
RUNSTATS TABLESPACE <database name>.<table space name>  
REPORT NO UPDATE NONE
```

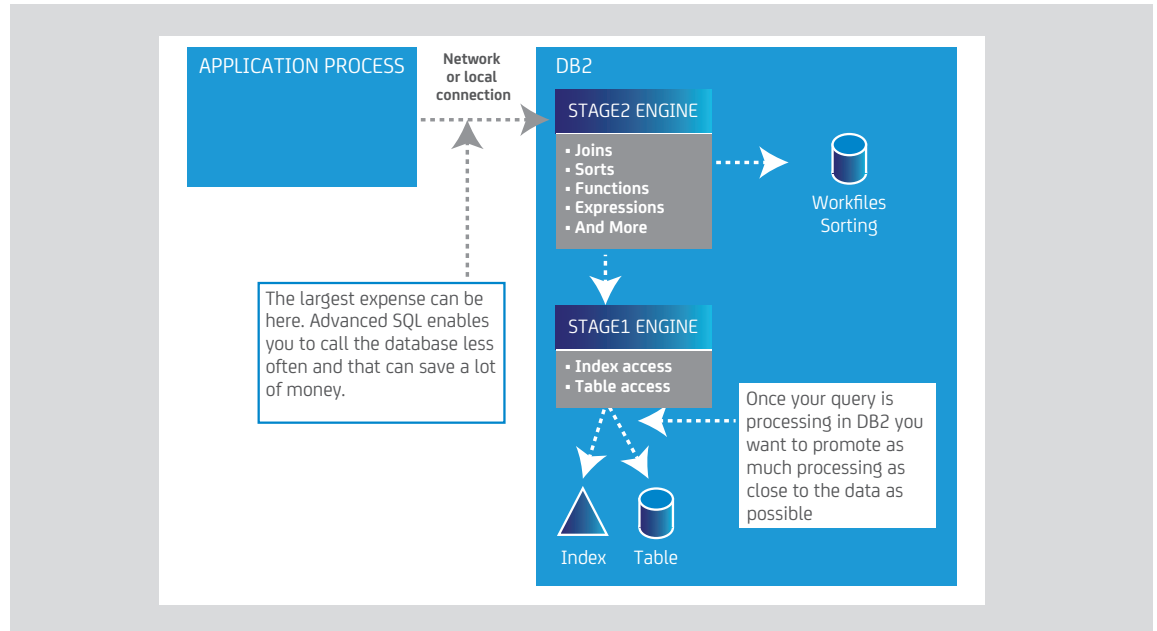
DB2 Engine Overview

I like to think of the DB2 for z/OS engine in relationship to SQL processing divided into three parts: Stage 1, Stage 2, and Stage 3. This simplistic approach is enough to give you the correct frame of mind to construct and tune SQL statements for performance, and help you understand the golden rule of SQL performance: “Filter as much as possible as early as possible”.

- Stage 1. This is the part of the DB2 engine that knows about tables and indexes. This is where you want the majority of filtering of data from tables and indexes to happen. It is the only place where the amount of data that is retrieved from mass storage (i.e. disk) can be limited.
- Stage 2: This is the part of the DB2 engine that processes, sorts, functions, expressions, and joins. It does not know about tables and indexes, so tables and indexes must be accessed and the data passed from Stage 1 to Stage 2 before any Stage 2 filters can be applied. Filtering performed in Stage 2 is not as efficient as filtering that is performed in Stage 1. However, it is significantly better than any filtering that is done in Stage 3.
- Stage 3: This is the fictitious step that has nothing to do with DB2 itself and everything to do with proper application development. It is filtering within the application process. To be more specific, it is when an application pulls data from the database only to discard that data upon further processing. Stage 3 predicates are wasteful and unnecessary 100% of the time. Any time a Stage 3 process is coded it would always be a better decision to put that filtering into the database!

The following diagram illustrates the DB2 engine and the three stages.

Figure A.



Predicate Types

The types of predicates and predicate processing are clearly documented in the DB2 for z/OS Managing Performance Guide. It is highly recommended that you read that section of the DB2 for z/OS Managing Performance Guide right now.

From a performance perspective your goal is to code Stage 1 indexable predicates, which are also known as index matching predicates. These are predicates that can match leading columns of indexes. It's important to look at a collection of predicates in your queries when determining how to construct indexes to support them. Predicates of the form `col=value` are the most likely to be indexable; `IN` predicates are likely to be indexable as well; `range` and `BETWEEN` predicates are indexable; but any predicates that can potentially match on columns after the range predicate is applied cannot be indexable (except in situations in which a query may get multiple index access).

Boolean term predicates are also important for index access. A Boolean term predicate is a simple or compound predicate, that when evaluated false for a given row makes the entire `WHERE` clause false for that row. You should code Boolean term predicates whenever possible to encourage index access. Examples of Boolean term predicates can be found in this section in the section on range-list access.

DB2 has the ability to transform predicates coded as Stage 2 into Stage 1 predicates, and is also capable of calling the Stage 2 engine while accessing indexes and tables in Stage 1. So, which predicates you should focus on is not always a clear cut. DB2 can also transform non-Boolean term predicates and has special access paths that can improve the indexability of non-Boolean term predicates. This is why the use of the `EXPLAIN` facility is critical in understanding how a predicate is interpreted, transformed and processed by DB2. Some of this is explained in the next section on query transformation.

Query Transformation and Access Path Selection

As best as I know, these are the major components to the DB2 engine as divided by roles within the DB2 development team.

- QST (query transformation)/parser
- APS (Access Path Selection – internally referred to as optimizer)
- Post-optimizer/parallelism
- ASLGEN
- Structure Gen
- Runtime

When most people talk about the processing of DB2 SQL statements, they speak of the DB2 optimizer. What many don't realize is that before the query passes through the optimizer it first passes through something called "query transformation". This component of the DB2 engine is critical to performance, and a vital part of query processing. It is important to understand query transformation because it is a separate piece of code from the optimizer, owned by a separate development team within DB2 development, and the first component to manage your queries as you send them in to be processed.

Query transformation is responsible for this type of processing, amongst others:

- Parse the query
- Look up DB2 objects
- Read statistics
- Rewrite the query for submission to optimizer
- Transform queries to special DB2 objects
 - Materialized query table
 - Temporal table
 - View

In order to be a truly effective SQL performance specialist you need to understand the role of query transformation in SQL processing. Some of the things that query transformation can do when processing a SQL query include:

- Transformation of subqueries. Table statistics are used to influence subquery transformation into either a subquery of a different type or a join. The subquery and join section of Section 3 of this book describes the guidelines from which to code, and the transformations will mostly follow these guidelines:
 - Correlated and non-correlated into joins
 - Correlated to non-correlated
 - Non-correlated to correlated

- Merging of table expressions with the outer query that references them. Read more about this in the nested table expression section of this section.
- Conversion of IN-list into a join. This utilizes the notion of relation division as a means of increasing the potential of matching index access. It also can eliminate the scanning of last IN-lists. See the comments on relation division in Section 3 of this book in the sections on joins, nested table expressions, and common table expressions.
- Rewriting of ORDER OF clause. ORDER OF clauses are simply rewritten into ORDER BY clauses based upon the expression that they reference. There's no magic here or increased potential for sort avoidance, just a simple cut and paste.
- Predicate transitive closure. If $a=b$ and $b=c$, then $a=c$. Query transformation analyzes predicates in ON and WHERE clauses in statements that contain joins. If filtering is applied to a column that is used in a join predicate, then query transformation may generate an additional predicate to add that filtering to the other table of the join. This gives the optimizer (aka access path selection) more opportunities for considering table access sequence with the goal of accessing the table that returns the fewest rows first.
- CASTING of data types. If there are mismatched data types in filtering or join predicates then query transformation will introduce CAST expressions to cast one side of the predicate to the other.
- Predicate and join distribution across a UNION or UNION ALL. Query transformation distributes predicates coded against a nested table expression or view that contains a UNION or UNION ALL across all blocks of the UNION or UNION ALL. It also distributes joins to these types of table expressions. More on this in the sections of this section on UNION and nested table expressions.
- Temporal query predicate generation and generation of UNION ALL for system-period temporal tables and archive tables. Time travel queries using a period-specification in a FROM clause will result in the generation of range predicates in the WHERE clause to do the time travel. For system-period temporal tables and archive tables query transformation will generate UNION ALL in a nested table expression to enable access to history and archive tables. The rules and performance implications for these table expressions are the same as for normal table expressions that contain UNION ALL. Please see the sections in this section on UNION and nested table expressions and the section on temporal tables and archive tables in Section 6 of this book for more details.
- Transformation of some non-Boolean term predicates to Boolean term. There is nothing but goodness to this feature as it increases the index-ability of non-Boolean term predicates.
- Predicate pushdown into nested table expressions and views. Again, nothing but goodness to this feature.
- Predicate pruning for UNION in view or table expression and predicate pruning of some always true and always false predicates. When predicates are coded against a nested table expression that contains a UNION or UNION ALL those predicates can be distributed across all blocks of the UNION or UNION ALL. Those predicates within each block of the union are then interrogated and compared to already existing predicates in the block and if two predicates result in an "always false" condition then that entire block of the union will be eliminated from the query.

- ORDER BY pruning. Query transformation will compare ORDER BY columns with columns that filter data in the WHERE clause predicate. If there is an equal's predicate against a leading ORDER BY column then that column will be eliminated from the ORDER BY such that there is a greater chance of sort avoidance.
- Join simplification. If a local predicate is coded against the null supplying table of an outer join and that local predicate does not consider null values then the outer join is transformed into an inner join due to the fact that null values from the null supplying table would be eliminated anyway. By transforming the outer join to an inner join access path selection then has more options for table access sequence and higher performance. Of course this also presents a challenge for developers because in many cases they have incorrectly coded this type of predicate and have inadvertently coded an inner join versus an outer join. Understanding this aspect of query transformation and being able to view the transformed query (Section 4 of this book) is a key component to understanding these types of problems.

With all of this potential for the query submitted to DB2 to be transformed to something that is logically equivalent, but significantly different physically, it is absolutely critical to have an understanding as to the effects of query transformation. Researching query transformation is a difficult proposition. Reading this section of this book a few times, and then looking up each of the features presented here in the DB2 for z/OS Managing Performance Guide, is a good start. The next best thing that you can do to understand query transformation is to write some test queries and use the EXPLAIN facility to expose the transformed queries. Section 4 of this book details how transformed queries are exposed in the EXPLAIN tables.

Single Table Access Paths

All table access paths are completely documented in the DB2 for z/OS Managing Performance Guide. In this book I simply reference the different types of access and discuss when this type of access is appropriate. Not all access paths are covered here as it is intended to be a compliment to the DB2 for z/OS Managing Performance Guide.

Matching Index Access

Matching index access is appropriate for all sorts of queries even when there are relatively small amounts of data in a table. Matching index access is great for OLTP queries and even certain warehouse type queries. It may be less appropriate for queries that process large volumes of data sequentially, but the optimizer would be less likely to choose an index in these situations unless it was to avoid a sort.

Index-only access is possible in situations in which an index supplies all of the columns required for the coded predicates and for the returned columns in the SELECT clause. It may be beneficial in some situations to add columns to an index to make the access index-only. DB2 allows for INCLUDE columns for unique indexes just for this purpose. Index-only access is a performance advantage in situations in which you cannot establish a common cluster between joined tables, and generally in situations in which you need extreme read access based upon a non-clustering key. Be cautious, however, when making indexes with lots of columns as this can seriously impact the performance of inserts and deletes against the index.

In order to match on index columns of a single index you must code Stage 1 indexable Boolean term predicates.

DB2 is also capable of what is known as multi-index access. This type of access path is utilized when there is a compound predicate coded against a table connected by ANDs or ORs. Multi-index access can use different indexes or the same index multiple times. Non-Boolean term predicates can qualify for Stage 1 multi-index access. Some non-Boolean term predicates can also qualify for range-list index access.

Index matching access is generally a very good access path for transactions, as well as for report writing queries. It's especially good for transactions that return little or no data; as filtering is at the Stage 1 index matching level and undesired data is eliminated with the least amount of data access. People generally strive towards index matching access, but are confused sometimes when they are expecting index matching and don't get it. The optimizer may consider alternate access paths when the amount of data qualifying, based upon statistics, is large and/or it is trying to avoid a sort. Clustering and having well-clustered (organized) data can also play a role in which index DB2 selects for a query. So, don't be alarmed when the level of index matching, or a good index matching index was not chosen. Instead, focus on figuring out why. Some of the reasons can include: another index eliminated a sort, a different table access sequence in a multi-table query eliminated more rows first, or too many rows qualified even when matching columns.

Range-List Index Access

Range-list index access was designed to support "pagination" type queries that support result sets that can span more than one screen. These queries typically employ a specific style of non-Boolean term predicates in order to position a cursor on the next or previous page. The compound predicate form may look something like this:

```
WHERE LASTNAME > 'RADY'
OR      (LASTNAME = 'RADY' AND FIRSTNAME > 'BOB')
```

In the past the recommended coding technique was to rewrite the predicate as non-Boolean term by adding a redundant predicate:

```
WHERE LASTNAME >= 'RADY'
AND      (LASTNAME > 'RADY'
OR      (LASTNAME = 'RADY' AND FIRSTNAME > 'BOB'))
```

This technique is no longer needed (as of DB2 10 for z/OS) as it should be accommodated by range-list index access. However, if you expect range-list but are not getting it then your predicates may not be indexable. In that case, you still may want to add the redundant predicates if index access is desired for the predicates coded.

IN-List Access

IN-list access is a variation of matching index access and comes in two flavors; in-list index scan and in-list direct table access. These can support single and multiple in-list predicates, as well as predicate combinations involving in-lists and additional range or equality predicates. The important thing to remember here is not to fear a query with an in-list and to consider the indexability and

filtering potential of in-list predicates. When doing an EXPLAIN against a query with multiple in-lists you should keep in mind that if in-list direct table access is selected you will see a nested loop join from the in-memory tables built from the in-lists to the table the predicates are coded against. Also, keep in mind that if you code multiple predicates against a table and use an in-list against one or more leading index columns, the remainder of the predicates may still be indexable. This can also be true for IN subqueries.

Non-Matching Index Access and Index Screening

It is important to remember that even though an EXPLAIN might show an index being utilized, you need to pay careful attention to the purpose of the index. This requires you to pay attention to the number of columns being matched to the index. If the number of columns matched is zero, then the access path is a non-matching index access. DB2 will choose this access path in situations in which predicate filtering is poor, or if the use of the index is beneficial to avoiding a sort and the materialization of the result set. DB2 may also choose a non-matching index access because index screening may provide some filtering.

Index screening is filtering that DB2 can apply at the index level when matching is not possible on the leading columns of the index or when there is limited matching of the leading index columns. For example, if one predicate matches on the first column of the index and another predicate matches on the third column of the index but there is no matching predicate on the second column of the index, then DB2 can screen on the third column of the index. What this means is that even though the screen predicate does not limit the range of keys retrieved from the index, it can eliminate qualifying rows before data retrieval. This elimination will result in reduced getpage count to the table.

Now, how do you know that index screening is actually happening? The output of EXPLAIN can indicate that a predicate qualifies as an index screening predicate. However, that is not the ultimate way of determining whether or not index screening is being applied at execution. You can assume, based upon the EXPLAIN output, that the index screening is happening. However, EXPLAIN is not always correct in this regard. If the getpage count for table access appears higher than expected for a query in which you expect index screening, then you may want to try running the query in a controlled test without the index screening predicate to see if the getpage count varies. If it doesn't then you could inquire with IBM why EXPLAIN says to expect index screening but it really isn't happening, or you can attempt to modify your query and/or indexes to promote a high level of matching index access.

Index screening is possible for many of the single and multi-table access paths, whether or not there is matching index access.

List Prefetch

As mentioned earlier, matching index access is great for queries that support transactions that return little or no data. When the matching predicates are predicted to return somewhat more than a relatively few rows, and the index being utilized is not the clustering index or is clustering but the table data is disorganized, then DB2 may elect to use the list prefetch access method. In other words, list prefetch is a desired access path when accessing a moderate amount of data that is not sequentially organized in the table. In this access path, one or more columns are matching, the index keys are retrieved and sorted in record identifier (RID) sequence which contains the page number, and then the pages of the table are accessed in a sequential or skip-sequential fashion. If the majority of

queries against a table are using list prefetch in an OLTP application, you may want to consider changing the clustering of the table to the index being selected for list prefetch. Be careful to make sure you are monitoring all of the queries against the table.

Historically, some customers have been wary of list prefetch due to the potential for DB2 to exhaust RID storage for some queries that use list prefetch and revert to table space scans. This is no longer the case as the subsystem can be configured to overflow rid storage to workfiles.

Hash Access

Hash access is a very specific access path for tables that are defined as hash tables, and for predicates that are equal predicates against the primary key column of the table. By utilizing a reserved hash area that is stored within the table space, DB2 can completely avoid index access and select the desired row directly. This should save both CPU and elapsed time because index access is completely avoided. However, hash access is only beneficial when the vast majority of queries (or better yet all queries) access using an equal predicate on the primary key values only. If there is any other access against the table, you should consider not using hash table access and, perhaps, design your indexes such that you can get index-only access (perhaps via index INCLUDE columns).

A hash table is also going to use more space than a normal table, which is important to think about when considering hash access. As always, you should benchmark hash access versus a more traditional access path to see which is best for your situation.

Multi-Table Access Paths

The main focus in regards to the performance of queries against multiple tables should be the table access sequence. In general, it is desirable to eliminate the most data possible as early in the query as possible. So, the performance goal for a multi-table query is to have DB2 select the table that returns the fewest rows first.

Subquery Access

Because the DB2 engine (query transformation and access path selection) can transform subqueries, it is difficult to predict and control which table will be accessed first when you code a subquery. In short, DB2 can:

- Convert a correlated subquery into a join or non-correlated subquery
- Convert a non-correlated subquery into a join or correlated subquery

In general, the DB2 engine is attempting to either filter the most data first, take advantage of the best index matching, or make more table access sequences available (via join) within access path selection. The best you can do in these situations is to understand the capabilities within the DB2 engine when dealing with subqueries, and to code the most efficient subqueries you can (see the section in Section 3 on coding efficient subqueries and joins).

Nested Loop Join

In a nested loop join the outer table (the table accessed first) is either scanned or probed and for each qualifying row the inner table (the second table accessed) is then searched. This generally means, from a performance perspective, that the inner table can be accessed never, once, or many times during the join process. In a nested loop join it is important that the table that filters the most

and returns the fewest rows is the first table accessed. A nested loop can be an extremely efficient access path for transactions that return little or no data, and is generally my preferred access path for joins supporting transactions.

It is important to remember that when you are joining two tables together and nested loop is the selected access path, the tables being joined should share a common cluster for the best performance. In this way DB2 will avoid random access to the inner table of the join and also take advantage of such performance enhancers as sequential detection and index lookaside (discussed later in this section). It is for this reason that I usually encourage the avoidance of system-generated keys in your database design. However, if you do use system-generated keys, at least let the child tables inherit the parent's primary key in an identifying relationship. That is, the parent key is the leading column of the child's primary key. In that way, you can establish a common cluster between tables that are commonly accessed in a join. When this happens a nested loop join will be a very efficient access path.

If DB2 determines that the clustering between two tables will not support efficient access to the inner table for a non-trivial amount of data, then it may select an alternate access path. These alternates include hybrid and hash join.

Merge Scan Join

In a merge scan join, DB2 attempts to access both tables via a scan in the order of the joined columns. The inner table is always placed into a workfile. The outer table is placed into a workfile and sorted if there is no available index on the joined columns. Once in join column order, the tables are scanned and merged together. Merge scan can be an effective join method for tables in which one or both tables are not in clustering sequence relative to the join columns and both tables are going to qualify many rows of data, or when there are no indexes to support the join columns. This makes merge scan a more effective join method than nested loop for these types of queries. In general, merge scan is the report writing join method while nested loop is the transaction supporting join method. As always, it depends! I have seen merge scan outperform nested loop for transactions and nested loop outperform merge scan for reports. However, for the most part, the general rule applies.

Hybrid Join

A hybrid join can only be an inner join, and is an alternative to a nested loop join when the inner table is not in the same clustering sequence as the outer table. In a hybrid join, the outer table rows are joined to the index keys of the inner table and then sorted by inner table RID much like what happens with a list prefetch. Next, the inner table rows are accessed in a sequential or skip sequential fashion. A hybrid join, like list prefetch, is appropriate for a moderate quantity of rows when the tables are out of cluster and will outperform a nested loop join in these situations.

Hash Join, Sparse Index, and In-Memory Data Cache

There is another alternative to hybrid join and nested loop join for moderate amounts of data, typically accessed in transactions of an OLTP system. This technique is a form of hash join that enables a faster search of data to join when tables are not in common clustering sequence. While hash join is a named access path in DB2 for LUW, on DB2 for z/OS it shows up as a nested loop join

with intermediate sorts and workfiles. These in-memory files can either contain sparse indexes or hashes, with the hashes called in-memory data cache. This in-memory cache is just that - held in memory. They can overflow into actual workfiles if memory is short as controlled by the MXDTCACH subsystem parameter. Whether or not DB2 uses a sparse index or in-memory data cache, the objective is to improve the performance of the lookup into the inner table of the join.

While this hash join method exists in order to improve the response time of nested loop join in some situations, I generally look at the situation in which this join technique is being used and consider the query for possible tuning. Typically the situation occurs in a transaction processing environment, and I consider adding indexes or changing the clustering of the tables involved in order to get a more efficient access for the transactions.

UNION and UNION ALL

The important thing to remember about UNION and UNION ALL (as well as EXCEPT, EXCEPT ALL, INTERSECT, and INTERSECT ALL) is that the table access is executed in separate blocks within the query access path. DB2 does not maintain state between the separate blocks. While UNION eliminates duplicates in the result set, it does also introduce a sort. The impact of UNION in a nested table expression or view can be significant, and that is discussed in the next section on nested table expression access.

Nested Table Expression Access

When I discuss nested table expressions I'm referring to views as well. I also consider common table expressions, scalar full-selects, and table functions as forms of nested table expressions, but the majority of discussion in this section will address nested table expressions (aka inline views) and views themselves.

Merge versus Materialization

One of the most important things to understand about nested table expressions is merge versus materialization. Is the table expression merged with the outer portion of the statement that references it or is it materialized into a workfile and then read a second time by the outer portion of the statement? When processing a nested table expression or view, the DB2 engine makes a decision about what to do with the nested table expression or view. The DB2 engine will either merge the expression/view with the outer portion of the statement that references it or will materialize it in a workfile before further processing. The DB2 for z/OS Managing Performance Guide clearly documents when a nested table expression or view is materialized, however, here are some simple examples where the nested table expression or view contains a GROUP BY (most times) or a DISTINCT, or a non-deterministic function.

The following query contains a table expression that is merged with the outer statement that references it:

```
SELECT LAST_INIT
FROM (SELECT SUBSTR(LASTNAME,1,1) AS LAST_INIT
      FROM EMP) AS INIT_TBL(LAST_INIT)
```

The merged statement, seen as the transformed query in an EXPLAIN output, would look something like this:

```
SELECT SUBSTR(LASTNAME,1,1)
FROM EMP
```

The following query contains a table expression that is materialized due to the presence of a DISTINCT within the table expression:

```
SELECT LAST_INIT
FROM (SELECT DISTINCT SUBSTR(LASTNAME,1,1) AS LAST_INIT
      FROM EMP) AS INIT_TBL(LAST_INIT)
```

The following query contains a table expression that is materialized due to the presence of a non-deterministic function, the RAND() function, which is one of the most common ways to force the materialization of a table expression.

```
SELECT LAST_INIT
FROM (SELECT RAND() AS BOBRADY, SUBSTR(LASTNAME,1,1) AS LAST_INIT
      FROM EMP) AS INIT_TBL(BOBRADY, LAST_INIT)
```

What is the advantage of merge versus materialization? According to the DB2 for z/OS Managing Performance Guidemerge is always a better performance option than materialization. This is mostly true, but situations with one or more levels of nesting and embedded functions and expressions combined with multiple references to these generated columns in the outer portion of the query can result in excessive CPU consumption when compared to a materialized equivalent. In the following example the SUBSTR function is actually executed twice per row rather than once per row due to the merging of the table expression:

```
SELECT MAX(LAST_INIT), MIN(LAST_INIT)
FROM (
  SELECT SUBSTR(LASTNAME,1,1) AS LAST_INIT
  FROM EMP) AS INIT_TBL(LAST_INIT)
```

In this example the difference in CPU consumed between a merged and materialized query may not make much difference, but in situations with many levels of nesting, functions and expressions, and multiple references to the columns resulting from the functions and expressions, it can be better to force materialization using a RAND() function. This has proved to be quite successful for large queries, saving hours of CPU time. Basically, if you have such a complex query and it is running for hours consuming mostly CPU then you may consider forcing materialization.

DB2 11 for z/OS has introduced a potential performance improvement to repeated executions of expressions in merged nested table expressions with the introduction of a feature called “expression sharing”. This feature is currently activated via a hidden installation parameter, QUERY_REWRITE_DEGREE, which is set to 1 by default. Setting the value to 2 will enable expression sharing and enable

the reuse of expression results in the outer portion of the query. This would, in effect, get all of the advantages of merge without the overhead of repeated expression execution. Keep in mind that this is a hidden installation parameter that applies to the entire subsystem and should be tested with extreme caution and under the guidance of IBM to see if it will help in your environment.

Another challenge in the performance of nested table expressions is the ability of DB2 to push predicates into a materialized table expression or view. Prior to DB2 11 for z/OS there is limited ability for DB2 to push predicates into table expressions, so you may have to do it manually. It's best to check the transformed query that is part of EXPLAIN output to determine whether or not a table expression has been materialized and which predicates against it have been pushed down into the table expression. If redundant predicates are generated by predicate transitive closure, please keep in mind that the process of predicate pushdown or predicate distribution is going to happen before transitive closure is applied. In this situation, if the predicates generated by transitive closure could be pushed down they are not and you will have to redundantly code the predicates yourself.

DB2 has the ability to merge some correlated nested table expressions with the outer portion of the statement that references them, and although I recommend correlated table expressions to force a nested loop access path (see Section 3), it may not always be effective.

Scalar Fullselect

Scalar fullselects are not merged with the referencing statement and are executed depending upon where they are coded in the SQL statement. In the following query, either correlated or non-correlated scalar fullselects are executed for every row processed within the SELECT clause. In the WHERE clause, the non-correlated scalar fullselect will be evaluated only once and the result will then be used in the predicate as Stage 1; but for a correlated reference it would be evaluated for each row as a Stage 2 predicate.

```
SELECT EMPNO, (SELECT COUNT(*)
               FROM   EMP SE
               WHERE  SE.WORKDEPT = E.WORKDEPT)
               AS DEPT_HEAD_COUNT,
CASE WHEN E.SALARY = (SELECT MAX(SALARY)
                     FROM   EMP MS)
      THEN 'MAX SALARY'
      ELSE 'REGULAR PERSON'
END AS MAX_INDICATOR
FROM   EMP E
WHERE  SALARY BETWEEN
      (SELECT AVG(SALARY)
       FROM   EMP) AND
      (SELECT MAX(SALARY)
       FROM   EMP EP WHERE EP.WORKDEPT = E.WORKDEPT)
```

Common Table Expressions

Common table expressions can be merged with the outer statement that references them, but can also be materialized. If a common table expression is referenced more than once it is materialized.

UNION in View and Table Expression Distribution

Placing a UNION ALL in a nested table expression or view can be a potent construct that can enable significantly powerfully complex processing within a single SQL statement. Some of this can include:

- Multi-table search
- Temporal table access
 - Both homegrown and DB2 provided
- UNION in a view
 - Flexible table design
 - Ultra-large “table” support

While this construct can be extremely powerful it can also become extremely expensive, so a thorough knowledge, combined with some testing, can create an understanding of the potential costs and benefits. One of the important aspects to understand is that any predicates and joins coded against the UNION in the table expression can be distributed across each block of the UNION if the table expression is merged and not materialized. The following query exhibits a very powerful search in which information about employees that are “useful” (working on or responsible for projects) is returned.

```
SELECT DISTINCT E.EMPNO, E.LASTNAME
FROM   EMP E
INNER JOIN
(
  SELECT EMPNO
  FROM   EMPPROJECT
  UNION ALL
  SELECT RESPEM
  FROM   PROJ) AS WORK(EMPNO)
ON WORK.EMPNO = E.EMPNO
WHERE E.WORKDEPT = 'C01'
```

The advantage is the power of the query, but the potential performance disadvantage is that there may be redundant table access. A human being may look at this query and say that three tables are being accessed: EMP, EMPPROJECT, and PROJ. However, the fact of the matter is that the join is distributed across the UNION ALL, so there are really four tables being accessed because the join to EMP is repeated for each block of the UNION. Since there is no “memory” between the query blocks the EMP table is read twice. This can be especially expensive when joining UNION ALL table expressions to other UNION ALL table expressions. Such is the case of system-period temporal tables when a period-specification is utilized in the FROM clauses. Imagine if the EMP and DEPT tables were

system period temporal tables then the following query, which appears to join two tables, would actually join $(2^n) \times n$ times where n is the number of tables in the coded join. In this case 8 tables are joined; EMP to DEPT, EMP history to DEPT, EMP to DEPT history, EMP history to DEPT history.

```
SELECT * FROM EMP FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP E
INNER JOIN    DEPT FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP D
ON E.WORKDEPT = D.DEPTNO WHERE E.EMPNO = '000010';
```

When joining to UNIONS and join distributions becomes significant, you may want to consider programmatic joins rather than SQL joins. You could also choose to introduce a correlated reference. The following example is a modification of our query to report on “useful” employees. In this example, correlation is used to avoid multiple accesses to the EMP table. Please be careful with these types of queries and test to see if they will really be of benefit in your situation.

```
SELECT DISTINCT E.EMPNO, E.LASTNAME
FROM    EMP E
INNER JOIN TABLE
(
  SELECT EMPNO
  FROM    EMPPROJECT EPA
  WHERE   EPA.EMPNO = E.EMPNO
  UNION ALL
  SELECT RESPEMP
  FROM    PROJ P
  WHERE   P.RESPEMP = E.EMPNO) AS WORK(EMPNO)
ON WORK.EMPNO = E.EMPNO
WHERE E.WORKDEPT = 'C01'
```

Using UNION ALL in a view can be a powerful construct that can serve many purposes:

- Rotate individual tables in and out of a view
- Make tables that are larger than largest possible tables
- Separate a very large table design into smaller tables
 - May be easier to manage
- Operate a temporal design
 - Base table
 - History table
 - Retrieve data from both as one

One common design is to create a history table for each year for a certain number of years. A view is created with a UNION ALL to concatenate several tables together and make them appear as one:

```
CREATE VIEW
ACCT_HIST
(ACCT_ID,ACCT_YEAR
,ACCT_ACT_TSP
,ACCT_BAL
,ACTIVE_FLG
,TRAN_CDE) AS
(SELECT ACCT_ID,ACCT_YEAR
,ACCT_ACT_TSP
,ACCT_BAL
,ACTIVE_FLG
,TRAN_CDE
FROM ACCT_HIST_2011
WHERE ACCT_YEAR='2011'
UNION ALL
SELECT ACCT_ID,ACCT_YEAR
,ACCT_ACT_TSP
,ACCT_BAL
,ACTIVE_FLG
,TRAN_CDE
FROM ACCT_HIST_2012
WHERE ACCT_YEAR='2012'
)
```

In this way, a query can be made against the ACCT_HIST view as it appears to contain the data for multiple years in a single table. Predicates coded against this view will be distributed across all blocks of the UNION ALL. If you notice there are already predicates coded within the view that seem unnecessary. However, when a query is coded against the view any predicates will be distributed across all the query blocks and any “impossible” predicates will cause an entire query block to be pruned. For example, the following query against the view will result in only the first block of the UNION ALL to be executed since the predicate ACCT_YEAR = ‘2011’ will be distributed resulting in a compound predicate of WHERE ACCT_YEAR = ‘2011’ AND ACCT_YEAR = ‘2011’ for the first query block and WHERE ACCT_YEAR = ‘2011’ AND ACCT_YEAR = ‘2012’ for the second query block.

```
SELECT *
FROM ACCT_HIST
WHERE ACCT_YEAR = '2011'
```

Please keep in mind that predicate distribution can occur for a variety of queries against UNION in a view. However, query block pruning, as demonstrated above, is only possible for predicates containing literal values and host variables and not joined columns. If you are expecting query block pruning based upon joined columns, then the performance choice would be a programmatic join so that the literal value or host variable can be provided in the predicate versus the join column.

Impact of UNION on System-Period Temporal Time Travel Queries

System-period temporal tables are a powerful feature that was introduced with DB2 10 for z/OS. Having the DB2 automation replace significant quantities of application programming was a huge advantage with regards to simplifying the application development process and reducing time to delivery for a solution. Customers began implementing system-period temporal base tables and history tables, and coding time travel queries against these tables. One thing that became evident quite quickly, however, was that always using a time travel query to retrieve data could become a challenge from a performance perspective. This was because a system-period time travel query is always transformed into a UNION ALL between the base table and the history table (see section above on UNION distribution of joins for more detail). Therefore, even if “as of now” data was desired, both the base and history table were accessed. This performance impact became even more significant with table joins, as joins have to be distributed across base and history table. So, what appeared to be a two table join between two system-period temporal tables with period-specifications actually became a join of two tables four times for a total access of eight tables. Given that current data represented the majority of typical access, a performance hit was taken 100% of the time. Application developers had to code separate queries to retrieve current data versus historical data in order to maintain an acceptable level of performance.

Imagine a system-period temporal table called T_EMP. In the following query, a period-specification is used to access the table:

```
SELECT LASTNAME
FROM   T_EMP FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP
WHERE  EMPNO = '000010'
```

After the query transformation component of DB2 has transformed the above query, it would look like this:

```
(SELECT LASTNAME FROM T_EMP WHERE ( T_EMP.EMPNO = '000010'
AND T_EMP.START_TS <= CURRENT_TIMESTAMP
AND T_EMP.END_TS > CURRENT_TIMESTAMP)
UNION ALL
SELECT LASTNAME FROM T_EMP_HIST WHERE ( T_EMP_HIST.EMPNO =
'000010'
AND T_EMP_HIST.START_TS <= CURRENT_TIMESTAMP
AND T_EMP_HIST.END_TS > CURRENT_TIMESTAMP))
```

If all the application wanted was current data, then it would have to access the T_EMP table directly without a period-specification. That is the only way the UNION ALL could be avoided in DB2 10 for z/OS.

With the introduction of the CURRENT TEMPORAL SYSTEM_TIME special register in DB2 11 for z/OS, this potential performance issue is addressed. During statement compile time (prepare for dynamic or bind for static) an extended section is added to the statement access path. DB2 will build two sections for the table access for queries that do not specify a period-specification; one without the UNION ALL and temporal predicates and a second with the UNION ALL and temporal predicates. DB2 will do this only if the package bind parameter SYSTIMESENSITIVE is set to YES, which is the default. The creation of the alternate access paths for dynamic statements is also dependent upon the package bind parameter for the dynamic package the application is using. The choice of access path at execution time is then dependent upon the setting of the CURRENT TEMPORAL SYSTEM_TIME special register. If the value is set to the null value, then the original section is executed and no UNION ALL or time travel predicates apply. If the value of the special register is set to something other than the null value, the extended section is used, including the UNION ALL and time travel predicates. Since the null value is the default for this special register, an existing table can be converted to a system-period temporal table without application impact at all. A small application change to include the setting (or not setting) of the special register can then dictate whether or not the query is a base query or time travel query, again without changes to the original statement. For example, the following statement against a system-period temporal table:

```
SELECT LASTNAME
FROM   T_EMP
WHERE  EMPNO = '000010'
```

would actually have two separate access paths if SYSTIMESENSITIVE was YES. The first access path would represent the statement as coded, and the second would represent the time travel query:

```
(SELECT LASTNAME FROM T_EMP WHERE ( T_EMP.EMPNO = '000010'
AND T_EMP.START_TS <= CURRENT TEMPORAL SYSTEM_TIME
AND T_EMP.END_TS > CURRENT TEMPORAL SYSTEM_TIME)
UNION ALL
SELECT LASTNAME FROM T_EMP_HIST WHERE ( T_EMP_HIST.EMPNO =
'000010'
AND T_EMP_HIST.START_TS <= CURRENT TEMPORAL SYSTEM_TIME
AND T_EMP_HIST.END_TS > CURRENT TEMPORAL SYSTEM_TIME) )
```

The value of the CURRENT TEMPORAL SYSTEM_TIME special register would control which access path is used, original or extended, and any query that requests “as of now” data using a CURRENT TEMPORAL SYSTEM_TIME set to the null value would not incur the overhead of the UNION ALL. This same performance feature is also available for the new archive enabled tables.

Query Parallelism

Query parallelism can be a significant performance enhancer from an elapsed time perspective, but will not result in less CPU consumed. Parallelism is controlled across a subsystem via several installation parameters (CDSSRDEF, PARAMDEG, PARAMDEG_DPSI, PARA_EFF) and via the CURRENT DEGREE special register or DEGREE bind parameter. Query parallelism allows for DB2 to receive a query for processing, break that query into multiple pieces to execute in parallel, and then merge the

individual result sets together for a final return to the process executing the query. There are two settings for parallelism at the application level: “ANY”, or “1”. These settings control whether or not parallelism is considered for queries issued by an application. When in use, the parallelism is limited only by available resources and the subsystem defined limits PARAMDEG (maximum degree of parallelism) and PARAMDEG_DPSI (maximum degree of parallelism when a DPSI is used to drive the parallelism). It is quite possible to achieve absolutely astounding query response time for large queries when parallelism is utilized and adequate system resources are available.

Query parallelism is great for data warehouses and report queries, but not great for transactions. Parallelism can be enabled by default (subsystem parameters) or at the application level, and if the application is an OLTP then parallelism may be a performance detriment. This negative is due to the fact that the extra overhead involved in establishing and coordinating the parallel tasks will almost always exceed what would have been the expected or normal response time for a query that processes little. In addition, that coordination will increase CPU consumption for transactions. Therefore, in general, using parallelism for transactions is not a good idea from a performance perspective. However, it could possibly be a good idea from a cost perspective since CPU for parallel tasks can be offloaded to cheaper zIIP processors.

Parallel tasks spawned within a query can be offloaded to zIIP engines if zIIPs are available. Again, this is great for large report type queries, but still may not be the best for transactions. The z/OS parameter IIPHONORPRIORITY setting can also significantly affect the execution of parallel tasks. Please see the article, “DB2 Query CPU Parallelism and IIPHONORPRIORITY” by Jeff Lane of J. Lane Consulting, on the IDUG Content Committee blog at the IDUG.org website. The general recommendation is to utilize parallelism for large queries at times when the machine resources are available, and to not use parallelism for transactions. Reference Sections 6 and 7 for database and subsystem configurations to improve parallelism. In addition, the DB2 Installation Guide and the DB2 for z/OS Managing Performance Guide both have great information on parallelism and tuning for parallelism.

Sequential Detection and Index Lookaside

The significance of these two performance enhancers cannot be overstated. Sequential detection is DB2’s ability to detect a sequential access pattern, at a thread level and across one or multiple units of work, for table spaces and indexes. When a sequential access pattern is detected DB2 will launch one or more sequential prefetch engines that are independent of the thread requesting the data, in an effort to stage the data in advance of the thread request. This can result in amazingly efficient read I/O response times or zero read I/O response times in some situations. DB2 will continue to monitor the access to confirm whether or not pages of data are being requested in a sequentially available fashion and can engage or disengage the sequential prefetch on the fly. Index lookaside is a method in which DB2 keeps track of the index value ranges and checks whether or not an index entry requested is on the same leaf page as the previous request. If DB2 determines that the key being requested is indeed on the same leaf page as the previous request, then the entry can be returned without a traverse of the index b-tree or even a getpage request for the leaf page. This can result in a tremendous amount of CPU saved for index intensive operations.

It is important to note that the mechanisms for sequential detection and index lookaside are stored in the thread storage of the thread making the SQL requests. Each thread thus has its own storage for these mechanisms independent of other threads, and whether or not this storage is maintained across the boundaries of a unit of work is dictated by the RELEASE setting of the package the thread

is using for the process. If the setting is `RELEASE(COMMIT)` then you are in essence telling DB2 that, “when I commit, I am not coming back”. So, the storage is released on every commit and the potential for index lookaside and sequential detection is reduced. If the setting is `RELEASE(DEALLOCATE)` then the memory for these mechanisms is retained across a unit of work, and the potential for sequential detection and index lookaside greatly increases. For this reason the setting of `RELEASE(DEALLOCATE)` is highly recommended for batch processing, as well as for high speed transaction processing. For transaction processing under CICS, you’ll have to use protected threads as well as `RELEASE(DEALLOCATE)`, and for remote threads, you will need to employ high performance DBATs (described in Section 3). `RELEASE(DEALLOCATE)` is further described in the `BIND` options section of Section 3.

Effective use of these performance enhancers depends greatly upon the design of your database and application processes. A common cluster amongst joined tables, or even tables accessed individually in a repeating pattern, can affect sequential detection and index lookaside. In addition, for large batch processes the input to the process can be sorted to match the clustering sequence of the search in order to take advantage of these features. Even with a massive skip sequential process, I have witnessed a benefit to sequential detection (you probably won’t get index lookaside for a skip sequential process); whereas, even with a terrible buffer hit ratio of -241%, sequential prefetch was still a better performer than random access. With an input to the process organized in the same sequence as the data stored in the indexes and tables, `RELEASE(DEALLOCATE)`, protected threads, and high performance DBATs, you can achieve extreme transaction rates by taking advantage of these performance enhancers.

Section 3:

Coding SQL for Performance

SQL is far more than a data access language. It is a fully functional programming language with significant flexibility. It's also fairly easy to learn, which could be a good or bad thing. Most of the time programmers are taught just enough SQL to access the information they need from the database, and not any more. This results in many simple SQL statements that pull the minimal required information to solve a given access need. In other situations, the developer may be using a tool to access the database, and so is not in complete control of the SQL being generated and executed. IBM is doing a great job of making SQL processing more efficient with every new release of the software. This offsets some of the performance implications of SQL that is generated by tools and created by developers with the minimum level of experience. If good performance is desired then DB2 is a good choice. If high performance and cost savings is desired then SQL must be coded with performance in mind. This does not mean that every statement has to be coded with performance in mind. It simply means that for critical processing where performance is of primary concern each SQL statement needs to take performance into consideration. This represents a trade-off: time spent coding and testing versus performance of the SQL statement at execution time. Good architects and developers need to understand when speed and flexibility of development are important and when SQL statement performance is important and strike a balance. Looking at the application design and predicted or current transaction rates will quickly point to the area of logic that will need the most attention. Is there a transaction that is executed 1,000 times per day? It may be wasteful to spend time designing this transaction with performance as the primary concern. What about the transaction that is executed 200 million times per day? Obviously any performance considerations made to the design of the SQL in this transaction will have a significant impact on performance and operational costs.

Spend your time reasonably and pick your battles accordingly. Speed and flexibility of application development is paramount. It will be completely impossible to inject SQL performance concepts into every piece of application logic written. In the vast majority of cases it's a matter of picking a few critical processes and focusing on performance for those that will make the world of difference.

The Most Efficient SQL Statement

Have you ever heard the joke that the most efficient SQL statement is the one that is never executed? Well, nothing could be truer! Calling the database can be expensive, especially when it's called across a network. The first rule for efficient SQL coding is to only call the database when it is absolutely necessary. The second rule is that if you must call the database, then you should do the most work in the fewest number of calls. The third and final rule of efficient SQL coding is that when you call the database and do the most work possible, you code the statement in a format that the database can use for performance.

First Rule: Call the Database Only When Necessary

This statement cannot be overemphasized! The single largest impact to application performance today is the frequency of SQL execution. Many generic application designs lead to the over-execution of simple SQL statements. The only way to achieve a higher level of performance in these situations is to change application code. There are no database or SQL solutions. Therefore, it is really important to consider the

frequency of access for certain portions of your application and to code database access efficiently in these situations. Do you already have an object in memory? Is it really important to re-access the database objects again? Here are some suggestions for avoiding frequent SQL executions:

- Cache common objects in memory. This is especially important for objects such as code tables. You can place access counters, update detection, or time limits against objects that are placed in cache and then refresh them at appropriate times. Examples of such objects include arrays in batch programs, CICS transient data queues, and Java Caching System.
- Utilize an optimistic locking strategy. DB2 supplied mechanisms, discussed in this section as well as in Section 6, can be used to help applications avoid unnecessary calls to the database to retrieve data prior to changing that data.
- Combine multiple SQL statements into one statement. More details on this below.
- Do not use DB2 for transient data storage. In many situations people are creating tables to persist data across disparate processes. They are familiar with DB2, so this is a simple thing to do. If you are concerned with performance, then you must understand that there is nothing good about this. This data belongs in shared memory, flat files, or message queues, and not in DB2. If you are forced to store transient data in DB2 then understand it is the worst choice from a performance perspective.
- Let DB2 automation do some of the work for you. Features such as triggers and database enforced constraints allow for database automation to replace multitudes of SQL statements. See sections in this section and Section 6 about utilizing database automation for performance.

This first rule is easily the most important. Data intensive work should be in the database engine. If it is not, then you end up with a proliferation of SQL calls that each, on their own, do very little work. If the work is in the database, then it can be exposed via EXPLAIN and can be tuned by a DBA. If the work is done in the application, there's nothing that can be done on the database side of things.

Second Rule: Do the Most Work in the Fewest Calls

One way to reduce the quantity of SQL issued in an application is to combine multiple SQL calls into a single SQL call. There are many ways to do this, and in fact, the vast majority of the content of this book is focused on advanced SQL calls and processing that are designed to reduce the number of times the database is actually called. The basic question to ask yourself is: If I'm going to access table A, will I also be accessing table B? If the answer is yes, then from a performance perspective you should be coding access to both tables in a single statement.

- Code joins versus single table access. This is as simple as it gets. Do you need data from two tables that are related? Code a two-table join versus a programmatic join via separate SQL statements. The advantage to a join versus individual SQL statements is twofold: first, it avoids excessive SQL calls, and second, DB2 chooses the most efficient access path versus the application developer coding the statements. That second point is extremely important to understand. If you are coding a join in a program, you are forcing the table access sequence, which is an extremely critical factor in database performance. If you code a SQL join the database optimizer makes a decision as to which table to access first. The database decision is based upon statistics and changes due to changes in tables,

indexes, and table contents. The multi-table access path coded in an application program does not take any of this into account and will not change unless the developer modifies the code. In simple tests that I have conducted, a simple two-table join outperformed an equivalent programmatic join by 30%.

- Use database enforced referential integrity. Database enforced RI is discussed in detail in Section 6 of this book, but the bottom line is that if there are relationships to check and deletes to restrict or cascade then have the database do the work rather than the application. This seriously reduces the calls to the database. From a performance perspective, one of the silliest decisions a DBA can make is to not implement a cascading delete in the database and insist the application cascade the deletes itself.
- Use triggers for multi-table changes. If there is a need to modify the contents of one table when the contents of another table change, then this can be accomplished within the database engine via database triggers. Triggers are discussed in detail in Section 6 of this book, but the bottom line is that they eliminate multiple database calls.
- Use some database automation. Sequence objects, identity columns, temporal and archive tables all allow for multiple database calls to be replaced by single calls that result in multiple actions. Please read about these features in Section 6 for more information.
- Read from result tables. The table-reference in the FROM clause of a subselect includes something called a data-change-table-reference, which allows for read access to a result set of an INSERT, UPDATE, DELETE, or MERGE statement. This includes the ability to generate additional data for processing within the context of the entire statement. Further information about this can be found in the nested table expression section of this section.

Using SQL coding techniques to consolidate database calls is extremely critical to application performance. The key to this is knowledge. Explore the database features and test them to see if they work in your environment. Then you can make common guidelines for coding these in your environment. The result can be significant improvements to application performance without much pain for the designers and developers.

Third Rule: Code Performance Effective SQL Statements

If you have paid respect to the first two rules of coding SQL for performance then you are well over 80% on your way to achieving high performance. As you code more complex SQL statements in an effort to reduce the calls to the database, you place more responsibility for the performance of those statements from the application developer to the database engine. This is very good for the developer, and in the vast majority of circumstances, it is very good for overall application and database performance. The DB2 engine has evolved over the years, and the SQL optimization in the product is substantially improved with each release of the product. SQL performance is statistics based and query performance will adapt as your databases change over time. In more than 90% of opportunities, DB2 will choose an access path that is the best performance choice.

If the ultimate in application performance is desired, there is still some responsibility in the hands of the developer and DBA to understand how query optimization works and how some SQL coding techniques can be utilized to give the DB2 engine better optimization choices. Remember, the basic underlying premise of great SQL performance is to filter as much as possible as early as possible.

Advancing to this level of SQL performance requires an understanding of how DB2 works. The majority of the rest of this section, as well as Sections 3, 4, and 5, are all dedicated to understanding SQL optimization and improving the performance of individual SQL statements.

An understanding of the relativity of SQL performance is important. What's the better performing SQL statement: the statement that does a table space scan or the one with matching index access? There is no correct answer except that it depends upon how much data is expected to be returned to the application. If the application requires all of the data from a table then a table space scan is a very appropriate and efficient access path. If zero or one row is expected then the matching index access is very likely the best performing access path. What about when the expected result set quantity is somewhere between zero and infinity? That is where the challenge lies, and an understanding of the engine is important to guarantee the best performance.

The Basics of Coding SQL Statements for Performance

There are some very simple and basic rules of coding an individual SQL statement for performance. For the most part, these recommendations will generally not result in significant performance improvements, but are good general techniques. If they are implemented as part of standard guidelines for SQL program across an enterprise, they can make a significant impact. There is certainly no reason to go back and change existing SQL statements to conform to these guidelines.

Retrieve only the Rows Required

In Section 2 of this book I discussed the types of predicates and how they are processed by DB2, including Stage 1 and Stage 2 predicates. There is also the fictitious Stage 3 predicate where rows are retrieved into an application and then discarded there. There is absolutely nothing efficient about Stage 3 predicates. Any filtering should be in a WHERE clause predicate in a SQL statement. In some situations, developers have coded Stage 3 predicates as a means of avoiding DBA's rejecting the SQL upon a performance review because the SQL contains Stage 2 predicates. The fact of the matter is that a Stage 2 predicate is always better than a Stage 3 predicate. In addition, there are constant performance enhancements to the DB2 engine and many predicates that used to be Stage 2 predicates are no longer Stage 2 predicates, or the processing of Stage 2 predicates has become more efficient. The bottom line is to filter the data in the WHERE clause predicate of a SQL statement.

Retrieve only the Columns Needed

This is challenging due to the fact that adjusting the number of columns returned from a table based upon the needs of a particular transaction or process means coding either more SQL statements or using dynamic SQL with constantly changing SELECT lists. The performance ramifications of lengthy columns lists can be significant depending upon the frequency of execution and number of columns retrieved. I have personally conducted performance tests where the only difference between statements executed were the number of columns retrieved and have seen significant CPU reduction for the reduced column list. So, while coding multiple statements may be a bad idea from a development perspective it is going to be smart from a performance perspective. This is one of those situations where it is definitely a benefit to identify the most often executed statements and transactions, and to consider the column requirements for those specific processes. If there is something to gain from a shorter column list in those situations it may be a benefit to code a special SQL statement. Always run some benchmark tests, as described in Section 9 of this book, and

document the potential CPU savings for these transactions. That should make it easier to make the decision whether or not special code is needed for those transactions. In situations with extremely high transaction rates, retrieving only the columns required can have a huge impact on CPU consumption and ultimately the operational cost of the application. There is a balance between the cost of additional program logic and CPU savings that is worthy of exploration.

Code the Most Selective Predicates First

DB2 processes predicates in a WHERE clause in a specific sequence that is well documented in the DB2 for z/OS Managing Performance Guide. Basically, predicates are processed, and thus rows filtered, in a given sequence by stage and then in a specific order within that stage. Predicate stages are documented in Section 2 of this book. In general the sequence of predicate execution is:

1. Index matching predicates in matching column sequence
2. Index screening predicates
3. Page range screening predicates
4. Other Stage 1 predicates
5. Stage 2 predicates

Within each of the steps listed above and after the index predicates, the predicates are applied in the following order:

1. Equality predicates
2. Range predicates
3. Other predicates

Within each of these subgroupings, predicates are evaluated in the order coded in the SQL statement. The bottom line rule of thumb is to code the predicates in the WHERE clause in the sequence of what you believe is the most to least restrictive.

Code Efficient Predicates

Assure that predicates are being coded efficiently and that these efficiencies are part of your SQL programming guidelines. Section 2 of this book documents the different DB2 predicate types. You should use Stage 1 index matching predicates whenever possible, trying to include all leading columns of the potential matching index. You should also avoid coding functions or expressions against columns of a table in a predicate as it is likely a Stage 2 predicate, and thus non-indexable. You should also use equalities whenever possible to ensure the finest level of filtering and greatest potential for index matching. There are many exceptions to this rule, and DB2 can convert what is potentially a Stage 2 predicate into a Stage 1 index matching predicate in some situations. Be mindful of the fact that a Stage 2 predicate that is connected to a Stage 1 predicate with an OR in a compound predicate makes the entire compound predicate Stage 2.

Use Explicit Join Syntax

I almost always encourage people to use explicit join syntax versus implicit join syntax. For example, the join between the EMP and DEPT DB2 sample tables can be coded one of two ways:

Implicit join syntax:

```
SELECT A.LASTNAME, B.DEPTNAME
FROM   EMP A, DEPT B
WHERE  A.WORKDEPT = B.WORKDEPT
AND    B.DEPTNO = 'C01'
```

Explicit join syntax:

```
SELECT A.LASTNAME, B.DEPTNAME
FROM   EMP A
JOIN   DEPT B
ON     A.WORKDEPT = B.WORKDEPT
WHERE  B.DEPTNO = 'C01'
```

While this is not particularly a direct performance recommendation, it can be extremely helpful when diagnosing SQL problems and also when tuning SQL statements. There are several reasons to favor explicit join syntax:

1. It is easier to read and understand than implicit join syntax, since the join predicates are in the ON clause and the filtering predicates are in the WHERE clause.
2. It is harder to miss coding a join predicate. The explicit join syntax requires a join predicate be coded in an ON clause. If you use implicit join syntax you are mashing together join predicates with filtering predicate, which can lead to confusion and poor performing SQL.
3. It is very easy to change an inner join to an outer join if you code explicit join syntax. In the example that uses explicit join syntax above you can simply replace the word “JOIN” with “LEFT JOIN” to change the query from an inner to a left outer join. If you are using implicit join syntax for your inner joins then you have to rewrite the query. You may want to convert inner joins to left joins when attempting to influence table selection sequences (see Section 5 for more details).

Code Efficient Host Variables

Even though the DB2 engine is extremely tolerant of data type mismatches, it is always a good practice to code and bind host variables that match the data type of the corresponding columns to your SQL statements. Using host variables or parameter markers in queries is extremely important from a performance perspective, and you should be encouraged to code using host variables and parameter markers versus using literal values, especially for dynamic SQL. Keep in mind, however, that using a host variable or parameter marker will generally hide a value from the optimizer and put DB2 in a position where it will have to use default filter factor calculations to determine the potential filtering for a predicate. This is not necessarily a bad thing as static, embedded SQL and dynamic statements coded with parameter markers can reuse already built access paths and avoid incremental binds or long prepares. However, keep in mind that the access path may not be optimal

in situations where there may be data skew. In these situations, there are many options for improving the access paths of these statements. All of these options are described in Section 5 of this book, and include:

- Access path reoptimization
- Selectivity overrides
- Statement and package level hints

If you do code literal values into dynamic SQL statements and are not accessing heavily skewed data, then a potential performance option may be literal concentration, which is described in Section 5 of this book.

Subqueries and Joins

When coding multi-table access in a SQL statement there are several choices with regard to performance. With every release of DB2 these decisions become less relevant due to the improved power of query transformation, also known as query rewrite (discussed in detail in Section 2 of this book). Nonetheless, I always code my multi-table access queries multiple ways, EXPLAIN each option, and benchmark each option as well. There are basically three ways you can code your multi-table access queries.

- Non-correlated subquery
- Correlated subquery
- Join

Which is the best for your situation? Well, it depends of course, but here are some guidelines to choosing the right query for the application. For reference here are three queries that each return the same result set coded in each of the three ways:

Non-correlated subquery:

```
SELECT WORKDEPT, EMPNO, LASTNAME
FROM   EMP EMP2
WHERE  WORKDEPT IN
      (SELECT DEPTNO
       FROM   DEPT DEPT
       WHERE  DEPTNAME LIKE '%SYSTEMS%')
```

Correlated subquery:

```
SELECT WORKDEPT, EMPNO, LASTNAME
FROM EMP EMP2
WHERE EXISTS
      (SELECT 1
       FROM DEPT DEPT
       WHERE DEPT.DEPTNO = EMP2.WORKDEPT
       AND DEPT.DEPTNAME LIKE '%SYSTEMS%')
```

Join:

```
SELECT EMP2.WORKDEPT, EMP2.EMPNO, EMP2.LASTNAME
FROM EMP EMP2 INNER JOIN
      DEPT DEPT
ON DEPT.DEPTNO = EMP2.WORKDEPT
WHERE DEPT.DEPTNAME LIKE '%SYSTEMS%'
```

Please keep in mind that the query transformation component and/or the access path selection components of the DB2 engine (discussed in detail in Section 2) can change non-correlated subqueries into correlated subqueries or joins, and correlated subqueries into non-correlated subqueries or joins. For our discussion here we'll discuss the performance of these queries as if they were not transformed by DB2.

Non-Correlated Subquery

Non-correlated subqueries that are not transformed into a correlated subquery or join are processed in a “bottom up” fashion. The subquery is executed first, and the result of the subquery is sorted and placed into a workfile, and is then used as a Stage 1 indexable predicate (described in Section 2) in the outer portion of the query. An index can be used for matching index access in the outer query. If an index is not used, then the predicate is processed as either Stage 1 or Stage 2. DB2 can build a sparse index on the fly in support of faster access to the table of values generated by the subquery.

Non-correlated subqueries are useful when you need multi-table access, but only need data from one of the tables, and they're especially useful when the data from the inner table needs to be aggregated or modified in some way. One way in which I've utilized non-correlated subqueries is in support of generic search screens that utilize a variety of range values as input. For example, suppose there is a generic search screen for employees in my DB2 sample database. The screen supports full range searches on the employee's education level, hire date, and bonus. If the query was coded as a simple generic query that contains the three range predicates, it can at best get single column matching index access.

```
SELECT *
FROM   EMP A
WHERE  BONUS BETWEEN ? AND ?
AND    HIREDATE BETWEEN ? AND ?
AND    EDLEVEL BETWEEN ? AND ?
```

Given that the domains of the education level and hire dates are limited, code tables can be built that contain the entire domain of values for these columns. Then non-correlated subqueries can be used to generate the qualifying values based upon filters against these limited domains for three column matching index access against the employee table (given the appropriate supporting index).

```

SELECT *
FROM   EMP A
WHERE  BONUS BETWEEN ? AND ?
AND    HIREDATE IN
      (SELECT VALID_DATE
       FROM   VALID_DATES
       WHERE  VALID_DATE BETWEEN ? AND ?)
AND    EDLEVEL IN
      (SELECT VALID_ED_LEVEL
       FROM   VALID_ED_LEVELS
       WHERE  VALID_ED_LEVEL BETWEEN ? AND ?)

```

Queries such as this have been used to enable generic searches against very large tables to utilize matching index access for dramatic query performance improvements.

Here are some general guidelines for when you want to code a non-correlated subquery versus a correlated subquery or join.

- When the expected result of the subquery is small compared to outer query
- When the subquery result is NOT large
- When the outer query can utilize a matching index on the subquery results
- When there is no available index on the matching columns of the subquery

Please remember that DB2 can transform a non-correlated subquery into a correlated subquery or a join. Nonetheless, you should still follow these guidelines. As always, an EXPLAIN and a benchmark test will tell you which type of query will perform the best.

Correlated Subquery

Correlated subqueries that are not transformed into a non-correlated subquery or join are processed in a “top-bottom, top-bottom, top-bottom, etc.” fashion. The outer query is executed first, and for each qualifying row of the outer query the correlated subquery is executed. The correlated subquery is considered a Stage 2 predicate for the outer query as the correlated subquery cannot be executed until the data from the outer query is available to resolve the correlated reference. The predicate that contains the correlated reference in the subquery can be an indexable Stage 1 predicate.

Correlated subqueries are useful instances when the result of a query is dependent upon an existence check, such as the following query.

```

SELECT EMPNO, LASTNAME
FROM   EMP E
WHERE  EXISTS
      (SELECT 1
       FROM   PROJ P
       WHERE  P.RESPEMP = E.EMPNO)

```

Here are some general guidelines for when you want to code a correlated subquery versus a non-correlated subquery or join.

- When the subquery result is potentially larger than the outer query
- When the outer query has no supporting index on the matching columns
- When there is good index matching potential for the subquery

Please remember that DB2 can transform a correlated subquery into a non-correlated subquery or a join. Nonetheless, you should still follow these guidelines. As always, an EXPLAIN and a benchmark test will tell you which type of query will perform the best.

Join

Joins are an important performance choice because DB2 can choose the table access sequence based upon available indexes and statistics (be mindful that subqueries can be transformed). Of course, if you need data from both of the tables involved then a join is going to be a more obvious choice over a subquery. If you decided to try coding a join versus a subquery, then be aware of the fact that a join may introduce duplicate rows in some situations, and you may need to add a DISTINCT clause in the SELECT to eliminate those duplicate rows in the result set.

Here are some general guidelines for when you want to code a join versus a correlated or non-correlated subquery.

- There's good matching index potential for outer query and subquery
- The subquery converted to a join doesn't introduce duplicate rows in the result

When thinking about join performance, the basic idea is that you generally want DB2 to first access the table that qualifies the fewest number of rows to. DB2 has a variety of join methods to choose from based upon the catalog statistics.

Anti-Join

An anti-join is a fabulous alternative to NOT IN and NOT EXISTS subqueries, and can represent a significant performance boost to these types of queries when data volumes are extremely large and/or workfile resources are constrained. An anti-join utilizes an "after join" predicate in an outer join to filter data based upon the fact that the relationship between the two tables does not exist. When an outer join is executed, DB2 supplies null values for the null-supplying table when the relationship between the two tables in the query does not exist. For example, in a left outer join the table on the right side of the join operation is the null-supplying table. When DB2 does not find rows in this right side table based upon the join predicate, nulls are returned for those left table rows. Therefore, if we code a predicate looking for those null values we can return rows of the left table that are not found in the right table. That's the same as a NOT IN or NOT EXISTS. The following three queries all return the same results, but their execution is extremely different. You can use the rules for coding subqueries versus joins above to help you determine which type of query to code; always remembering that EXPLAIN and benchmark tests will help you make the ultimate decision.

Non-correlated subquery:

```
SELECT WORKDEPT, EMPNO, LASTNAME
FROM   EMP EMP2
WHERE  WORKDEPT NOT IN
      (SELECT DEPTNO
       FROM   DEPT DEPT
       WHERE  DEPTNAME LIKE '%SYSTEMS%')
```

Correlated subquery:

```
SELECT WORKDEPT, EMPNO, LASTNAME
FROM   EMP EMP2
WHERE  NOT EXISTS
      (SELECT 1
       FROM   DEPT DEPT
       WHERE  DEPT.DEPTNO = EMP2.WORKDEPT
       AND    DEPT.DEPTNAME LIKE '%SYSTEMS%')
```

Anti-join:

```
SELECT EMP2.WORKDEPT, EMP2.EMPNO, EMP2.LASTNAME
FROM   EMP EMP2
LEFT JOIN
      DEPT DEPT
ON     DEPT.DEPTNO = EMP2.WORKDEPT
AND    DEPT.DEPTNAME LIKE '%SYSTEMS%'
WHERE  DEPT.DEPTNAME IS NULL
```

Table Expressions

Nested table expressions (hereafter often referenced as NTE) allow you to harness the power of SQL. They allow for the creation and manipulation of intermediate SQL results for further processing in an outer portion of a statement. Not only can you utilize nested table expressions to solve complex problems, but you can code them in such a way as to have an important performance impact. Table expressions can be a performance advantage or disadvantage and require a significant amount of practice to learn exactly how they will perform in your environment. From a logical perspective, a nested table expression produces an intermediate result table that can be further processed, but from a physical perspective, the table expression is subject to the concept of merge versus materialization that is described in Section 2 of this book.

In my opinion, table expressions include the following constructs:

- Nested table expressions
- Views
- Common table expressions

- SQL table functions
- Scalar fullselects in SELECT and WHERE clauses

There are many ways you can utilize table expressions for performance.

- Combining line item and aggregate information together in the same query for reports or transactions
- Searches using relational division
- Making decisions based upon aggregated or modified results
- Building queries piece by piece
- Pushing down during join predicates in outer queries
- Forcing materialization
- Forcing table access sequence

The number one performance advantage in using nested table expressions is the fact that they can replace significant quantities of application code and seriously reduce multiple calls to the database to solve a complex business problem. I have literally transformed application processes that ran for days into single SQL statements that ran for minutes by utilizing table expressions. This is especially true when an application is issuing many SQL statements to process data in pieces, and those statements are combined into one statement. A classic example of using the power of nested table expressions is in exploiting relational division. Relational division is a sexy way of saying join, but it allows you to search for things inside other things. That is, you can join a small quantity of data into a large quantity of data in order to find a relationship between the two. Analysis can be applied to the result using table expressions. Relation division is an extremely powerful tool in OLTP, decision support, and especially warehouse situations. In the following example, one of the employees in our sample database, Christine Haas, has decided to leave the company. Before she leaves we want to make sure that someone else has worked on projects she has worked on. Relational division provides a solution by joining the result of a query that returns the projects that Christine has worked on into the projects that others have worked on.

```
SELECT DISTINCT E.EMPNO
FROM   EMPPROJECT E
INNER JOIN
  (SELECT EPA.EMPNO, EPA.PROJNO
   FROM   EMPPROJECT EPA
   WHERE  EPA.EMPNO = '000010') AS F(EMPNO, PROJNO)
ON     E.PROJNO = F.PROJNO
AND    E.EMPNO <> F.EMPNO
ORDER BY E.EMPNO
```

Taking things a step further, we have come to realize that Christine is leaving due to the fact that she believes she works harder than the other employees and logs more time on projects than others that have worked on the same projects. Relational division can prove her right or wrong.

```
SELECT E.EMPNO, E.PROJNO, C_TIME, SUM(E.EMPTIME) AS O_TIME
FROM   EMPPROJACT E
INNER JOIN
  (SELECT EPA.EMPNO, EPA.PROJNO, SUM(EPA.EMPTIME)
   FROM   EMPPROJACT EPA
   WHERE  EPA.EMPNO = '000010'
   GROUP BY EPA.EMPNO, EPA.PROJNO) AS F(EMPNO, PROJNO, C_TIME)
ON     E.PROJNO = F.PROJNO
AND    E.EMPNO <> F.EMPNO
GROUP BY E.EMPNO, E.PROJNO, F.C_TIME
HAVING SUM(E.EMPTIME) > (SELECT SUM(G.EMPTIME)
                        FROM   EMPPROJACT G
                        WHERE  G.EMPNO = '000010'
                        AND    G.PROJNO = E.PROJNO)
```

I've also utilized relational division and common table expressions as another way to increase matching index access for search queries. The query below solves the same search request as the previous non-correlated subquery example earlier in this section using relational division and achieving three column matching index access (given the appropriate supporting index). Notice the use of the RAND() function in the common table expression to force materialization and encourage DB2 to access the result of the common table expression first in the access plan.

```
WITH SEARCH(HIREDATE, EDLEVEL, N) AS
  (SELECT VALID_DATE, VALID_EDLEVEL, RAND()
   FROM   VALID_DATES, VALID_EDLEVELS
   WHERE  VALID_DATE BETWEEN ? AND ?
   AND    VALID_EDLEVEL BETWEEN ? AND ?)
SELECT EMP.*
FROM   EMP EMP
INNER JOIN
  SEARCH SRCH
ON     EMP.HIREDATE = SRCH.HIREDATE
AND    EMP.EDLEVEL = SRCH.EDLEVEL
WHERE  EMP.BONUS BETWEEN ? AND ?
```

Correlated versus Non-Correlated Nested Table Expression

As with subqueries, nested table expressions can be correlated or non-correlated. Likewise, the way that DB2 processes non-correlated and correlated nested table expressions is extremely similar to the way it processes non-correlated and correlated subqueries that are not transformed. Thus, the same bottom-top and top-bottom, top-bottom situation exists. Here are two queries that are logically equivalent, but perform completely differently.

Non-correlated nested table expression:

```
SELECT EMPNO, SALARY, DEPTSAL.AVGSAL
FROM EMP EMP
LEFT JOIN
(SELECT WORKDEPT, AVG(SALARY)
FROM EMP
GROUP BY WORKDEPT) DEPTSAL(WORKDEPT, AVGSAL)
ON EMP.WORKDEPT = DEPTSAL.WORKDEPT
```

Correlated nested table expression:

```
SELECT E.EMPNO, E.SALARY, DEPTSAL.AVGSAL
FROM EMP E
LEFT JOIN
TABLE (SELECT F.WORKDEPT, AVG(F.SALARY)
FROM EMP F
WHERE F.WORKDEPT = E.WORKDEPT
GROUP BY WORKDEPT) DEPTSAL(WORKDEPT, AVGSAL)
ON E.WORKDEPT = DEPTSAL.WORKDEPT
```

The non-correlated table expression is great for larger volumes of data, especially when most or all of the data in the table expression will be used by the outer portion of the query. The correlated table expression is going to be better for transactions where very little data inside the correlated table expression will qualify. In the example above, the non-correlated version is better since all of the data is processed from both table references. If the predicate WHERE E.EMPNO = '000010' were added to each of the queries, then there is a very good chance that the correlated version would be the better performer since only a portion of data from each of the table references would be accessed. In contrast, the non-correlated version would still process the entire table expression before filtering. Since the results of the non-correlated subquery can be materialized into a workfile, you could run into workfile resource limitations. This could make the correlated table expression a better choice in those situations in regards to getting the statement to run, but the CPU usage could be higher. Keep in mind that, due to the top-bottom-top-bottom processing of the correlated nested table expression, the cost of such a statement can be extremely high for large volumes of data qualifying for the outer query. In these cases, the non-correlated version is definitely the better choice.

If you are coding scalar fullselects in a SELECT clause be aware of the fact that, regardless of whether or not it is correlated or non-correlated, it will be executed once for every row processed by that SELECT clause (see Section 2 for details).

Here are some general guidelines for the coding of nested table expressions:

- Use non-correlated if the NTE will process all of the data from the tables in the NTE
- Use correlated if the NTE will process only a fraction of the data from the tables in the NTE
- An available index on the correlated NTE query column(s) in the correlated predicate is critical

Be very aware of the concept of merge versus materialization discussed in the table expression section of Section 2 of this book and also the options available for improved performance of nested table expressions.

SQL Features and Performance

There is a very robust set of database features available on DB2 for z/OS. There are a lot of good reasons to utilize these features, and many of those reasons have to do with performance. Remember that you are working with data in the database, and if you have processes that are data intensive those processes should be executing close to the data. Most of the features discussed here can be utilized to save processing time and CPU cost. As always, testing a solution is the ultimate answer as to whether or not it will perform.

Stored Procedures

There are several reasons to use stored procedures, but the number one reason is to improve performance of transactions that issue multiple SQL statements over a network. From a performance perspective the main objective when implementing a stored procedure is to replace remote SQL calls with a single call to a stored procedure. The stored procedure would contain the multiple SQL statements and some business logic, and would return a result back to the caller. If you are writing single statement stored procedures, then you are hurting more than helping performance by doing nothing more than increasing the code path required to execute the statement.

There are three types of stored procedures:

- External
- External SQL
- Native SQL

External stored procedures are written in one of several programming languages including COBOL, Assembler, and Java, and execute in a workload managed application environment (WLM) stored procedure address space (SPAS). This SPAS is external to the DB2 address spaces. Therefore, any calls to these external stored procedures are cross-memory calls. IBM has optimized calls to these external routines, but the overhead of calling them has not been completely eliminated. This makes native SQL procedures a more attractive choice, since they execute inside the DB2 engine. However, external stored procedures do serve a strong purpose in that you can easily convert legacy programs into stored procedures and thus make these programs available to remote SQL calls. Also, if you have the need for a stored procedure that includes intensive non-database related processing it may be more efficient to write this stored procedure in a high performance language such as COBOL or Assembler as opposed to a native SQL procedure. Also, there are certain things, such as calling external services and directly accessing mainframe resources, which can only be accomplished in an external procedure. While I have found that the flexibility and portability of Java as a stored procedure programming language is extremely important and powerful, the expected CPU cost of running these procedures on DB2 for z/OS is about twice that of COBOL.

External stored procedures that execute in a WLM managed application environment are not zIIP eligible, even if they are called remotely.

SQL procedures can be either native or external. If they are created as an external SQL procedure, they are converted into C programs, compiled, and installed as if they were a normal external stored procedure. From a performance perspective, they are probably going to have a level of performance slightly less than the equivalent COBOL or Assembler programs. So, the advantage is in the flexibility and the portability of the SQL procedure, as well as the ability to draw on a comparatively more available talent pool of technicians. SQL external procedures exist primarily for compatibility with older releases of DB2 where there were no native SQL procedures. From a performance perspective, native SQL procedures are the better choice. Native SQL procedures execute internal to the DB2 engine, and can utilize zIIP engines for remote requests. While a native stored procedure may not outperform an external procedure that is non-SQL process intensive, the elimination of any cross memory call will offset some of that cost. For procedures that are primarily SQL statements, the native SQL procedure is going to be the best performance choice.

Triggers

Triggers are SQL based processes that are attached to tables and activated in response to DML statements. Triggers exist as a means of controlling and guaranteeing certain processing happens in a centralized fashion when DML is issued against a table. This includes such functionality as:

- Data validation
- Data replication and summarization
- Automated population of column values
- Stored procedure and UDF invocation
- Messaging
- Advanced referential integrity enforcement

Triggers can be abused, but they are generally a good thing from a performance perspective. If you think about what is being done in the body of a trigger and then compare that to the equivalent work if it was performed by an application program, you are again looking at a situation in which multiple SQL statements can be replaced by a single SQL statement and a trigger. The only time I can imagine triggers being a performance detriment would be if they were used to perform simple data validation that could be performed locally within an application, such as code validation or existence checking (for cached items). In that situation, there could be multiple messages going back and forth between the application and the database as successive validations failed. Otherwise, triggers are generally good from a performance perspective.

DB2 Built-In Functions and Expressions

DB2 comes bundled with a great variety of built-in scalar functions. This allows for incredible flexibility of SQL as a programming language and allows for developers to transform simple data elements into virtually anything ready for display or further implementation. Some of the things I've generated using built-in scalar function include:

- Page 1 for customer invoices
- Web pages
- Complete XML documents
- Assembler DSECTs

The bottom line on scalar functions in a SELECT from a performance perspective is that they cost you more relative to the equivalent application logic. So, if scalar functions are used simply for data manipulation then you can expect them to always be more expensive than equivalent application code. You should be coding these sorts of things into statements for portability and flexibility, and not performance. The same is also true for various expressions, such as CAST and CASE expressions, when used purely for data manipulation.

If your use of functions and expressions in a SQL statement results in further filtering of data, then the use of those functions or expressions can be a performance benefit as compared to sending all of the data back to an application program and doing the filtering there.

When coding CASE expressions that contain multiple search conditions, make sure to code the search conditions in a sequence such that the conditions that should test positive most often are coded earlier in the expression as CASE takes an early out.

Aggregate functions can be a performance advantage when compared to sending all of the data back to an application program for aggregation. The DB2 for z/OS Managing Performance Guide contains recommendations regarding how to code for aggregate function performance.

User-Defined Functions

UDFs are a really cool feature allowing a user to extend the capabilities of the DB2 engine beyond what is available and built into the product. As an example, I was once involved with an application that had its own customized soundex function. This function was written in IBM Assembler language back in the 1970's and the developers had lost the source code. We were able to write a COBOL wrapper to call the Assembler program and install it all as a DB2 UDF. Suddenly our legacy soundex function was available to execute within the context of a normal SQL statement and to access remotely from the web. This represented a huge advantage for the enterprise by solving a very serious programming issue. The downside was that it was relatively expensive to execute the function in this manner. Keep in mind that a UDF is going to be executed for each reference in a SELECT list for every row processed. UDFs exist primarily for functionality and not performance. If performance is your primary concern and UDFs are only going to be used for data manipulation and not data filtering or aggregation, then they should be avoided in favor of the equivalent application code.

There are several types of UDFs:

- Inline SQL scalar functions
- Non-line SQL scalar functions (discussed in the next section)
- External scalar functions
- External table functions

Inline scalar UDFs enable DBAs and developers to place commonly executed SQL expressions into a function stored in the DB2 system catalog. These functions are then available to use enterprise-wide. This enables a higher level of accuracy in function execution and a centralized control such that if the function changes it only needs to change in one place. The downside is decreased performance in that parameters passed to the function will be cast to the data types of the variables used in the function regardless of whether or not the data type needs to be cast. This can result in up to 10% overhead for the invocation of an inline SQL function as compared to the same expression coded directly in a statement.

External scalar and table functions behave in a similar manner to external stored procedures from a performance perspective. They are written in a host language and are executed in a WLM managed application environment SPAS. As such, there is similar amount of cross memory overhead as with an external stored procedure. However, given the nature of scalar UDFs executed in a SELECT clause and table UDFs executed for blocks of rows returned, there can be hundreds, thousands, or millions of external function calls per execution of the SQL statement that invokes them. So again, these functions are important for flexibility and reusability of new and existing program logic and are not necessarily a performance feature.

Non-Inline SQL Scalar Functions and SQL Table Functions

Non-inline SQL scalar functions allow for common SQL and business logic functionality to be incorporated into a scalar UDF. They can contain SQL procedure language (SQLPL) control statements similar to a native SQL procedure. These functions have a lot of the same performance advantages as native SQL stored procedures. However, just as with every UDF they could be executed many times in a SQL statement even if the results are redundant. Therefore, their use should be carefully considered with the understanding that better performance is probably possible by using a native SQL procedure. The big advantage with these UDFs is functionality, flexibility, and control, and not so much performance.

SQL table functions allow for a table expression to be embedded in a function that returns a table to the caller. They do have some promise as a performance feature, but come with some challenges. The execution of these functions is very similar to how correlated nested table expressions are executed. In fact, I would be as bold as to compare the execution of views and SQL table functions to that of nested table expressions and correlated nested table expressions, equating views to nested table expressions and SQL table functions to correlated nested table expressions. My nickname for SQL table functions is “parameterized views”, as parameters can be passed into the table function and used within the context of the statement embedded in the function. As of the writing of this handbook, parameters passed to SQL table functions and used in predicates within the function may be Stage 1 or Stage 2 predicates. The best thing to do is to create the function, write a SQL statement against the function, and EXPLAIN it. See APARs PM82908 and PM95541 for details. Here is an example of a SQL table function that can be used as a parameterized view:

```
CREATE FUNCTION DANL.EMP1 (INDEPT CHAR(3))
RETURNS TABLE (WORKDEPT CHAR(3), AVGSAL DEC(9,2))
RETURN
SELECT F.WORKDEPT, AVG(F.SALARY)
FROM DANL.EMP F
WHERE F.WORKDEPT = INDEPT
GROUP BY WORKDEPT;
```

Multi-Row Statements

Multi-row FETCH, INSERT, DELETE, and MERGE exist for one reason: performance. The purpose of multi-row operations is to reduce the number of SQL calls to complete a batch operation. Remote SQL calls that take advantage of block fetching using non-ambiguous, read-only cursors will automatically get multi-row FETCH between the DB2 distributed address space and the DB2 database manager

address space. Unfortunately, multi-row operations beyond the automatic FETCH for remote applications are only available to locally executing applications written in COBOL, Assembler, or C programming languages. Multi-row statements are not available to Java programs.

I have benchmarked COBOL programs that use multi-row FETCH and discovered up to 60% reduction in CPU when comparing single row FETCH to multi-row FETCH for a sequential batch cursor that retrieved 50 rows per FETCH, and a 25% reduction in CPU for an online cursor that averaged 20 rows per FETCH.

Select From a Table Expression

As mentioned earlier in this section, table expressions are a powerful construct that allow you to replace multiple SQL statements with a single multi-function statement. The concept is that the result of a SQL statement is a table, and if the result is a table, then that table can be input to another statement. Tables in and tables out! This concept is true for data change statements as well, including INSERT, UPDATE, DELETE, and MERGE. The result table from a data change statement is called a data-change-table-reference and this table can be referenced in the FROM clause of a fullselect. The keywords FINAL TABLE and OLD TABLE are used to indicate whether the view of the data affected by the data change statement is before or after the change has been applied. Being able to return data from a data change statement means multiple SQL calls can be combined for significant CPU and elapsed time savings. The following example selects data that has been added to a table via an INSERT:

```
SELECT EMPNO, FIRSTNME, LASTNAME
FROM   FINAL TABLE (
        INSERT INTO EMP2 (FIRSTNME, LASTNAME)
        SELECT FIRSTNME, LASTNAME
        FROM   EMP
        WHERE  WORKDEPT = 'C01' )
```

So, in a single statement a table is read, filtered and inserted into a second table. The results of that insert, including the system generated column EMPNO and identity column, are returned in the final SELECT. With three operations in one SQL statement the potential cost savings is quite obvious.

The power and potential cost savings can be taken even further through the use of INCLUDE columns within the data change table. These INCLUDE columns allow you to generate new data within the context of the data change statement that can then be referenced in the outer SELECT. These constructs allow for an even greater potential to replace several SQL statements and program logic with a single SQL statement. The following gives employees in department C01 a raise and reports on that raise in a single SQL statement, including a generated, old salary value.

```
SELECT EMPNO, OLD_SALARY, SALARY
FROM   FINAL TABLE (
        UPDATE EMP
        INCLUDE (OLD_SALARY DEC(9,2))
        SET SALARY = SALARY * 1.05
        ,OLD_SALARY = SALARY
        WHERE WORKDEPT = 'C01' )
```

As far as I can tell from multiple tests and implementations, there are no downsides to these types of statements, and only a benefit from a reduction in SQL calls within a transaction.

Common Table Expressions

Common table expressions are temporary tables that exist within the execution of a single SQL statement. There is a maximum of 225 tables that can be referenced in one statement. So, you could theoretically have 224 common table expressions in a statement. This allows you to take what would otherwise be a multi-step, multi-statement process and condense it all into one SQL statement execution. This is an obvious performance enhancement when multiple SQL calls can be combined into a single statement. Your performance will vary, and it's always a good idea to test these statements. The following example shows how multiple common table expressions can be specified in a single statement to process and reprocess data in a single complex operation.

```
WITH DEPTSAL (DEPTNO, AVGSAL) AS
  (SELECT EMP.WORKDEPT,AVG(EMP.SALARY)
   FROM EMP EMP GROUP BY EMP.WORKDEPT)
,HIGHLOWSAL (HIGHAVG, LOWAVG) AS
  (SELECT MAX(AVGSAL), MIN(AVGSAL)
   FROM DEPTSAL)
SELECT DEPTNO, AVGSAL FROM DEPTSAL
WHERE AVGSAL = (SELECT HIGHAVG
                FROM HIGHLOWSAL)
ORDER BY DEPTNO
```

The section joins in this section contain an example of a common table expression used in relational division, and Section 6 contains an example of a common table expression used in benchmark testing.

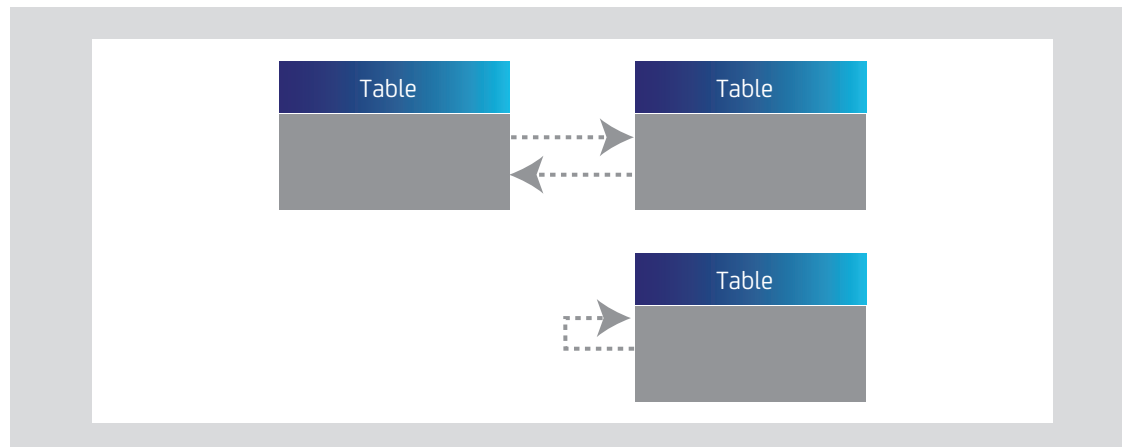
A common table expression can be merged with the subselect that references it, as is described in Section 2 of this book, but if a common table expression is referenced more than once it will be materialized. Common table expressions are key to constructing recursive SQL, which can prove to be a performance improvement in certain situations and are discussed in the next section of this section.

In addition to being useful in eliminating multiple database calls, common table expressions can prove to be a potential performance improvement for complex statements where the same complicated table expression may need to be referenced multiple times in the same statement. By taking that one complicated table expression, placing it in a common table expression, and then referencing the common table expression in place of the multiple complex table expression, the common table expression will be materialized and the subsequent table scans may prove to be more efficient than running the complex table expression multiple times.

Recursive SQL

When the FROM clause of a common table expression references itself, it is known as a recursive common table expression, or recursive SQL. A recursive SQL statement can be a tremendous performance advantage when it is used to replace multiple SQL statements and application logic with a single SQL call. This saving is most significant when the tables accessed are in a circular relationship, where two tables reference each other or a single table references itself as in the following figure.

Figure B.



Recursive SQL is perfect for traversing these circular references in a single statement, as opposed to doing the same thing within a program loop. This is perfect for traversing hierarchies and networks in such things as bill of material application, scheduling, banking, and web applications. In the following example, a web site menu is stored within a hierarchy in a single table. The web site menu is a hierarchy.

- A “PRODUCTS” menu item may have sub-items
 - “MEN’S CLOTHING”
 - “SHIRTS”
 - “SUITS”
 - “PANTS”
 - “WOMEN’S CLOTHING”
 - “DRESSES”
 - “BLOUSES”
 - “SHOES”
 - “CHILDREN’S CLOTHING”
 - “BOYS”
 - “GIRLS”

The single table that contains the web menu content contains a NODE column that uniquely identifies a position in the menu, and a PARENT_NODE column that identifies the parent node of a particular node. So, the “SHIRTS” menu item would have a parent node of the “MEN’S CLOTHING” menu item. Assuming that the main node, “PRODUCTS” is node zero, a recursive SQL statement can traverse this hierarchy and produce the web menu in a single statement, replacing what would otherwise be a loop in an application program with multiple SQL statements. This could result in a significant performance advantage (and did in a real life implementation).

```

WITH GEN_MENU(NODE, PARENT_NODE, NODE_NAME, NODE_DESC, N) AS
(SELECT NODE, PARENT_NODE, NODE_NAME, NODE_DESC, 1
 FROM   MENU_ITEM
 WHERE  NODE = 0
 AND    ACTIVE_FLG = 'Y'
 UNION ALL
 SELECT B.NODE, B.PARENT_NODE, B.NODE_NAME, B.NODE_DESC, N+1
 FROM   MENU_ITEM B , GEN_MENU A
 WHERE  B.PARENT_NODE = A.NODE
 AND    N < 20
 AND    ACTIVE_FLG = 'Y')
SELECT NODE, PARENT_NODE, NODE_NAME, NODE_DESC, N
FROM GEN_MENU

```

Recursive SQL is also useful in generating data and statements to be used in testing and benchmarking statements and table designs. See Section 9 of this book for details. This ability to generate data is also useful in production implementations, as in the following example where a recursive SQL statement is used to generate 100 sequence values in a batch application and avoid 100 separate statements.

```

WITH GEN_KEYS(N) AS
(SELECT 1 FROM SYSIBM.SYSDUMMY1
 UNION ALL
 SELECT N+1 FROM GEN_KEYS
 WHERE N < 100)
SELECT NEXT VALUE FOR SEQ1
FROM GEN_KEYS

```

Recursive SQL can potentially introduce an infinite loop, so it is very important to code a stop predicate in the recursive portion of the statement. This is typically done by introducing a counter and testing for a maximum value of the counter as in each of the previous examples.

BIND Options and Performance

Certain BIND options will promote a higher level of performance.

- **RELEASE(DEALLOCATE).** Setting RELEASE to COMMIT tells DB2 that when a thread issues a COMMIT statement the thread will go away and not come back. This is good for concurrency in that various locks will be released and open cursors closed. It is bad for performance if the application continues to issue SQL statements. That's because all resources that were released have to be reacquired. During execution certain resources are obtained and held including: xPROCS (fast code paths for common statements), sequential detection, index lookaside, and object locks. If RELEASE(DEALLOCATE) is used, these performance enhancers will be retained across a COMMIT and significant CPU savings will be realized. RELEASE(DEALLOCATE) will limit the concurrency of your high performance applications with DB2 commands and certain utilities. Therefore, an appropriate available plan should be in place when using RELEASE(DEALLOCATE) and appropriate maintenance

windows established. DB2 11 for z/OS now allows for commands and utilities to break into threads to force a maintenance process. Also, high performance DBATs that enable RELEASE(DEALLOCATE) for dynamic remote transactions can be turned on and off.

- **CACHESIZE.** Setting the CACHESIZE such that all authorization ids that use a package can be held in memory will avoid catalog lookups for incoming authorization ids.
- **CURRENTDATA.** The default is NO, and allows DB2 to use lock avoidance. This default is appropriate and should be used to avoid the overhead of taking read locks instead of latches as well as to enable block fetching for remote cursors.
- **DEFER(PREPARE).** This setting allows for dynamic SQL programs to combine PREPAREs of statements with the invocation of the statement and therefore reduce the number of messages issued by remote applications.
- **DEGREE.** This setting controls the degree of parallelism. Set this to 1, the default, for transactions that process little or no data per statement. Parallelism is an important feature for query elapsed time performance of large queries, but can be a performance detriment for transaction processing systems.
- **IMMEDIATEWRITE.** The default is NO and should stay no, except for extreme situations. Setting this to YES forces all writes to DASD immediately upon COMMIT. This could be important in some data sharing environments where readers are reading with UR, but in most cases setting this to YES will simply slow the application down and increase CPU consumption.
- **KEEPDYNAMIC.** Setting this to YES will cache prepared dynamic statements in thread storage and keep those statements across a commit. It requires coordination with application development to enable this to work properly. The existence of the global dynamic statement cache and high performance DBATs reduce the need for KEEPDYNAMIC in my opinion.
- **REOPT.** This option can be used to invoke various forms of statement reoptimization at execution time based upon host variable values. Setting any runtime reoptimization may improve access paths for situations where host variables or parameter markers are in use and data values in the accessed tables are skewed, but statements incur overhead due to incremental binds that happen at run time. I've only had success with runtime reoptimization when isolating single statements in their own package.
- **VALIDATE.** This option controls when authorization to objects is checked. Setting this to BIND will use the package owner's authid for object access authorization and avoid a look up at run time. This is the performance choice.
- **CONCURRENTACCESSRESOLUTION.** This may be an interesting choice for performance, and I have only begun to test it with some promising results. By setting this option to USECURRENTLYCOMMITTED an application that attempts to read data that is locked by an insert or a delete operation will not wait for the lock to be released, but instead access the currently committed row, if one exists, and continue the operation. This can really change the access paradigm for an application and can be considered a significant performance feature, especially when coupled with an optimistic locking strategy. However, it only works for rows impacted by insert and delete operations, and doesn't work for updates. This makes its use limited unless you implement "updateless" operations for select applications, which means doing deletes and inserts instead of updates.

Please also see Section 5 of this book for more information on BIND options used to control SQL access paths.

Dynamic SQL and Performance

The standard for programming against a DB2 for z/OS database has been steadily making the move from traditional programming languages, such as COBOL issuing static embedded SQL locally, to remote web-based applications coded in Java and issuing dynamic SQL statements. Dynamic SQL in remote applications can achieve a level of performance that is almost as good as static embedded SQL in local applications if a few performance guidelines are followed.

Block Fetching

For SELECT statements that are read-only in nature, you should specify FOR FETCH ONLY or FOR READ ONLY at the end of the statement. This will encourage DB2 to utilize block fetching for these remote cursors. Block fetching will return multiple rows to the remote application in a single DRDA message, as well as enable multi-row fetch between the DB2 distributed address space and the DB2 database manager address space. If you have SELECT statements that you expect will return no more than one row, then specify OPTIMIZE FOR 1 ROW FETCH 1 ROW ONLY at the end of the statement to disable any block fetching. Please refer to the DB2 for z/OS Managing Performance Guide for more details on ensuring block fetch.

Utilizing Dynamic Statement Cache

Dynamic SQL statements can be cached either globally in the global statement cache or locally in thread storage via the KEEP DYNAMIC bind option. Doing so will have a profound impact on performance as statement prepares will not have to go through a complete compilation process if an equivalent compiled statement is already located in cache. The global dynamic statement cache is enabled via EDMSTMTC subsystem parameter, which indicates the size of the cache. Increasing the cache size can improve the statement hit rate in the global cache and improve query performance by replacing long prepares (full statement compile) with short prepares (reusing an already compiled statement). A local statement cache can be enabled via the KEEP DYNAMIC bind option for the package used by the dynamic SQL in the application. The MAXKEEPD subsystem parameter controls how many statements will be retained in the local statement cache.

Parameter Markers versus Embedded Literals

For transaction processing applications where data skew is not an issue, it is strongly advised to use parameter markers in the predicate in your SQL statements as opposed to literal values. This will allow DB2 to take advantage of the dynamic statement cache and reuse already prepared statements. In order to reuse statements in the global dynamic statement cache, DB2 attempts to match the incoming statement text byte for byte with previous statements issued by the same authorization id. If literal values are used and constantly changing, then the match rates for inbound statements will be very low and statement prepare costs will be higher.

If your data is highly skewed, you may want to embed literal values in your SQL statements in order to potentially get better access paths. However, this should be done in conjunction with the gathering of distribution statistics or histogram statistics as described in Section 2 of this book.

Enabling Literal Concentration

For applications that have already existing SQL statements using embedded literal values in predicates or for packaged applications that only issue SQL using embedded literals, DB2 provides a potential performance improvement by means of literal concentration. This feature is enabled via the `CONCENTRATE STATEMENTS WITH LITERALS` attribute of the `PREPARE` statement, or via the `statementConcentrator` connection property in a JDBC application. When literal concentration is enabled, DB2 will replace literal values in the SQL statements with parameter markers, thus enabling a higher level of dynamic statement cache reuse and a much lower `PREPARE` cost. DB2 will only concentrate literal values for statements that use 100% literal values. Statements that have a mixture of literal values and parameter markers are not eligible for literal concentration.

Utilization of High Performance DBATs

One of the challenges of remote application performance is in balancing high performance with thread reuse and availability. DBAs had to balance performance and availability requirements with memory consumption. IBM had to provide a balance in DB2 thread memory management, as this was a big challenge for large implementations through DB2 9. As of DB2 10 for z/OS, much of the memory constraints have been lifted and the number of active threads possible in a single DB2 subsystem has dramatically increased.

Remote threads have traditionally been pooled in the DB2 distributed address space. That is, many threads are connected to the DB2 distributed address space, but a fewer number of threads that are actually active are connected from the DB2 distributed address space to the DB2 database manager address space. These connections to the database manager address space are called Database Access Threads or DBATs. Once a thread issues a commit it is considered inactive and releases its associated DBAT. In that way, a pool of DBATs can be used by many connections. The number of connections, as well as the number of DBATs, is controlled by subsystem parameters `MAXDBAT` and `CONDBAT` that are described in the DB2 Installation Guide and in Section 7 of this book. Since DBATs are released and shared across multiple connections, thread reuse and certain DB2 performance enhancers are not available for remote connections. This also means that such things as sequential detection and index lookaside are not available for remote connections.

High performance DBATs were introduced in DB2 10 for z/OS and allow for remote DRDA connections to behave in a similar fashion to CICS protected threads, as they take advantage of thread reuse and the various performance enhancers. These performance enhancers include dynamic prefetch, index lookaside, and xPROCS (fast path code paths for repeated statements). These performance enhancers are traditionally enabled via the `RELEASE(DEALLOCATE)` bind parameter, but prior to DB2 10 this bind parameter was ignored for packages accessed via remote threads. DB2 10 introduced the command `MODIFY DDF PKGREL(BNDOPT)` which will now cause DB2 to respect the `RELEASE(DEALLOCATE)` parameter for your remote packages and enable all the performance enhancers available for reused threads. When implemented, high performance DBATs can have a profound impact on remote application performance and a significant reduction in CPU consumed by remote applications. Enabling `RELEASE(DEALLOCATE)` will reduce the concurrency of the applications with certain DB2 utilities and commands, much in the same way as CICS protected threads. This is why DB2 will terminate high performance DBATs after 200 executions. Also, by issuing the `MODIFY DDF PKGREL(COMMIT)` command, the high performance DBATs will be returned to normal DBATs, which will allow for utilities and commands to run without conflict.

Section 4:

Utilizing DB2 EXPLAIN

The DB2 EXPLAIN facility is a means for estimating SQL execution cost and for externalizing the transformed (rewritten) query and selected access path. It is one of two key techniques, with the other being benchmark tests, for predicting the performance of production queries and applications. Using EXPLAIN should be a part of every SQL development and testing process. EXPLAIN can externalize the following information:

- Access path
- Transformed query text
- Filter factors
- Predicted cardinalities
- Missing or conflicting statistics
- Statement cost information

EXPLAIN Processing

There are three different ways to invoke an EXPLAIN of a SQL statement or group of SQL statements:

- EXPLAIN statement
- EXPLAIN attribute of a PREPARE statement
- EXPLAIN BIND parameter

The result of an EXPLAIN is the population of a set of explain tables that must pre-exist before EXPLAIN can be processed for a given userid. This userid is the authorization id of the process performing the EXPLAIN, or the owner of a package which is bound with EXPLAIN(YES) or EXPLAIN(ONLY). The set of explain tables must exist with a creator of the userid. As of DB2 11 for z/OS there are 20 EXPLAIN tables that are used to interpret the predicted query performance or to control the access path selected by the optimizer.

Interpreting EXPLAIN Results

Given that there are 20 different EXPLAIN tables and most of the information contained in the tables is difficult to interpret, it is strongly advised that you utilize a tool to interpret the data in the EXPLAIN tables. There are many tools that can be used to interpret EXPLAIN⁸ table data, and one of those tools, the IBM Data Studio, is available free. This tool, however, is PC-based and requires a remote connection to the DB2 data server on which you are to interpret EXPLAIN table data.

The PLAN_TABLE

The main EXPLAIN table is the PLAN_TABLE. This table contains the basic access path information for a given query and can be used quite effectively on its own to interpret an access path. You can query the PLAN_TABLE directly in order to interpret access path information for a given query or set of queries. The PLAN_TABLE contains information about whether or not an index is used to access a

table, the number of matching columns, the join method, and whether or not sorts and workfiles are utilized (as described in Section 2 of this book). The description of the columns of the PLAN_TABLE, along with all of the other EXPLAIN tables, is located in the DB2 Performance Guide. The output for a particular statement access path recorded in the PLAN_TABLE can be identified by the columns COLLID, PROGNAME, QUERYNO, and EXPLAIN_TIME. If the results are ordered by QBLOCKNO and PLANNO, this will yield access path information for a particular query within a collection and package, for the time of the EXPLAIN, and ordered in the query block and plan step within each query block. In the following example, it's a single statement EXPLAINED from within SPUFI.

```
EXPLAIN PLAN SET QUERYNO=999 FOR
  SELECT *
  FROM   DANL.EMP
  WHERE  WORKDEPT IN
        (SELECT WORKDEPT
         FROM   DANL.EMP
         GROUP BY WORKDEPT
        );
COMMIT;
SELECT * FROM PLAN_TABLE
WHERE  COLLID   = 'DSNESPCS'
AND    PROGNAME = 'DSNESM68'
AND    QUERYNO = 999
ORDER BY QBLOCKNO, PLANNO;
```

The following is an example of executing an EXPLAIN statement dynamically for a single query and then using a query to read the PLAN_TABLE data for the explained statement. Some formatting is also applied that focuses on some of the most important aspects of the access path. I will typically run multiple EXPLAINS for a given query with variations to indexes or the query text in order to determine variations in the access path.

```
EXPLAIN PLAN SET QUERYNO=999 FOR
  SELECT *
  FROM   DANL.EMP
  WHERE  WORKDEPT IN
        (SELECT WORKDEPT
         FROM   DANL.EMP
         GROUP BY WORKDEPT
        );
COMMIT;
SELECT SUBSTR(P.PROGNAME, 1, 8) AS PROGNAME
      ,SUBSTR(DIGITS(P.QUERYNO), 6) CONCAT '-'
      CONCAT SUBSTR(DIGITS(P.QBLOCKNO), 4)
      CONCAT '-' CONCAT
      SUBSTR(DIGITS(P.PLANNO), 4) AS QQP
      ,SUBSTR(CHAR(P.METHOD), 1, 3) AS MTH
```

```

        ,SUBSTR(CHAR(P.MERGE_JOIN_COLS), 1, 3) AS MJC
        ,SUBSTR(P.CREATOR, 1, 8) AS TBCREATOR
        ,SUBSTR(P.TNAME, 1, 18) AS TBNAME
        ,SUBSTR(P.CORRELATION_NAME, 1, 8) AS CORR_NM
        ,P.ACCESTYPE AS ATYP
        ,SUBSTR(P.ACCESSNAME, 1, 15) AS A_NM
        ,P.INDEXONLY AS IXO
        ,CHAR(P.MIXOPSEQ) AS MIX
        ,CHAR(P.MATCHCOLS) MCOL
        ,P.SORTN_JOIN CONCAT P.SORTC_UNIQ CONCAT
        P.SORTC_JOIN CONCAT P.SORTC_ORDERBY CONCAT
        P.SORTC_GROUPBY AS NJ_CUJOG
        ,P.PREFETCH AS PF
        ,P.COLUMN_FN_EVAL AS CFE
        ,P.PAGE_RANGE AS PGRNG
        ,P.JOIN_TYPE AS JT
        ,P.QBLOCK_TYPE AS QB_TYP
        ,P.PARENT_QBLOCKNO AS P_QB
        ,P.TABLE_TYPE AS TB_TYP
        ,P. EXPLAIN_TIME AS B_TM
FROM    PLAN_TABLE P
WHERE   P.QUERYNO = 999
ORDER  BY PROGNAME, B_TM, QQP, MIX;

```

Advanced EXPLAIN Analysis

If you do not have access to an EXPLAIN tool, you can still get some advanced EXPLAIN information from some of the EXPLAIN tables beyond what you can get from the PLAN_TABLE alone. While each EXPLAIN table can be read independently for analysis of a particular query, I do typically ignore many of them due to complexity and/or proprietary information contained in the tables. However, some of the tables do contain some information that is not too difficult to interpret, and can be queried individually for a specific query and EXPLAIN_TIME.

▪ DSN_FILTER_TABLE

This table contains information about how predicates are used in a query. This includes the predicate number, the order number of the predicate that reflects the order of predicate processing, the stage the predicate was processed (MATCHING, SCREENING, STAGE1, STAGE2), and the PUSHDOWN column that reflects whether or not a Stage 2 predicate was actually processed from Stage 1 index or data processing. You can use the information from this table together with the DSN_PREDICATE_TABLE to identify the predicate text along with the information in this table by matching the PROGNAME, QUERYNO, QBLOCKNO, PREDNO, and EXPLAIN_TIME columns of this table to the DSN_PREDICAT_TABLE.

- **DSN_PREDICAT_TABLE**

This table contains information about all the predicates in a query. This includes the predicate number, whether or not it is a Boolean term, whether or not the predicate was added to the query during query transformation, the filter factor for the predicate, and the predicate text itself. This is really useful for seeing information about predicates generated via predicate transitive closure or for temporal or archival time travel access. You can use this information together with the DSN_FILTER_TABLE by matching the PROGNAME, QUERYNO, QBLOCKNO, PREDNO, and EXPLAIN_TIME columns of this table to the DSN_FILTER_TABLE.

- **DSN_QUERY_TABLE**

This table contains the formatted original and transformed queries. You can read the query text from the table in order to determine the level to which the query was rewritten by the query transformation component before being sent to access path select. While querying this table is as simple as selecting the NODE_DATA column to get the query text for a TYPE of either “BEFORE” or “AFTER”, the result is a bit difficult to read due to the fact that the result is in XML meant to be interpreted by a tool. Nonetheless, it can be useful when a tool is not available to get some information about the transformed query.

- **DSN_DETCOST_TABLE**

There is a lot of valuable cost information in this table and it's really important for understanding the intricacies of the cost calculations at all levels of an access path. However, without guidance from IBM or a very careful process of trial and error EXPLAINS to understand the subtleties of the data contained in this table, you best leave this one alone. I do, however, pay attention to the ONECOMPROWS column as that contains the number of rows expected to qualify for a table after all local predicates are applied. This can be an important tool in understanding why DB2 picked a certain table access sequence for a multi-table query. There can be multiple rows for a given query block (QBLOCKNO) and plan step (PLANNO) so I always return the lowest ONECOMPROWS for a given PROGNAME, QUERYNO, QBLOCKNO, PLANNO, and EXPLAIN_TIME.

The following is my favorite advanced EXPLAIN query that returns the valuable PLAN_TABLE access path information along with filter factor, predicate stage, Boolean term, and predicate text predicate information from DSN_FILTER_TABLE and DSN_PREDICAT_TABLE, and the number of rows expected after local filters for a plan step from the DSN_DETCOST_TABLE.

```

EXPLAIN PLAN SET QUERYNO=1 FOR
  SELECT *
  FROM   DANL.EMP
  WHERE  WORKDEPT IN
        (SELECT WORKDEPT
         FROM   DANL.EMP
         GROUP BY WORKDEPT
        )
;
WITH MAXTIME (COLLID, PROGNAME, QUERYNO, EXPLAIN_TIME) AS
  (SELECT COLLID, PROGNAME, QUERYNO, MAX(EXPLAIN_TIME)
   FROM   PLAN_TABLE
   WHERE  COLLID   = 'DSNESPCS'
   AND    PROGNAME = 'DSNESM68'
   AND    QUERYNO = 1
   GROUP BY COLLID, PROGNAME, QUERYNO)

SELECT SUBSTR(P.PROGNAME, 1, 8) AS PROGNAME
      ,SUBSTR(DIGITS(P.QUERYNO), 6) CONCAT '-'
      CONCAT SUBSTR(DIGITS(P.QBLOCKNO), 4)
      CONCAT '-' CONCAT
      SUBSTR(DIGITS(P.PLANNO), 4) AS QQP
      ,SUBSTR(CHAR(P.METHOD), 1, 3) AS MTH
      ,SUBSTR(CHAR(F.PREDNO), 1, 3) AS P_NO
      ,SUBSTR(CHAR(P.MERGE_JOIN_COLS), 1, 3) AS MJC
      ,SUBSTR(P.CREATOR, 1, 8) AS TBCreator
      ,SUBSTR(P.TNAME, 1, 18) AS TBNAME
      ,SUBSTR(P.CORRELATION_NAME, 1, 8) AS CORR_NM
      ,DEC(D.ONECOMPROWS, 10, 1) AS ROWS_POST_FILTER
      ,P.ACCESSSTYPE AS ATYP
      ,SUBSTR(P.ACCESSNAME, 1, 15) AS A_NM
      ,P.INDEXONLY AS IXO
      ,CHAR(P.MIXOPSEQ) AS MIX
      ,CHAR(P.MATCHCOLS) MCOL
      ,F.STAGE AS STAGE
      ,DEC(E.FILTER_FACTOR, 11, 10) AS FF
      ,E.BOOLEAN_TERM AS BT
      ,SUBSTR(E.TEXT, 1, 30) AS PRED_TEXT30
      ,P.SORTN_JOIN CONCAT P.SORTC_UNIQ CONCAT
      P.SORTC_JOIN CONCAT P.SORTC_ORDERBY CONCAT
      P.SORTC_GROUPBY AS NJ_CUJOG
      ,P.PREFETCH AS PF
      ,P.COLUMN_FN_EVAL AS CFE

```

```

        ,P.PAGE_RANGE AS PGRNG
        ,P.JOIN_TYPE AS JT
        ,P.QBLOCK_TYPE AS QB_TYP
        ,P.PARENT_QBLOCKNO AS P_QB
        ,P.TABLE_TYPE AS TB_TYP
        ,P.EXPLAIN_TIME AS B_TM
FROM    PLAN_TABLE P
INNER JOIN
        MAXTIME      M
ON      M.COLLID      = P.COLLID
AND     M.PROGNAME     = P.PROGNAME
AND     M.QUERYNO     = P.QUERYNO
AND     M. EXPLAIN_TIME = P. EXPLAIN_TIME
LEFT JOIN
        DSN_FILTER_TABLE F
ON      M.COLLID      = F.COLLID
AND     M.PROGNAME     = F.PROGNAME
AND     M.QUERYNO     = F.QUERYNO
AND     P.QBLOCKNO    = F.QBLOCKNO
AND     P.PLANNO      = F.PLANNO
AND     M. EXPLAIN_TIME = F.EXPLAIN_TIME
and     P.ACCESTYPE Not IN ('MX', 'MI', 'MU')
LEFT JOIN
        DSN_PREDICAT_TABLE E
ON      F.PROGNAME     = E.PROGNAME
AND     F.QUERYNO     = E.QUERYNO
AND     F.QBLOCKNO    = E.QBLOCKNO
AND     F.PREDNO      = E.PREDNO
AND     M. EXPLAIN_TIME = E.EXPLAIN_TIME
LEFT JOIN
TABLE    (SELECT MIN(X.ONECOMPROWS) AS ONECOMPROWS
        FROM    DSN_DETCOST_TABLE X
        WHERE   M.PROGNAME = X.PROGNAME
        AND     M.QUERYNO = X.QUERYNO
        AND     P.QBLOCKNO = X.QBLOCKNO
        AND     P.PLANNO = X.PLANNO
        AND     M. EXPLAIN_TIME = X.EXPLAIN_TIME) AS D
ON 1=1v
ORDER BY PROGNAME, B_TM, QQP, MIX, F.PREDNO;

```

DSN_STATEMNT_TABLE

This table contains information about the estimated cost of a SQL statement. It has been used for a number of years by many people to compare the cost of one query versus another when query tuning is performed. Some people even have shop standards that require a minimum query cost proven via an EXPLAIN before a query can be migrated to a production environment. Others analyze this table to look for queries that exceed a predetermined cost value. While these techniques may be effective in determining relative query costs in many situations, it can be perilous since the cost figures are an estimate of query cost based upon optimizer calculations. I personally never rely on the cost figures in this table alone as proof of the cost of one query versus another. If I do use the values in this table, it is done in conjunction with the access path information provided by the PLAN_TABLE and other EXPLAIN tables. Even so, I rely much more on performance benchmarks than query cost estimates to determine the predicted query cost once a query is promoted to production. Please see Section 9 of this book for information on how to perform performance benchmarks.

Externalizing the Dynamic Statement Cache via EXPLAIN

With an ever increasing workload utilizing dynamic SQL, it has become harder for DBAs and application developers to collect and report on SQL statements since the statement text is no longer stored in the DB2 system catalog as it is with static embedded SQL programs. We can use the DB2 EXPLAIN facility to collect the text of dynamic SQL executing in a DB2 subsystem, as well as optionally, collect some performance metrics for the dynamic SQL and EXPLAIN the dynamic SQL.

The EXPLAIN STMTCACHE ALL SQL statement will externalize the dynamic SQL statements in a DB2 subsystem to a table called DSN_STATEMENT_CACHE_TABLE for the authorization id of the process executing the EXPLAIN STMTCACHE ALL. If the authorization id of the process has SYSADM or SQLADM authority, then all of the statements in the cache will be exposed; otherwise only the statements for the authorization id will be exposed. Once the EXPLAIN STMTCACHE ALL statement is executed the DSN_STATEMENT_CACHE_TABLE table for the authorization id can then be queried to retrieve the statement id, statement token, and statement text. If actual EXPLAIN output is desired, an EXPLAIN STMTCACHE STMTID or EXPLAIN STMTCACHE STMTOKEN can be issued and will result in the population of all of the EXPLAIN information for the identified statement in the same way as for a normal EXPLAIN execution. Keep in mind, that EXPLAIN STMTCACHE ALL only collects statement text and optional statement metrics and populates the DSN_STATEMENT_CACHE_TABLE table.

If IFCIDs 316 and 318 are active, the DSN_STATEMENT_CACHE_TABLE is populated with a significant quantity of extremely valuable performance metrics for individual dynamic SQL statements executing in a subsystem. These metrics include such things as:

- Number of executions
- Getpages
- Rows processed
- Rows read
- Table space scans
- Sorts

- Elapsed time
- CPU time
- Locks
- The statement text

and many more. The DB2 Performance Guide Appendix B contains a description of all the columns. One of the most valuable things to get from the DSN_STATEMENT_CACHE table is the total CPU consumed by a SQL statement since it was cached and the performance metrics collected by the IFCIDs. The reason this is so valuable is because it shows the total CPU, which can be a factor of individual statement cost and/or frequency of execution, and is thus an extremely important tool in determining which statement or process to tune first. It also makes explaining the statement cache an important benchmarking tool. What follows is my favorite, although simple, query for determining which statements I need to gather more metrics on for possible analysis and tuning:

```
SELECT cached_ts, STAT_EXEC,  
       dec(stat_elap,12,2) as stat_elap,  
       dec(STAT_CPU,12,2) as stat_cpu,  
       substr(STMT_TEXT,1,50)  
from   ESPNLCP.DSN_STATEMENT_CACHE_TABLE  
where  primauth = '<authid>'  
and    explain_ts > '2013-10-04-00.00.00.000000'  
order by stat_cpu desc
```

Please see Section 8 for more information on turning on IFCID 316 and 318.

Section 5:

Controlling SQL Access Paths

The good news is that with every new version of DB2 the query processing becomes more complex and access path issues become rarer and rarer. The bad news is that with every new version of DB2 it becomes more difficult for a user to manipulate the statement and database in a manner that will influence DB2 to a more favorable access path when the chosen path is not desirable. The good news is that there are now many choices for using built-in features to control the access path of certain statements. The bad news is that the choices are confusing, overlapping, and in some cases complicated. Are you confused yet? It's OK, I'm confused as well.

In all seriousness, controlling access paths should, in the vast majority of cases be a non-issue. As long as the catalog statistics accurately represent the data, the data is well organized, and developers and DBAs have paid a modest amount of attention to input values, the percentage of SQL statements that need special attention should be few and far between. With that being said, stability, which is

completely different from tuning, is a major concern and should be controlled. DB2 provides many methods of ensuring and preserving SQL access paths that can be utilized to promote stability. These options will be the major focus of this section.

Parallelism

Parallelism is great for large report or warehouse queries, and not good at all for transactions that process little or no data. There are several subsystem parameters that control parallelism (see section 7) as well as a BIND parameter and special register (see Section 2).

Maintaining Access Paths

DB2 provides for, and is pretty much guaranteeing that, at least for static embedded SQL, you can ensure that an access path will not change when rebinding a package. As of DB2 11 for z/OS, this is true even when moving from one release of DB2 to the next. So, there is a choice as to whether or not an access path is retained at migration time, and that is certainly a nice conservative thing to do. However, I would advise at some point to allow DB2 to potentially find a new access path that takes advantage of new DB2 features. In addition, it is typical that certain performance features are disabled during the execution of packages bound in a prior release of DB2; therefore, rebinding packages on migration is always recommended whether or not you elect to retain access paths.

Access paths for static embedded SQL can be managed via a feature generally referred to as “access plan stability” or “plan management”. This feature⁹ is enabled via several BIND options. A combination of these BIND options pretty much gives full control over access path changes, or access path stability, across REBINDS. Please be mindful of the fact that the different options operate independent of each other, with the exception of APRETAINDUP which works together with PLANMGMT. Also please keep in mind that the PLANMGMT and APRETAINDUP options are only REBIND options while the other options pertaining to access path comparison and retention can work with BIND or REBIND! Many of the options can be combined, really allowing for a customization of your access path management for static embedded SQL. It is important to sit down, understand the options, run some tests, and then develop a strategy of using all or some of these options in your environment.

PLANMGMT, SWITCH, and APRETAINDUP

This option allows for the retention of access path information for previous binds of a package. There are three settings for this option that allow for no retention or the retention of up to two previous binds for the package. This is a rebind-only option that permits access paths to be saved. In many enterprises people have established a regular policy of the “three R’s” (REORG, RUNSTATS, and REBIND). I’m not a big fan of this as a generic strategy since an access path may go bad at any time and really put a dent in production performance (please see Section 2 for my thoughts on RUNSTATS). If you do employ a strategy that includes regular, unregulated rebinds, this option will let you back out all of the access paths that resulted from a REBIND of a package by using the SWITCH REBIND option. In that way, if an access path goes bad you can back out. The APRETAINDUP option is used together with PLANMGMT on a REBIND to tell DB2 what to do if you REBIND and there are no access path changes. This allows for a REBIND to not overlay a previous access path if there were no differences.

PLANMGMT is the better way to preserve access paths across DB2 migrations.

APCOMPARE

This option can be used on a BIND or REBIND and is used to control the result of the command if access paths are different from a previous BIND or REBIND. Please keep in mind that this option is independent from the other plan management and access path stability options. The way DB2 goes about searching for previous access paths is not simple and a review of this process as described in the DB2 Performance Guide is advised. The option allows for no action to be taken, a warning to be issued or the rebind process to be terminated if access paths are different from a previous bind or rebind of a statement in the package.

APPLCOMPAT

This option is new as of DB2 11 for z/OS and allows for the guarantee of access path stability across the migration of DB2 from DB2 10 to DB2 11 (and presumably going forward as well). One major hesitation in regards to version migration is the fear of access path regression. This option eliminates this fear for static embedded SQL, and applies to both BIND and REBIND. Please keep in mind that this option is independent from the other plan management and access path stability options. The way DB2 goes about searching for previous access paths is not simple and a review of this process as described in the DB2 Performance Guide is advised.

APREUSE

This is a pretty powerful BIND and REBIND option that tells DB2 to reuse existing access paths for recognized statements when possible, and then specifies what action to take if an access path from a previously existing statement cannot be reused. The choices are to simply perform the bind, fail to bind if paths cannot be reused, or bind all the statements and reuse paths when possible also issuing a warning message for the paths not reused (new for DB2 11 for z/OS). It should be noted that prior to DB2 11 APREUSE(ERROR) applied to the entire package. If any single statement failed then the entire package would fail. With DB2 11 APREUSE(WARN) is at the statement level. So only the statements that failed would revert back to original access path.

Optimization Hints

DB2 allows for the creation and use of optimization hints at the statement level and user level¹⁰. The use of optimization hints is available for static and dynamic statements and is available for a given user or globally. Using optimization hints is documented in the DB2 Performance Guide and DB2 Command Reference. They involve the use of user EXPLAIN tables and system catalog tables. You can reference the DB2 documentation for instructions on how to use hints. I'll only present the options here.

- Statement level optimization parameter hints. This allows you to specify certain optimization parameters that are to be considered, independent of system or application level defaults, based upon a particular statement text.
- Statement level access path hints. These are specified access paths for particular statements and the scope of the application is variable. The hints can be controlled globally or at a package and/or authorization id level.

- Statement level selectivity overrides. These are predicate level filter factor hints for individual statements. This is new for DB2 11 for z/OS and presents an opportunity to do predicate level influencing. This should eliminate the need for the various desperation tricks I mention in this section.
- PLAN_TABLE hints. These are the more traditional hints that allow you to specify an access path for a particular package and authorization id.

Hints rely on EXPLAIN tables that exist for a particular authorization id, or are globally based upon a set of DB2 system catalog EXPLAIN tables. Hints are activated via a BIND PACKAGE or BIND QUERY command, depending upon whether or not the hint is plan table level or global. There also exists an entire infrastructure for managing, testing, and moving hints to and from the DB2 system catalog tables.

Influencing the Optimizer

This section is going to talk specifically about manipulation in an effort to influence the optimizer, and should be used in context with everything I've talked about in this section, as well as in Sections 2, 3, 4, and 6 of this book. You are way better off trying to influence the optimizer using conventional techniques rather than trying to trick it into doing your bidding. What follows is a list of conventional techniques in my order of preference.

- Proper RUNSTATS (Section 2)
- Coding SQL for proper performance (Sections 2 and 3)
- Plan management (Section 5)
- Access plan stability (Section 5)
- Literal Concentration for dynamic SQL (Section 3)
- Runtime reoptimization (Section 3)
- Optimization hints (Section 5)

There are several less conventional techniques that people have used throughout the years, some of which are documented in the DB2 Performance Guide. However, as DB2 SQL optimization evolves more and more of these unconventional techniques become obsolete or even impair proper SQL optimization. For this reason you can expect to see the techniques that are documented in the DB2 Performance Guide to be removed from that documentation in the future. I mostly employ the practice of properly coding SQL for performance as is documented in Sections 2 and 3 of this book. This involves not just coding the SQL properly, but also understanding which coding techniques will offer the best level of performance for certain situations. From a simplistic point of view I consider whether or not the SQL is going to process very little data or a lot of data and make the coding decisions based upon that simple information. Of course I am always using EXPLAIN and always benchmarking every statement written. What follows is a bullet point list of some techniques I've used to influence the optimizer. Most of these are useful primarily for multi-table queries or for trying to convince DB2 to use one index versus another index. I consider most of these acts of desperation, unless otherwise specified in the description.

- **OPTIMIZE FOR.** You can plan the OPTIMIZE FOR clause at the end of a SQL statement in order to communicate to the optimizer exactly how many rows you intend to process from the SQL

statement. This is NOT the number of rows you expect the optimizer to process, but instead the number of rows you intend to process. The clause was created in response to the fact that some pagination queries might qualify a serious number of rows, but only display a page worth on a screen before exiting a FETCH loop. If you are going to use OPTIMIZE FOR then the general guideline is honestly specify the number of rows you intend to process for the query. If you are going to process the entire result set then do not specify the clause at all. Also be aware that by specifying OPTIMIZE FOR you may inaccurately influence the query cost estimate in an EXPLAIN, making it unusable in my opinion for statement cost comparison (it's an easy way to get a query to pass a cost analysis comparison based upon an inaccurate EXPLAIN only to have it cripple production with incredibly bad performance). This option also influences the block size used for block fetching for remote applications. So, abusing OPTIMIZE FOR can hurt performance as much or more than it can help performance. A common tuning choice for transactions that has been used quite extensively is OPTIMIZE FOR 1 ROW. This choice will discourage DB2 from considering sorts, sequential prefetch, and list prefetch for some SQL statements. The behavior of OPTIMIZE FOR 1 ROW changed in DB2 10 for z/OS. The change is described in APAR PM56845 and gives you the choice to modify the behavior of OPTIMIZE FOR 1 ROW and/or introduce OPTIMIZE FOR 2 ROWS into your tuning arsenal.

- **VOLATILE table option.** This option, as described in Section 6 of this book, will encourage DB2 to choose an access path that favors an index. I recommend this option for tables that are indeed volatile and have used it in these situations with great success. You could use it to influence access path as well for other tables, but it is an act of desperation and could hurt more than help.
- **Predicate enablers.** This type of technique is generally used to influence DB2 to pick an index over a table space scan, or to pick one index versus another index. You can modify indexes such that index-only access is possible or to increase the matching number of columns based upon existing predicates in a query. You can also attempt to add predicates to a query, including redundant predicates. This is especially helpful if the predicates use host variables or parameter markers rather than literal values. You can add a predicate that provides no actual real filtering for a query, but will increase the filter factor calculated for a table or the filter factor calculated for index access on a table. The filter factor may then influence the choice of one table versus another in a multi-table statement or the choice of one index versus another if it is perceived to increase filtering or matching. For example if an index contains COL1, COL2, COL3, and COL4 and your query has a compound predicate "WHERE COL1 = ? AND COL4 = ?" you could add another predicate such as "AND COL2 BETWEEN ? AND ?" to increase match columns and influence the access path if it's of potential benefit. The values provided for the parameter markers would represent the minimum and maximum column values.
- **Predicate disablers.** This technique is used to discourage DB2 from choosing a particular index or to modify the table access sequence in a multi-table query. It is based upon the concept that a Stage 2 predicate connected to a Stage 1 predicate with an OR makes the entire compound predicate Stage 2. For example, "COL1 = ?" is Stage 1 indexable, whereas "(COL1 = ? OR 0=1)" is Stage 2 and non-indexable, even though the right side of the OR is meaningless. This is a quite common technique that I have used three times in the last 18 years.
- **Using ORDER BY.** DB2 will consider an index if it can be used to avoid a sort. By coding an ORDER BY, specifying the leading columns of an index that is desired, you can possibly influence DB2 to use that index over other indexes or a table space scan, which is not so much an act of desperation.

- **Influence table access sequence.** I have spent a lot of time tuning multi-table statements and attempting to encourage DB2 to pick the tables in a sequence that I think is more efficient only after the statement appeared as a potential performance problem. I certainly don't view many of the choices here as an act of desperation, except for use of predicate enablers and disablers. The issues here are many due to the fact that DB2 does a great job of determining which table is accessed first based upon catalog statistics and the simple concept that the table that returns the fewest rows should be accessed first in a multi-table operation. One of the problems is something I like to call "one is the lowest number". To be specific, when estimating the rows to be returned from a table any number lower than one is rounded up to one. So, even though all the tables in a multi-table statement (think join) that are predicted to return one row are thrown to the beginning of the join operation, relative to each other they are considered the same in that they return one row. What if the operation includes inner joins that could result in rows being eliminated by a table that will more often return zero rows than one row? I'm going to want that table accessed before the tables that actually return one row. In general, using these techniques I'm trying to encourage DB2 to pick first the tables that I know will return fewer rows even though, for some reason, DB2 does not know. This could involve the whole one row thing, some data skew without distribution statistics, parameter markers and host variables, or the fact that I know the input to the query and DB2 does not (again, parameter markers and host variables).
- **Toggle between joins, correlated, and non-correlated subqueries when possible.** Look at the columns being returned. Are they coming from all the tables? If not, you have options for coding the statement as either of the three, including anti-joins. See Section 3 for the details.
- **Increase indexability of the table you want accessed first.** This could be one or more of a combination of things including adding columns to the index of the table you want accessed first, or the use of predicate enablers or disablers.
- **Use predicate enablers or disablers.** The same technique to encourage the use of one index over another can also influence table access sequence.
- **Code an ORDER BY on the columns from the table you want accessed first.** DB2 may consider the index to avoid a sort, and DB2 can use an index to avoid a sort in a join when all of the columns of the sort come from only one table and that table is accessed first in the join sequence.
- **Use a correlated nested table expression.** I actually do this quite often, even though with each version of DB2 it does become slightly less effective. DB2 will have limited access paths to choose from if you use correlation and will almost always use nested loop join. To create this effect, take a table in a join that you do not want accessed first, put it in a table expression and create a correlated reference to the table that you want accessed first. DB2 generally cannot access the table in the table expression until the data is available from the table in the correlated reference. This can be really effective and is great for transactions, but really a bad idea for larger queries. See Section 3 for more details.
- **Force materialization of the table to be accessed first.** You can put a table you want accessed first into a nested table expression and force materialization (see Section 2 on merge versus materialization). You can use DISTINCT, GROUP BY, or ORDER BY in the table expression to force a sort. This could be really useful in situations where joined tables are not in good clustering

sequence and DB2 has chosen nested loop join. It may also be an alternative to hybrid join or nested loop join with intermediate sorts (or hash join) but you really need to benchmark these attempts to see how effective they are.

- **Code non-transitive closure predicates.** Depending upon the version of DB2 you are using, LIKE and subqueries are not available for transitive closure (see Section 2 section on query transformation). DB2 uses transitive closure to generate local predicates against more tables in a join. In that way, more access path choices and indexes are available for consideration. If you want to discourage this behavior and disable transitive closure for a given predicate, you could consider using a subquery. To use a subquery, you could change the following, “ON A.COL1 = B.COL1 WHERE A.COL1 = ?”, to “ON A.COL1 = B.COL1 WHERE A.COL1 = (SELECT ? FROM SYSIBM.SYSDUMMY1)”.

Each effort to influence SQL optimization with a trick should only take place after other options have failed.

Section 6:

Table and Index Design for High Performance

Logical database design is the responsibility of the data administrator. Physical database design is the responsibility of the DBA and is to be performed in cooperation with the application development team. The database design decisions should be based on facts. To sit in a room and debate with people about how you think something will perform is a ridiculous waste of time, and most often results in an over-designed database and/or process that performs horribly.” I don’t sit in such meetings anymore and have little to no tolerance for them. The basic rule of thumb is to build the database that the logical modelers developed, as long as it is in third normal form, and use database automation when possible to improve performance and reduce application development time. This saves significant time in developing a physical model and leaves the performance concerns to be addressed in two tasks: fix it when it’s broke, and let the database tell you how to design the database.

If the development life cycle does not contain allotted time for performance analysis and tuning then management is not concerned with performance and you shouldn’t care about performance because no one will ever support your recommendations. Of course, management is concerned about performance and so there better be a tuning phase of development. If you subscribe to the “fix it when it’s broke” philosophy of performance tuning then this phase of development is the time for EXPLAIN, benchmarking, and monitoring of the application and database in action. Changes can be implemented at that time to improve the performance of the application as prescribed by a service level agreement (SLA). If you do not have SLAs and management is expecting the application to run “as fast as possible”, then recommend they buy an infinite number of machines and take a vacation. If extreme performance challenges are expected then a performance based design is a good move; but what choices should be made? The answer is “it depends” or better yet “I don’t know”. This is the point at which you can have a meeting to discuss the possible design choices for high performance, and then establish a means to test and prove them.

So, you need to come up with a high performance design, but how to begin? My philosophy has been and always will be to let the database tell you how to design the database. If you understand the performance features of the database (many of which are covered in this book) then you already have a starting point. Now, which performance features are advantages for your particular database and application and which will hurt the performance of your database or application? The answer is in performance testing, but with potentially no database, data, or application code available to test, what do you do? The answer is to simulate the entire design, or portions of it in a proof of concept (POC). Since, in the vast majority of database applications, most of the business logic is contained in the application layer, you can simulate the database layer quite easily with dumbed down SQL access and table designs that reflect the ultimate design. Using SQL to generate SQL and data, SPUFI and REXX for your application layer, you can build test cases in as little as a few hours or days. Then using the testing techniques described in Section 9 of this book you can make variations to the database and application design and measure the effects. In this manner, the ultimate database design you produce will be one based upon the performance measurements taken during the POC, and not based upon the guessing about performance in a meeting room. It is this very technique that has led to some of the most extreme database applications reaching astounding transaction rates and data volumes from day 1 and with little to no post-implementation tuning.

Selecting a Table Space Type and Features

The types of table spaces available today include segmented, classic partitioned, universal range-partitioned and universal partition-by-growth. Given the growing number of database features that are dependent upon universal table spaces, it seems quite logical to choose either range-partitioned or partition-by-growth universal table spaces. It does not seem reasonable to choose segmented unless you have many small code tables that you'd like to share a table space, or classic partitioned unless you are pre-DB2 10 and want to use the MEMBER CLUSTER feature.

Partitioning

Table spaces can be partitioned either as universal partition-by-growth or range-partitioned. These are the modern ways to partition data within a DB2 data server and should be the choices made going forward. Also available is table and index controlled classic partitioning. However, there are no advantages moving forward to selection these options. Since partitioning is a table feature, please see the section in this section about range-partitioning a table to understand the performance advantages to range-partitioning.

An additional advantage to partitioning is that certain table space settings are possible at the table space partition level. This allows for certain performance adjustments to be made via ALTER statements as certain partitions become more or less active, including free space, TRACKMOD, compression, and group buffer pool caching.

Group Caching

When operating in a data sharing environment you can tell DB2 which level of caching is desired for the group buffer pools. The choices are ALL, CHANGED, and NONE. Generally, the default option CHANGED is adequate to balance between group buffer pool performance and concurrency when updates are allowed across all data sharing members. There are some delicate choices here, however, between the distribution of processes, updates across members, range-partitioning configuration, and

performance. In many situations, the majority of high performance updates to a table come in the form of large scale batch operations. This is where something I like to call “application controlled parallelism” comes into play and where group caching can play a major role in the performance of these large scale operations. In these situations, partitions match processes, group caching is set to NONE, locking is at the partition level, readers that read across partitions are doing so as uncommitted readers using WITH UR (if possible), and system affinities are applied. All of this is detailed in the section of this section on range-partitioning of a table. Setting the group caching to NONE is only useful for this situation where all large scale update processes are directed to individual members and executed during online quiet times.

For any large scale batch operation, it is strongly advised to attempt to remove group buffer pool dependency for all of the objects that the batch operation will touch. This is accomplished via the following DB2 command:

```
-ACCESS DB(<database name>) SP(<space name>) PART(n)  
MODE (NGBPDEP)
```

The general guideline is to issue this command on the data sharing member where the large scale batch process will run. However, I have had the best success in avoiding the overhead of group caching for group buffer pool dependent objects by running the command on each and every data sharing member.

Lock Size

Setting the lock size is a practice in the balance between performance and concurrency. The lock size of a table space has to be balanced with the expectations of the application, how the application is coded, the concurrency control setting for the application, and the isolation level that is set by the application. Many times people overlook this balance and attempt to resolve perceived concurrency or performance problems by setting the lock size to row. What they don't realize is that by setting the lock size to row you can be dramatically increasing the number of locks held by an application and managed by DB2. This increased number of locks can consume significant memory and CPU resources as well as dramatically increase the number of messages sent between data sharing members and the coupling facility. I have only used row level locking in extreme concurrency situations on tables that have a small quantity of rows shared by a high number of concurrent threads. I have never used a lock size of row in a high volume, high transaction rate situation because every test I have conducted has demonstrated significant CPU and elapsed time resource consumption over other lock sizes. In almost all implementations, I use a lock size of PAGE or ANY, but mostly PAGE, so as to avoid lock escalations.

If possible in your environment, you can use one of my favorite configurations - what I call “application controlled parallelism”, which is described in the table range-partitioning section in this section. In these situations, any transactions that update objects or any high volume batch update operations are split by a logical value that matches the range-partitioning and are applied to specific partitions. The lock size is set at the partition level to minimize locking overhead. All readers use

WITH UR to read uncommitted, and in some cases the update transactions employ an optimistic locking strategy. See the rest of this section for more details on these options.

Another option to adjusting the lock size that is worth investigating is currency control or concurrent access resolution. This is a feature that allows readers to see only data that is currently committed or to wait for an outcome. By setting the readers to read only currently committed data, they will not wait for any locks, but instead return only the data that has already been committed by other applications. This is a really cool feature, but is limited in its implementation on DB2 for z/OS in that the currently committed reader will avoid uncommitted inserts and deletes, but not updates. Therefore, in order to implement this strategy, all updates would be required to be executed as deletes and inserts.

Please review all options, including uncommitted read, optimistic locking, concurrency control, and other isolation level options prior to, or in conjunction with, selecting a lock size.

MEMBER CLUSTER

This feature is specifically designed for high performance and availability table spaces that are subject to very high insert activity. Setting MEMBER CLUSTER together with PCTFREE 0 and FREEPAGE 0 for a table space does two things: it establishes an alternate insert algorithm by placing the majority of inserts to the end of the table space or target table space partition; and it also sets one space map page for every 99 data pages and reserves the space map pages exclusively for every data sharing member acting against the table space or table space partition. The space map feature can significantly improve the performance and concurrency of high insert applications that are distributed across data sharing members because the unshared space map pages are not a source of contention in the coupling facility. Setting MEMBER CLUSTER, like APPEND ON at the table level, will ignore clustering and place inserts at the end. However, unlike APPEND ON, it will be more aggressive in searching for available free space before extending a table space.

Page Size

You want to, basically, select a page size that is meaningful for the table, and not necessarily for a belief that a large page size will result in cheaper I/O and getpage costs. In any performance test I have conducted that compared one page size over another (e.g. 4K versus 8K), there was never a performance gain realized for sequential or random access. There are always exceptions, however, so don't be shy about running some tests of alternate page sizes, but they shouldn't necessarily be high on the priority list.

A larger page size becomes more of a performance concern when you run into situations in which you are wasting space, thus impacting I/O and memory performance, due to hitting the 255 rows per page limit before filling a page. Also keep in mind that compression will play a role in the number of rows that can fit on a page as well. Larger page sizes are more critical when it comes to LOB storage. Keep in mind that a LOB table space has a limit of one LOB per page. If you are storing mostly smaller LOBs, a smaller page size may save space and improve performance.

Compression

I'm a big fan of compression and apply compression more often than not to a table space. Compression not only saves on mass storage space, but improves memory utilization in the buffer pools and improves the performance of many types of processes. I have found that in the vast majority of cases the cost of compression is more than offset by the improved I/O and memory efficiencies, and will generally accept a compression rate of 45% or higher for my table spaces. The only time I avoid compression is for tables that store primarily numeric values or tables with a compression rate of less than 45%.

Table Design Features and Performance

Range-Partitioning

There are several reasons for choosing range-partitioning:

- Rotating and aging data
- Separating old and new data
- Separating data logically for region or business segment based availability
- Spreading data out to distribute access
- Logically organizing data to match processes for high concurrency

One of the performance advantages to range-partitioning is in distributing the data to spread out overall access to the table space. What this means is that index b-tree levels can potentially be reduced due to reduced entry count per partition and that can reduce I/O's and getpages to indexes. In addition, free space searches could be more efficient in a partitioned situation since the data is broken up and the searches are on a partition-by-partition basis. The same is true for any partitioned indexes defined against the table space.

From a performance perspective, range-partitioning is useful to enable a high degree of parallelism for queries and utilities. In addition, queries coded against range-partitioned tables can take advantage of partition elimination, also sometimes called page range screening, when predicates are coded against partitioning columns. These predicates can contain host variable or parameter markers, literal values, and, beginning with DB2 11 for z/OS, joined columns.

For the highest level of partitioned table access possible, I have subscribed to a design technique called "application controlled parallelism". This is a technique that is useful when there exists high volume batch processes acting against very large OLTP data stores. In these situations, it is a careful balance between partitioning, clustering (or intentionally not clustering), and distribution of transactions in a meaningful way across multiple application processes. What I mean by a meaningful way is that the distribution of inbound transactions has to match the partitioning and/or clustering of the data in the table space partitions accessed during the processing. In this way, each independent (or sometimes dependent) process operates against its own set of partitions. This will eliminate locking contention and possibly allow for a lock size of partition and use of MEMBER CLUSTER to further improve performance of ultra-high insert tables.

When the input is clustered in the same sequence as the data is stored, or you are using MEMBER CLUSTER or APPEND ON to place data at the end of a table space partition, DB2 can take advantage of sequential detection to further improve the performance of these types of processing. In these

situations I have achieved transaction rates as high as 10,000 per second, single table insert rates as high as 13,000 per second, and application wide insert rates as high as 17,000 per second with my next target set at 21,000 per second. These are not test cases but real world results.

On a final note about the distribution of transactions across partitions, please be aware that I subscribe to a meaningful distribution of data across partitions and an application process pattern to match. I have seen many implementations where data and transactions are randomized in order to distribute access randomly and eliminate “hot spots”. In my opinion this typically results in a random access nightmare and potentially extreme locking conflicts. In every one of my successful implementations there were hot spots and DB2 handled them just fine. Between the power of index lookaside, sequential detection, good buffer pool configuration, and advanced DASD subsystem, there are no true physical hot spots.

Materialized Query Tables

A Materialized Query Table (MQT) is a table that is designed to help the performance of complex queries that are coded against one or more very large tables. The idea is that a single table can be used to present a summary of data that is normally produced by a complex query and then have DB2 intercept the complex query when it is executed and direct the request to the MQT instead of the original large tables. In this way, MQTs are most effective for report queries that aggregate data, especially in situations where there are many similar report queries that can share the same MQT and thus avoid many redundant accesses to the original tables that the MQT replaced. What follows is a simple example of the definition of an MQT (untested, just an example for discussion purposes).

```
CREATE TABLE EMPINFO (EMPNO, SALARY, AVGSAL) AS
  (SELECT EMPNO, SALARY, DEPTSAL.AVGSAL
   FROM EMP EMP
  LEFT JOIN
   (SELECT WORKDEPT, AVG(SALARY)
    FROM EMP
   GROUP BY WORKDEPT) DEPTSAL(WORKDEPT, AVGSAL)
  ON EMP.WORKDEPT = DEPTSAL.WORKDEPT )
DATA INITIALLY DEFERRED
REFRESH DEFERRED
MAINTAINED BY SYSTEM
ENABLE QUERY OPTIMIZATION;
```

If we assume that the EMP table is extremely large, then the aggregation of values in the table expression can become quite expensive. The ENABLE QUERY OPTIMIZATION setting will give DB2 the ability to analyze queries that are similar to the one coded in the MQT definition and elect to use the MQT to retrieve the data rather than the original query. Keep in mind that the data in the MQT may not be completely in sync with the data generated by the original query, and the ability of DB2 to select the alternate access is dependent upon the CURRENT MAINTAINED TABLE TYPES and CURRENT AGE special registers, as well as the SPRMMQT subsystem parameter. Please review the DB2 Administration Guide and SQL Reference for more information on the optimization of queries eligible to use MQTs. In my opinion, if you want to properly take advantage of an MQT then you are coding a

query directly against the MQT, which will both guarantee the performance advantage as well as the query results since the data in the MQT is actually a snapshot in time. That snapshot may be important and relative for report queries that are issued repeatedly and expect the same results within a given period of time.

People often confuse MQTs as a performance choice for transaction processing systems. Please keep in mind that this feature is designed in support of report queries and data warehouses, not transactions. The query transformation feature is only available for dynamic SQL and the MQT is not guaranteed to be a current representation of the data in the original tables that the MQT is based upon. The confusion begins when you believe the MAINTAINED BY SYSTEM setting means that DB2 is going to update the MQT every time a change is made to one of the original tables. This is not the case at all. The only way to refresh an MQT is via an unload and a LOAD, use of a REFRESH TABLE statement, or by creating a set of triggers that would maintain the MQT in sync with the source tables.

Storing Large Strings and Large Objects

There are multiple choices for the storage of large strings which basically include the use of VARCHAR and CLOB data types. A single VARCHAR column can be defined up to 32,704 bytes for a table stored in a table space with a page size of 32K. If your long character strings are going to be under that length, a VARCHAR is the preferred method of storage since all of the data can be contained in the base table and can be compressed. You may even want to consider using multiple rows of VARCHAR fields for long string storage as even this could possibly outperform CLOBs, but of course, additional programming will be required in order to assemble and disassemble the strings.

CLOBs can be up to almost 2GB in length and so a CLOB data type is a good choice for very large strings when you desire a more simplified approach to data access. Keep in mind, however, that there are tradeoffs with CLOB performance and management. Large objects are stored in separate table spaces from the base table data, and access to these auxiliary LOB table spaces is completely random. In addition, all LOBs generally require special attention when running LOADs, unloads, REORGs, and other DB2 utilities. LOB table spaces cannot be compressed and a maximum of only one LOB can be stored per page of a LOB table space. So, it is completely senseless and a large performance detriment to define small LOBs or to define large LOB columns only to put small LOBs into them.

You can elect to store all or a portion of a LOB into the base table via the INLINE option of a LOB column definition. This allows for a LOB definition with all the advantages of storing one or more 2GB objects in a table, while still taking advantage of potential sequential access and compression that is available for the base table. Using the INLINE LOB feature will only be a performance improvement if all or most of the LOBs stored in the table will fit into the base table portion without overflowing into the LOB table space. If even a modest percentage (depends upon the LOB size, but I would say 10% of LOBs) end up overflowing, then in-lining the LOB data can actually result in a performance detriment due to the extra access to the LOB table space and potential extreme waste of space in the LOB table space.

XML Storage

Is the utilization of an XML column a performance feature? Well, it depends! From a pure performance perspective you need to answer some questions.

- Is the database going to be used simply to deposit and retrieve XML?

If this is true then an XML data type is not the performance choice. You are better off storing the XML as a large string and avoid the overhead that would be incurred with an underutilized XML column.

- Do applications need to share the data in both XML and relational format?

In this case you may be better off storing the data in relational tables. DB2 provides all of the functionality to both shred and build XML documents, as well as do direct translation from relational to XML and back. This may give the highest level of flexibility, and may avoid the unnecessary and costly duplication of data. Just be sure to test the overhead of the execution of the many functions required to do the back and forth conversion.

- Is direct access and validation of the XML documents required?

The performance choice here just may very well be to utilize an XML column. This gives direct access to all or a portion of an XML document, enables searching and indexing of portions of a document, allows the translation to and from relational, enables updates of only a portion of a document, and enables document validation.

From a development perspective you may consider the use of an XML data type to be a performance advantage. Here are some of the built-in XML features of DB2:

- XML schema storage and validation
- XQuery and XPATH expression processing
 - Variable value processing
 - Looping
 - Data manipulation
- Conversion to and from relational data
- Partial document update
- XML indexing

Taking advantage of these features can be a performance boost from a development perspective in that processes that may take months to code in Java or another language may take days or even minutes to code using the DB2 built-in XML functionality. In addition, if analysis of all or a portion of an XML document can take place before all it is returned to an application, then this can represent a significant performance improvement over simply passing the entire document to and from an application, especially when that passing of data is across a network.

Hash Access Enabled Tables

A hash access enabled table is a very specific table design directed at a very specific and limited form of access. It is intended for tables that have a unique primary key and are always or extremely frequently accessed by queries that use the primary key for direct row access. In other words, the application is always accessing the table via a predicate such as `PRIMARY_KEY_COLUMN = <value>`. A table can be enabled for hash access by using the phrase `ORGANIZE BY HASH UNIQUE` in the table definition. You must specify the columns used for the hash as well as the amount of extra table space storage required to hold the hash values.

The performance advantage to hash tables is that you do not need an index and the simple direct predicates will find the location of the data by using the hash. This can avoid the I/O's and get pages that would otherwise be associated with matching index access.

The performance disadvantage to hash tables would be in having predicates other than the simple primary key predicate coded against the table. In addition, the hashing of the data in the table reduces the opportunity for sequential detection. In other words, if you have large volume transaction processing that involves repeated primary access based upon input from a sequential file then there is a very good chance that clustering, which is not possible for hash, by the input key sequence will outperform the hash access due to the availability of sequential detection and index lookaside.

The storage area specified for the hash area may not be large enough, so DB2 creates an overflow index to use when the area is completely full. You can adjust the storage size and REORG to fit your needs, and tell REORG to calculate the size of the hash area automatically. You should also monitor the table space hash access in the SYSTABLESPACESTATS real-time statistics catalog table to make sure the hash access is effective.

Optimistic Locking Support

A properly implemented optimistic locking strategy can represent a significant performance and concurrency improvement to an application that updates data. Here is optimistic locking in a nutshell:

- All tables contain a timestamp column that represents the time of last update or insert.

The timestamp column in the table is the key to optimistic locking and enables concurrent processes to share data without initial concern for locking. As an application carries the data along it holds the responsibility for maintaining the update timestamp in memory and checking it against the table in the database for a given primary key. The update timestamp itself is not a part of the primary key, but is used for change detection for the row represented by the primary key. What follows is a very simple example of such a table definition.

```
CREATE TABLE BOB1
(PRI_KEY BIGINT NOT NULL
,DATA_COL CHAR(50) NOT NULL WITH DEFAULT
,UPDATE_TS TIMESTAMP NOT NULL);
ALTER TABLE BOB1 ADD CONSTRAINT PK_BOB
PRIMARY KEY (PRI_KEY);
```

- All tables are read with isolation level uncommitted read.

Using an isolation level of UR means you can potentially be reading dirty uncommitted data, however the application will not wait for any lock that is held on a piece of data that has been updated by a concurrent in-flight transaction. The read of the data absolutely has to include, at the very least, the primary key value and the update timestamp. This update timestamp has to be retained for use in testing the validity of the data held in memory by the process. What follows is an example of such a read.

```
SELECT PRI_KEY, DATA_COL, UPDATE_TS
FROM   BOB1
WHERE  PRI_KEY = ?
WITH UR
```

- The timestamp column is tested and updated with every update.

When a process moves to update data it has previously retrieved, it has to do two things. First, it has to test the update timestamp to see if the state of the data in the table is the same as when it was initially retrieved, and second, it needs to update the update timestamp to reflect the change in the state of the data as a result of its processing. What follows is an example of such an update. In the example, the update timestamp is tested for the value that was retrieved when the row was originally read and updated to the timestamp of the update.

```
UPDATE BOB1
SET    DATA_COL = ?
,      UPDATE_TS = ?
WHERE  PRI_KEY = ?
AND    UPDATE_TS = ?
```

This technique does a great job of improving the performance and concurrency of an application. It does, however, relieve the database engine of some of the concurrency control by using uncommitted read and puts a tremendous responsibility on the application to properly maintain the update timestamps and, thus, the data integrity.

DB2 provides support for optimistic locking which removes some of the application responsibility in the control of optimistic locking. This support comes in the form of an automated generated column called a row change timestamp column. This column can be generated by default or always by DB2, but for optimistic locking situations it is strongly advised to force it as generated always. The advantage of this generated column is that it moves the responsibility of updating the timestamp column out of the hands of the application and into the hands of DB2, thus guaranteeing the accuracy of the timestamp value enterprise-wide. This is quite obvious if we compare the fully application-controlled, optimistic locking strategy laid out above with the same strategy that uses the built-in DB2 functionality.

- All tables contain a timestamp column that represents the time of last update or insert as maintained by DB2.

```
CREATE TABLE BOB1
(PRI_KEY BIGINT NOT NULL
, DATA_COL CHAR(50) NOT NULL WITH DEFAULT
, UPDATE_TS TIMESTAMP NOT NULL GENERATED ALWAYS FOR EACH ROW
      ON UPDATE AS ROW CHANGE TIMESTAMP );
ALTER TABLE BOB1 ADD CONSTRAINT PK_BOB
PRIMARY KEY (PRI_KEY);
```

- All tables are read with isolation level uncommitted read.

The difference here is in how the update timestamp is retrieved. The row change timestamp column is not named in the SELECT but instead specified. You could also select the column by name if desired.

```
SELECT PRI_KEY, DATA_COL, ROW CHANGE TIMESTAMP FOR BOB2
FROM   BOB1
WHERE  PRI_KEY = ?
WITH UR
```

- The timestamp column is tested and updated with every update.

While the testing of the timestamp still has to be performed by the application, the updating of the timestamp happens automatically by DB2. This ensures a higher level of data integrity for optimistic locking.

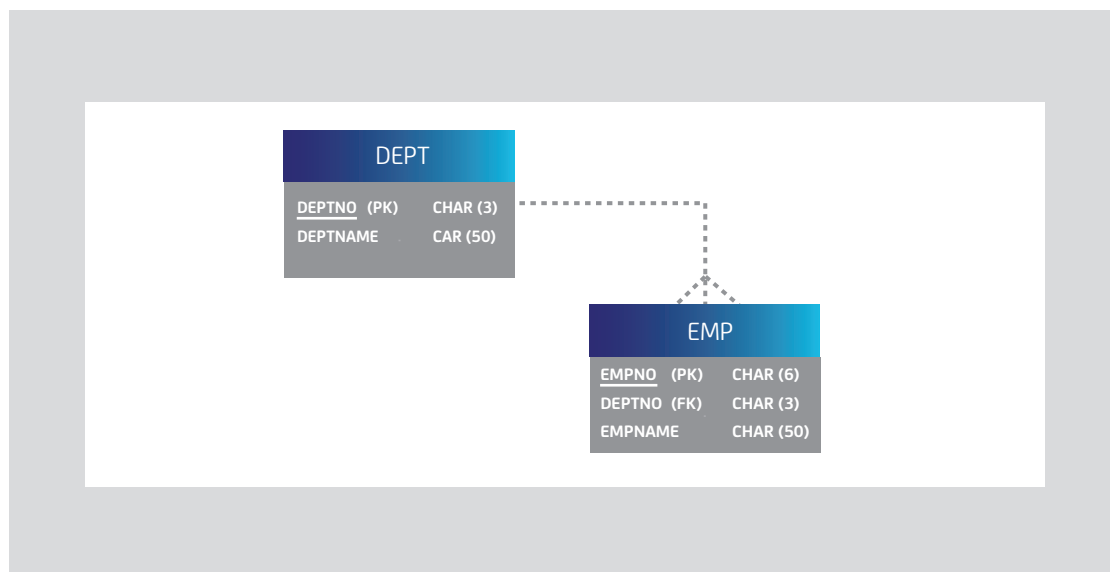
```
UPDATE BOB1
SET     DATA_COL = ?
WHERE  PRI_KEY = ?
AND    ROW CHANGE TIMESTAMP FOR BOB1 = ?
```

Database Enforced Referential Integrity

There is a lot of debate within the DB2 community with regards to the cost, benefits, and manageability of database enforced RI. From a performance perspective, I am a fan of database enforced RI. Although, everything is relative, the less controlled your data is the more the database enforced RI is going to cost. If you consider the fact that the most expensive thing you can do is to make a call to the database engine, then database enforced RI is going to eliminate many calls to the database. Enforcing RI is a data intensive process, and any data intensive process should be handled as close to the data as possible. For example, utilizing cascading deletes might be viewed as too resource intensive, especially for multiple levels of cascading. However, what is the alternative? The answer is many calls to the database to perform the cascading of the deletes at the application level, which will ultimately result in a longer overall code path and greater quantity of CPU consumed. Also, while one may contend that an application can check a parent key once prior to the insertion of many child rows of data, DB2 has the ability to cache repeated calls for parent key lookups and so the impact of such operations on performance is minimized.

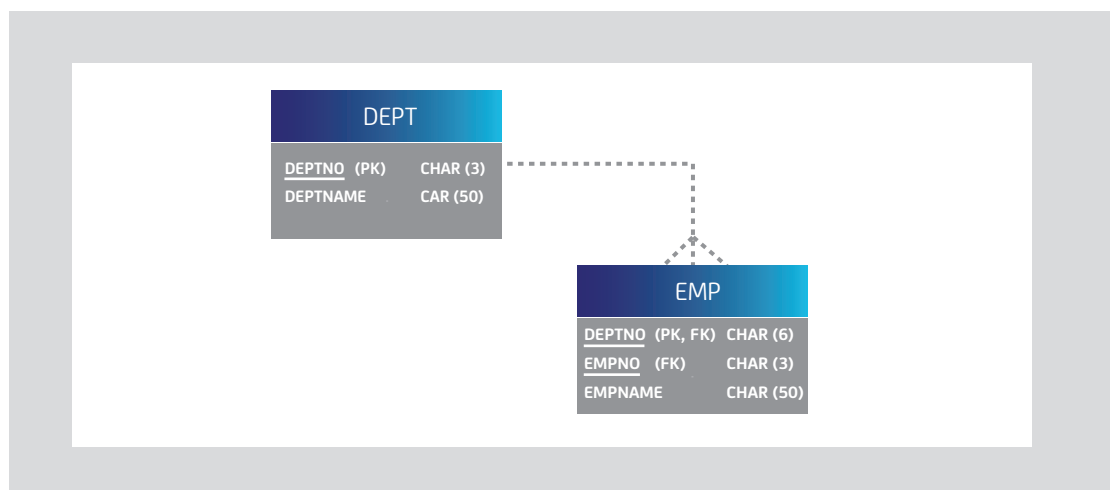
An identifying relationship is one in which an instance of the child entity is identified through its association with a parent entity; and the primary key attributes of the parent entity become primary key attributes of the child. If you examine the DB2 sample database, you will see that the primary key of the employee table is completely independent of the department table, but there is a foreign key to the department table. This is considered a non-identifying relationship. The following simplified example shows the original relationship.

Figure C.



If the relationship between the employee and the department was identifying, then the employee table would contain the department number primary key of the department table as part of the primary key of the employee table, as in the following simplified example.

Figure D.



Actually, not only is the department number part of the primary key of the employee table, but it is the leading column. From a pure performance perspective there are two advantages to establishing these types of relationships in the physical model and DDL: reduction in the number of indexes required to support the relationships, and the ability to use the primary key indexes as clustering indexes. These two things are extremely important for performance or database enforced, or even application enforced, RI as it minimizes the number of indexes in the database, reduces the frequency of random I/O's to navigate relationships in the database, and promotes common clustering of related tables. In the example with the non-identifying relationship, there would be one supporting index for the primary key on the department table, one supporting index for the primary key on the employee table, and one supporting index for the foreign key on the employee table. Clustering would

be via primary key indexes. In the example that utilizes the identifying relationship, there would be one supporting index for the primary key on the department table and one supporting index for the primary key and foreign key on the employee table. Clustering would again be via primary key indexes. Not only does one index serve two purposes for the employee table, but the common clustering will result in fewer I/O's, less random I/O, and promote DB2's ability to utilize the dynamic prefetch and index lookaside performance enhancements described in Section 2 of this book.

It is possible to achieve a common cluster with the design that utilizes a non-identifying relationship by establishing cluster based upon the foreign key index of the department number on the employee table. However, full clustering may be desired in situations where employees are being processed in sequence, and that may require the creation of a third index on the employee table using the department number and employee number. This is all obviously relative to your design and the processes that will be acting on the tables. Nonetheless, from a performance perspective, the identifying relationships utilizing the parent primary keys as leading columns, is the proper choice.

The pros and cons of utilizing identifying relationships have to be carefully considered during the development of the database such that performance can be balanced with functionality and flexibility of the design. There will be tremendous pushback from developers and data administrators against DBAs who want to construct these types of identifying relationships, especially for object-relational designs. In many cases I simply accept the original design that utilizes system-generated keys and non-identifying relationships for most objects in the database, but then utilize the identifying relationship design (with leading column parent keys) for objects that have high performance considerations. The only way to convince modelers and developers is to establish benchmark tests in a proof-of-concept (POC) and demonstrate the performance advantage and disadvantage of each design all while making sure that management understands the operational costs of each design. Hopefully, a balance can be struck! This is the daily dance we play in high performance design. Please see Section 9 for details on establishing performance benchmarks and POCs.

Always support your foreign keys with the proper backing indexes.

Check Constraints

Table check constraints allow us to add automated validation of column values to the definition of tables in our databases. For example, if there is a one-character column in the employee table called SEX that represents the gender of the employee represented by the row in the table, then a check constraint can be established on that column to force the value to either "M" or "F". Then, when a row is inserted or modified the input code is verified against the check constraint. If the constraint is violated, the statement fails.

While table check constraints can be a valuable tool in controlling and centralizing the control of column values, they are not at all to be considered a performance feature. This is especially true for simple code value validations. Once a check constraint is violated, control is returned back to the application. If there are multiple constraints against the table then there can possibly be multiple messages back and forth between the data server and application just to get a single row into the database. In addition, column validation is using data server CPU resources, which could be more expensive than application server CPU resources. Placing column validation in the application layer

will significantly reduce the number of calls to the data server and save CPU. It is important to perform a cache refresh on the application server at a frequency that can ensure a relatively high level of accuracy of the valid values.

Triggers

Triggers are a form of routines that are activated when a data change statement is initiated against a table with a trigger defined on it. Triggers are useful for many reasons:

- Setting advanced default values for columns
- Data validation
- Replication of data from one table to another
- Sending messages to other applications
- Invoking stored procedures when a data value changes
- Performing external operations when a table is modified

I'm a big fan of triggers, both from a functionality perspective and a database performance perspective. From a functionality perspective, triggers save development time because complex data validation and complex table changes can be performed within the context of a trigger; and in general, it is easier and faster to develop and validate the logic in a trigger than it is in the equivalent application logic. From a performance perspective, the operations performed by triggers generally tend to be data intensive. "If you change the data in TABLE A then this value in TABLE B will also change". Well, if the value in the second table always changes when the value in the first table changes then why call the data server twice to make the overall change? Instead, if the value in the second table can be changed within the action of changing the first table, then by all means you should define a trigger to perform that action. This will reduce the number of calls to the data server, as well as potentially decrease the duration of the unit of work required to make the change. The result is an improvement in database performance and concurrency. For example, if there is an update to a detail table and the update requires an update to a corresponding summary table, then a trigger could very well be the performance choice for making that associated update.

I have used triggers for my data-intensive multi-table operations with great success in many situations and encourage the use of triggers for these types of operations. In addition, I have also used triggers in support of database enforced RI that goes beyond the built-in capabilities of DB2 with great success. Although your performance may vary, I have found few situations in which triggers were a performance detriment when compared with the equivalent application logic that makes multiple calls to the data server.

Views

Views are a very complex subject from a performance perspective, and one could write an entire book that studies the effects of views on performance. In an effort to keep things simple I tend to avoid defining views in high performance designs. To be more specific, simple views against single tables that contain either all of the columns or a subset of the columns for security purposes are just fine. Even single table views with predicates can be useful in a high performance design. I tend to avoid more complex views that contain multiple table references, aggregation, or DISTINCT. The reason

being is that I want to be more in control of the SQL coded at the application level. In addition, DB2 has some issues with the ability to push predicates coded against views into the view definitions themselves. This is especially true in the case when view references in a SQL statement result in materialization, or when there are joins to views where predicate transitive closure can be applied to joined columns. With each new version of DB2 there are improvements to the ability to push predicates into views, but I still like to have full control. The issue of predicate pushdown is extremely difficult to understand, and thus is one reason I avoid views in high performance designs. It is always best to conduct thorough tests when using more complex views to make sure all filtering is getting pushed as close to the data as possible. It is especially important to be mindful of, and carefully test, predicate pushdown when using outer joins. Additional important considerations need to be made when using UNION in a view. Please see nested table expression in Section 2, as well as the temporal table section of this section, for more details.

When views are used and DB2 fails to adequately push predicates into a view, I tend to use either correlated nested table expressions or SQL table functions to achieve the desired pushdown. If you consider the following view and query, the local predicate will not be pushed into the view even though we only need the average salary information for one department.

```
CREATE VIEW DANL.V_AVGSAL (WORKDEPT, AVGSAL) AS
SELECT WORKDEPT, AVG(SALARY)
  FROM DANL.EMP
   GROUP BY WORKDEPT;
SELECT E.EMPNO, E.SALARY, DEPTSAL.AVGSAL
  FROM DANL.EMP E
 LEFT JOIN
      DANL.V_AVGSAL DEPTSAL
ON DEPTSAL.WORKDEPT = E.WORKDEPT
AND E.WORKDEPT = 'C01';
```

The performance choice would be to avoid the view and utilize either a correlated nested table expression or a SQL table function (with appropriate fixes applied).

```
SELECT E.EMPNO, E.SALARY, DEPTSAL.AVGSAL
  FROM DANL.EMP E
 LEFT JOIN TABLE
 (
  SELECT WORKDEPT, AVG(SALARY)
    FROM DANL.EMP E2
   WHERE E2.WORKDEPT = E.WORKDEPT
```

```

        GROUP BY WORKDEPT) AS DEPTSAL(WORKDEPT, AVGSAL)
    ON DEPTSAL.WORKDEPT = E.WORKDEPT
    AND E.WORKDEPT = 'C01';
CREATE FUNCTION DANL.EMP1 (INDEPT CHAR(3))
RETURNS TABLE (WORKDEPT CHAR(3), AVGSAL DEC(9,2))
RETURN
SELECT F.WORKDEPT, AVG(F.SALARY)
    FROM DANL.EMP F
    WHERE F.WORKDEPT = INDEPT
    GROUP BY WORKDEPT;
SELECT E.EMPNO, E.SALARY, DEPTSAL.AVGSAL
    FROM DANL.EMP E
LEFT JOIN TABLE(DANL.EMP1(E.WORKDEPT) AS DEPTSAL(WORKDEPT,
AVGSAL)
ON DEPTSAL.WORKDEPT = E.WORKDEPT
AND E.WORKDEPT = 'C01';

```

Views and “Instead Of” Triggers

There is an advantage to using views in conjunction with “instead of” triggers in certain object-relational designs. This all revolves around the concept that the developer is going to want to map single tables to single objects in the data access layer of the application. This type of design results in a proliferation of SQL calls to retrieve related objects in support of a transaction. Let’s assume, using the DB2 sample database, that an application will always collect information about departments and employees at the same time. If an object is mapped to the department table and another object mapped to the employee table, then each invocation of a transaction to get the data will result in two calls to the data server. If we were to create a view that joined the department and employee tables together, the data access layer of the application can map only one object to the view, which it treats as a table. Any requests to populate the object result in only one call to the data server versus two. Updates to the object are not possible in this situation, however, unless “instead of” triggers are employed. In addition, the underlying join of the two tables may represent a denormalization to the application, which may or may not be acceptable.

An “instead of” trigger is a type of database trigger that is defined against a view and causes an action to be fired whenever there is a data change statement executed against the view. This potentially allows for updates, inserts, and deletes against a multi-table view. From an application perspective a change is being made to a table, but in reality the trigger intercepts the change and the logic within the trigger can take an alternate action, such as properly updating the underlying base tables.

I’ve never actually deployed any design using this technique and so please thoroughly test this as a potential option to denormalization in these sorts of situations.

Temporal and Archive Tables

DB2 temporal tables are an outstanding database automation feature that can replace significant amounts of application programming. An application-period temporal table is a table that has been defined with the appropriate time columns along with the specification of a business period. They’re

useful for managing data active during a certain time, and for allowing data to be staged (active in the future), current and historical. An application-period temporal table can be useful for such things as reference tables and code tables where descriptions will change over time, or for more involved applications such as the active period of a specific insurance policy. The application-period is created by including a “PERIOD BUSINESS_TIME” clause that identifies the start and end time columns. These columns can be DATE or TIMESTAMP data types. A system-period temporal table is a table that has been defined with the appropriate timestamp columns along with the specification of a system-period; it can be useful when all changes to a table need to be tracked with a retained history to fulfill auditing or compliance requirements. The system-period is created by including a “PERIOD SYSTEM_TIME” clause that identifies the start and end timestamp columns. When you define a system-period for a table, DB2 is in control of the start and end timestamps; the timestamps reflect the state of the data for the time specified. Data that’s currently effective is represented. Optionally, you can define a history table and associate it to the base table. Any changes to data then results in “before” images of the changed data being automatically replicated to the history table, with appropriate updates to the timestamps in both the base and history tables.

The main incentive for using DB2 automated temporal tables is indeed the automation. DB2 can automate the updating and control of temporal data, which can result in a dramatic reduction in application development time for these sorts of implementations. DB2 will also automate the movement of data from base to history tables for system-period temporal tables. Time travel queries can be managed by either the application or DB2, with the DB2 option being the obvious choice for reduction in application development time.

From a performance perspective, be aware of the following situations when implementing temporal tables:

- Time travel queries will introduce range predicates into what may seem to be simple primary key queries against application-period temporal tables. In addition, time travel queries against system-period temporal tables will introduce a UNION ALL between the base and history table. Please see the information on UNION ALL and nested table expressions in Section 2 for details.
- When implementing a design that incorporates application-period temporal tables know that database enforced RI is not directly possible for these tables. DB2 doesn’t currently support time-based referential integrity. You can define a primary key on an application-period temporal table, but can’t establish relationships with other tables. Since the whole point of application-period temporal tables is to add a time component to what was otherwise a primary key, this adds some complexity to a database design that uses application-period temporal tables. One solution is to establish separate key tables that support the database-enforced referential integrity, while their application-period temporal children support storage of the detailed elements.
- System-period temporal history tables should be implemented as high performance insert-only tables. Please see the section on these types of tables in this section.

For more details and examples about the time travel and temporal tables, please refer to the June/July 2012 issue of Enterprise Tech Journal and my article, “How to Leverage DB2’s Automated Time Travel Queries and Temporal Tables”.

<http://enterprisesystemsmedia.com/article/how-to-leverage-db2s-automated-time-travel-queries-and-temporal-tables>

Archive tables share some similarities to system-period temporal tables, especially in the area where DB2 can automatically move data from a base table to a history/archive table. Archival tables are designed with a base table, also called the archive-enabled table, and an archive table. When archiving is activated, any rows deleted from the archive-enabled base table are then inserted into the archive table. Archival activation can be controlled via two built-in global variables. Therefore, it is quite easy to control whether or not deleted rows are automatically moved to the archive table, and also whether or not a SELECT from the archive-enabled table will consider data from both the base and archive table. When the reading of archived data is activated, the underlying table access is via a generated UNION ALL between the base and archive table. This UNION ALL access is very similar to the access generated for system-period temporal tables, so please reference that information in this section, as well as the information on UNION ALL and nested table expressions in Section 2 for details. Just as with system-period temporal history tables you can elect to design the archive table as a high performance insert table.

CCSID

Are you thinking about the CCSID of your character data in use as a potential performance advantage or disadvantage? It is quite easy to contemplate the performance advantage of EBCDIC with a code page of 037 when your data is stored and accessed by locally run COBOL programs also using a code page of 037. This is how applications were run back in the “good old days” of local access by batch jobs run on the mainframe. That world no longer exists, and yet in the vast majority of situations, tables are defined on DB2 for z/OS as CCSID EBCDIC. By not considering CCSID when creating tables, you run the risk of increased CPU consumption and elapsed time spent performing code page conversion for all of your character columns. If your primary access is via local programs written in traditional programming languages, then by all means use EBCDIC. However, if the majority of access is going to be via Java then consider UNICODE in order to avoid code page conversion. If the vast majority of the access is performed locally (local connection only) using a type 2 driver then the character strings should be stored as UTF-16. If the majority of access is performed remotely (even via local hipersocket connection) then the character strings should be stored as UTF-8. Be sure to test this thoroughly to see if it’s really going to be a benefit in your situation. Also, while advances in tooling on the mainframe have pretty much made the code set of the data stored irrelevant as far as viewing goes, it’s still important to be sure all the tooling you are using on the mainframe can deal with data stored in Unicode.

It should be noted that for SELECTs DB2 will send whatever the CCSID of the table is back to the application, and the conversion will happen in the driver. Also, in the latest version of the Java driver data conversion can happen within the driver before it sends data to the server if a describe has been done on the table in advance. Finally, you can direct the type 2 driver to talk UTF-8 to the data server by setting the `db2.jcc.sendCharInputsUTF8` property to a value of 1.

Sequence Objects and Identity Columns

Sequence objects and identity columns are both system-generated, numeric values that are most often used to generate next key values. They are a direct replacement, from a performance perspective, to the outdated technique of utilizing next key tables to generate ascending numeric keys. Identity columns and sequence objects are very similar in their behavior and settings, with the primary difference being that an identity column is part of a table definition and directly controls a column value, and a sequence object is its own independent object, completely separate from a table definition. While I generally discourage the use of system generated keys as part of a high performance design, these generated

values are the way to go when the design either requires system generated key values, or the design can tolerate the performance degradation that can be associated with system generated values and the resulting random database access. In short, you want to maintain a common clustering between tables that will be accessed together in a transaction, and if you are going to use sequence objects and identity columns for generated key values, make sure those keys are inherited as identifying (part of the child table's primary key value) and are common clustering maintained. As far as identity versus sequence object, I prefer the sequence object such that the actual sequence is independent of the table. Only in situations in which the assignment of system generated values is dependent upon a LOAD process would I consider the use of identity columns.

One of the great advantages of sequence objects and identity columns is the automatic assignment of values and retrieval of those values from a data-change-table-reference in an INSERT within a SELECT. This can replace multiple SQL statements with a single statement and dramatically improve the performance of an application. An example of this type of statement can be found in Section 3 of this book.

There are two settings which can greatly affect the performance of the assignment of values for sequence objects and identity columns: CACHE and ORDER. The impact of these settings can be quite dramatic in a data sharing environment.

- CACHE

This setting affects the rate at which a sequence object or identity column's value is generated from the DB2 system catalog and held in memory. The settings are either NO CACHE or CACHE n, where n is a number greater than or equal to two and represents the upper range of values that can be held in memory at any given time. The default is CACHE 20, which means that 20 values will be generated and cached if the cache is empty and a value is requested. Each time the cache is refreshed results in a synchronous I/O to the DB2 system catalog. Setting NO CACHE means that every request will result in a synchronous I/O, and thus is the antithesis of the proper performance setting. NO CACHE would only be a choice if a continuous sequence of values assigned is needed, but again this is not the performance choice. The downside of caching of values means that values that have been cached but not committed to an application will be lost upon the shutdown of a subsystem. In a data sharing environment, using CACHE n and NO ORDER is the high performance choice. Also, in a data sharing environment each member of the group will get its own cache. As far as the number of values to cache, the default of 20 works quite well until the requests for values gets into the thousands per second, dependent of course on available system resources (CPU, memory, DASD, etc). However, if your application is requesting sequence values in excess of 1,000 per second, I would strongly recommend increasing the cache value via an ALTER until any noticeable stress is relieved. This stress would be indicated in an accounting report for an application by excessive system catalog I/O activity and lock contention (if there is a high level of concurrency). The locking contention can be greatly exaggerated in a data sharing environment.

- ORDER

This setting specifies whether or not values are generated in the order of the request. The default setting is NO ORDER and this is the performance setting. Setting ORDER makes no sense from a performance perspective, and from an application perspective, the ordering of values is only guaranteed within a single application process and also requires the setting of NO CACHE to

guarantee the order. So, while it may make sense to specify ORDER for specific test cases in a controlled testing environment where performance is of no concern, there really is no need to set ORDER on in a production environment. The obvious performance choice is CACHE n and NO ORDER.

Volatile Tables

The VOLATILE table setting tells DB2 how to base its access to the table. By setting VOLATILE you are telling DB2 to use an index for access whenever possible; basically telling it to ignore the catalog statistics for the table. This is an especially useful setting in situations in which a table may be used to hold transient data, such as a transaction input staging table. These tables may fluctuate dramatically in size and the statistics may not accurately reflect the quantity of data in the table at any given time. VOLATILE helps guarantee the stability of the table access in these situations, especially for an application that uses dynamic SQL, since statement prepares might happen, often making the access path less predictable. I have used the VOLATILE setting quite successfully for tables that can dramatically vary in size from process to process, and recommend this setting specifically for these types of tables. However, keep in mind that for transaction input staging tables, if the entire table is being read all the time and the order of the data is irrelevant and there is no concurrency between processes, then you are better off not specifying VOLATILE or even not specifying an index on the table so that a table space scan is guaranteed. Otherwise, VOLATILE is the way to go.

Append

The APPEND table setting affects the behavior of inserts against a particular table. By default the setting is APPEND NO, and any rows inserted into the table are based upon the clustering index (except if MEMBER CLUSTER is set for the table space). Setting APPEND YES tells DB2 that any inserts into the table will ignore clustering and free space completely and place the new data at the end of the table. This is definitely a setting to consider for a high performance insert configuration, and can be an option to set alone or use in conjunction with the MEMBER CLUSTER table space setting. Keep in mind that by setting APPEND YES you are telling DB2 to only put inserted rows at the end of the table and ignore any space that may have been acquired by deletes. Therefore, the table space may need frequent REORGs if clustering is important for efficient query processing against the table. This also works a bit different than MEMBER CLUSTER. Please see the details about MEMBER CLUSTER in this section as well as the section on design recommendations for high performance insert in this section for more information about fast insert processing.

Index Choices

A good database design is going to require indexes. You create indexes to satisfy very specific requirements:

- Primary key support
- Foreign key support
- Unique rule enforcement
- Clustering
- Performance
- Partitioning support (but no longer required for partitioning)

Indexes are obviously important for good database performance. However, you need to deploy indexes wisely and conservatively. Many times the ready solution to a poor database design or any poorly performing query is to make a supporting index. People don't often consider the negative performance implications of creating a new index. Basically, for every new non-clustering index created another random reader is introduced for every insert and delete operation, as well as for any update operation that changes key values. Once an index is added, it is difficult to confidently remove it. In many situations in which indexes have been recklessly added to a database, you'll see the most expensive SQL statements, from a CPU resource perspective, being inserts and deletes, and the only way typically to tune these statements is to remove indexes. It is for these very reasons that I begin every new database design with a simple philosophy; one index and one index only per table. The objective is that this index is going to satisfy all of the requirements identified at the beginning of this section. If you can design for only one index per table then you are well on your way to the ultimate in high performance database design. If you carefully follow the guidelines in this section for partitioning, clustering, include columns, referential integrity and key inheritance, then you are most of the way there in developing a design that minimizes the number of indexes created. Also, remember that this is the design direction for performance, and you may be trading off flexibility in your efforts to reduce the number of indexes.

Partitioned and Data Partitioned Secondary

Partitioning can be useful for many reasons (see the section in this section on partitioning). For indexes there can be a benefit in that the dataset supporting the index partitions will be smaller and easier to manage, and the b-tree structure can possibly have fewer levels. Partitioned indexes can only be created against range-partitioned tables. You can also specify non-partitioned indexes.

There are two types of partitioned indexes: partitioned, and data portioned secondary. When the leading columns of the index key match the leading columns of the portioning key then the index is considered partitioned. When the leading columns do not match, the index is considered data partitioned secondary. Both partitioned and data partitioned secondary indexes can be used for clustering, which allow for some really great opportunities for doing a form of multi-dimensional clustering where one set of values is used to partition and a separate set of values is used to cluster within those partitions.

The primary benefit for performance is that queries can take advantage of the partitioning key columns to eliminate partitions during query processing. In order for a query to utilize partition elimination, it has to have predicates coded against the columns of the partitioning key. These columns can be parameter markers and host variables. Beginning with DB2 11 for z/OS you can get partition elimination on joined columns as well. Keep in mind that if you match on leading columns of a partitioned or non-partitioned index you can get index matching access and not partition elimination. For data partitioned secondary indexes, you can get partition elimination if you match on the index leading columns and include predicates against partitioning key columns as well.

Include Columns

Index include columns are a good way to get index-only access when needed against a given table without the overhead of creating an additional index. Since include columns can only be added to unique indexes the major advantage is to add them to existing unique indexes in order to achieve index-only access. Be mindful of the fact that adding additional columns to the index will increase

the index size and possibly increase the number of index levels. In addition, adding columns to the index could increase index activity if those columns are modified via updates.

After upgrading from DB2 9 for z/OS, it may be beneficial to perform an index consolidation by eliminating indexes that contain the same leading columns but have a column set that is a super set of an already existing unique index. Assuming that the extra index was created for performance and for certain queries to achieve index only access, the additional columns can be included in other existing unique indexes and then the extra index eliminated. What follows is my favorite query that identifies potential redundant indexes. This query looks at indexes within a schema (creator) to determine if there are any potentially redundant indexes. By redundant indexes, I mean indexes that have the same leading columns as other indexes. Once this query identifies potential redundancies then it is up to the DBA to determine whether or not an index can be eliminated. The query works by first finding indexes with different names that contain the same leading columns. It then takes the result of that comparison to produce a report that aligns the indexes, showing the full key of each index, and highlighting where the columns match up.

```
WITH IXMATCH(CREATOR, NAME, ICREATOR,
             INAME1, ICOLCOUNT1, IUNIQUE1,
             INAME2, ICOLCOUNT2, IUNIQUE2)
AS
(
  SELECT I.TBCREATOR, I.TBNAME, I.CREATOR,
         I.NAME, I.COLCOUNT, I.UNIQUERULE,
         I2.NAME, I2.COLCOUNT, I2.UNIQUERULE
  FROM SYSIBM.SYSKEYS AS K
  INNER JOIN
        SYSIBM.SYSINDEXES AS I
  ON    K.IXNAME = I.NAME
  AND   K.IXCREATOR = I.CREATOR
  INNER JOIN
        SYSIBM.SYSINDEXES I2
  ON    I.TBNAME = I2.TBNAME
  AND   I.CREATOR = I2.CREATOR
  AND   I.NAME < I2.NAME
  INNER JOIN
        SYSIBM.SYSKEYS K2
  ON    K2.IXNAME = I2.NAME
  AND   K2.IXCREATOR = I2.CREATOR
  AND   K.COLNAME = K2.COLNAME
  AND   K.COLSEQ = K2.COLSEQ
  WHERE K.IXCREATOR='<schema name here>'
  AND   (K.ORDERING <> ' ' AND K2.ORDERING <> ' ')
  AND   K.COLSEQ = 1
)
```

```

SELECT COALESCE(I1.TABLE_NAME,I2.TABLE_NAME) AS TABLE_NAME,
       COALESCE(I1.INDEX_NAME1, I2.INDEX_NAME1) AS INDEX_
NAME1,
       I1.COLNAME,
       COALESCE(I1.COLSEQ,I2.COLSEQ) AS COLSEQ,
       I2.COLNAME,
       COALESCE(I1.INDEX_NAME2, I2.INDEX_NAME2) AS INDEX_NAME2
FROM
(
  SELECT IM.NAME,
         IM.INAME1 CONCAT '(' CONCAT IUNIQUE1 CONCAT ')',
         IM.ICOLCOUNT1,
         IM.IUNIQUE1,
         K.COLNAME CONCAT '(' CONCAT ORDERING CONCAT ')',
         K.COLSEQ,
         IM.INAME2 CONCAT '(' CONCAT IUNIQUE2 CONCAT ') '
  FROM   IXMATCH IM
  INNER JOIN
         SYSIBM.SYSKEYS K
  ON     IM.ICREATOR = K.IXCREATOR
  AND    IM.INAME1 = K.IXNAME)
  AS I1(TABLE_NAME, INDEX_NAME1, COL_CNT,
        UNIQUE, COLNAME, COLSEQ,
        INDEX_NAME2)
FULL OUTER JOIN
(
  SELECT IM.NAME,
         IM.INAME2 CONCAT '(' CONCAT IUNIQUE2 CONCAT ')',
         IM.ICOLCOUNT2,
         IM.IUNIQUE2,
         K.COLNAME CONCAT '(' CONCAT ORDERING CONCAT ')',
         K.COLSEQ,
         IM.INAME1 CONCAT '(' CONCAT IUNIQUE1 CONCAT ') '
  FROM   IXMATCH IM
  INNER JOIN
         SYSIBM.SYSKEYS K
  ON     IM.ICREATOR = K.IXCREATOR
  AND    IM.INAME2 = K.IXNAME)
  AS I2(TABLE_NAME, INDEX_NAME2, COL_CNT,
        UNIQUE, COLNAME, COLSEQ,
        INDEX_NAME1)
  ON    I1.TABLE_NAME = I2.TABLE_NAME
  AND   I1.INDEX_NAME1 = I2.INDEX_NAME1

```

```

AND I1.INDEX_NAME2 = I2.INDEX_NAME2
AND I1.COLSEQ = I2.COLSEQ
ORDER BY TABLE_NAME
        , INDEX_NAME1
        , INDEX_NAME2
        , COLSEQ
WITH UR

```

Extended Indexes

It is possible to create extended indexes, which include indexes on expressions as well as indexes on XPATH expressions. These types of indexes can be extremely useful for improving the performance against portions of XML documents and for various search queries. For example, in the following situation a query was written to search on the last name of an employee in the sample database. However, if the name can contain both uppercase and lowercase characters, the search may have to incorporate a Stage 2 predicate.

```

SELECT EMPNO
FROM EMP
WHERE UPPER(LASTNAME) = UPPER(?)

```

This Stage 2 predicate is not indexable. However, an extended index can be created on the result of the expression and then Stage 1 index matching access is possible.

```

CREATE INDEX DANL.XEMP_LN
ON DANL.EMP
(UPPER(LASTNAME, 'EN_US') ASC);

```

Keys and Clustering

As mentioned throughout this section and other sections of this book, clustering is important for two major reasons: sequential processing of large batches of input and efficient access of tables in multi-table statements. When determining proper clustering of your tables, consider the major processes that will be hitting the data, and then cluster based upon the ordering of the input for the most significant process. You may need to choose more than one major clustering sequence for the tables in your database based upon whatever major inputs are predicted and the table accessed during those processes. This goes hand in hand with the concepts of partitioning for application controlled parallelism, using identifying relationships for primary/foreign keys, minimizing the number of indexes, and sorting input to major processes by the clustering keys of the tables that will be processed. Careful testing of predicted access patterns will tell you how to cluster. Remember to let the database tell you how to design the database. See Section 9 for details on testing.

Design Recommendations for High Performance Insert

Many of the techniques described in this book can be combined in the design of tables that can process an extremely high insert rate. Using these techniques has resulted in real-life insert rates as high as 13,000 per second, and application-wide insert rates as high as 17,000 per second with my next target set at 21,000 per second. Use the following combination of design ideas presented in this book to achieve ultra-high insert rates.

- Use range-partitioning to separate the table into partitions using a separation that can also be applied to the data that is input to the high volume application process. That data would then be fanned out and input to multiple parallel application processes that each act upon their own partition.
- Cluster the table to be inserted into by the clustering key of the data that is input to the process performing the inserts. If clustering is to be used, a proper quantity of free space and free pages should be maintained to preserve that clustering as data is added. Monitor the insert activity via the real-time statistics tables to determine the frequency of REORGs.
- As an alternative to clustering for the inserts, you can instead cluster for data retrieval and then utilize APPEND YES and/or MEMBER CLUSTER.
 - APPEND YES for pure insert only strategies.
 - MEMBER CLUSTER in a data sharing environment and for an insert algorithm that will slightly more aggressively look for free space versus APPEND YES.
 - Use both settings for high volume insert in a data sharing environment for a process that does not need to search for free space.
- Set PCTFREE and FREEPAGE to 0 if you are using MEMBER CLUSTER and/or APPEND YES.
- Set TRACKMOD NO if incremental image copies are not desired.
- Create only one index on the table.
- Use a larger index page size. I like 8K. If not using an ascending key, set up PCTFREE in the index such that there is never a page split between REORGs. Do not use FREEPAGE for the index in this situation.
- Make sure to use pre-formatting for the table space, especially if you are using MEMBER CLUSTER and APPEND YES.

In addition to these settings you can set up a REORG strategy that can accommodate the readers of this data. If you have clustered by something other than an ever-ascending key, then you need to REORG the table space to keep the data organized for efficient read access. Also, if you are not using an APPEND strategy for your table space (MEMBER CLUSTER or APPEND), you'll need frequent table space REORGs to avoid excessive free space search during the insert processing. You'll also want to develop an index REORG strategy such that you perform REORGs on the index at regular intervals. For indexes that do not have an ever ascending key, you'll want to REORG often enough so that there is never an index page split during the high volume insert processing.

Minimizing Change

Another technique for achieving a high level of performance from a physical design is to enact a policy of minimizing the amount of change that happens within the physical database. There are two industry wide trends that lend themselves to a proliferation of change in DB2 tables. The first is that there is a significant quantity of migration from legacy systems (flat files, VSAM, IMS, etc.) into the more modern relational database structure and DB2. The second is the use of a service-oriented architecture to manage large data stores in an enterprise environment.

Minimizing Change to Tables

In a legacy migration situation, many times records from files are simply converted to rows in tables. In addition, legacy enterprise applications may be retained, or worse yet, layered upon to avoid a lengthy application rewrite. While this strategy of minimal application impact will save time in a legacy migration scenario it can severely impact operation costs. In other words, porting your flat file to a relational database will get you a modern access method and backup/recovery support, but will cost you a significant increase in CPU consumption unless you make programming changes. In many cases management is looking for minimal operational cost along with minimal development cost (Ha!). In these situations one relatively easy thing to do is to attempt to minimize change to the tables involved. If the application performs a “full record replacement” strategy for applying changes, then it may be a benefit to identify exactly what changes most often in the tables affected and then separate that data into its own separate table. Careful analysis typically shows that most data in these sorts of processes does not change. This separation of data can save a lot of resources if a change detection process is also employed (see the next section for details).

I have in the past utilized encoding schemas to detect and avoid the application of non-changes to data. This typically involved identifying the data that does not typically change, separating the data that changes from the data that does not change, and then hashing the data values of the data that does not change into a value that can be used for change detection. This hash value is stored along with the key value of the data that does frequently change and then changes to the more stable data can be avoided by comparing the hash value of the incoming change with the hash value stored.

Figure E.

Original set of tables where all data changes all the time.

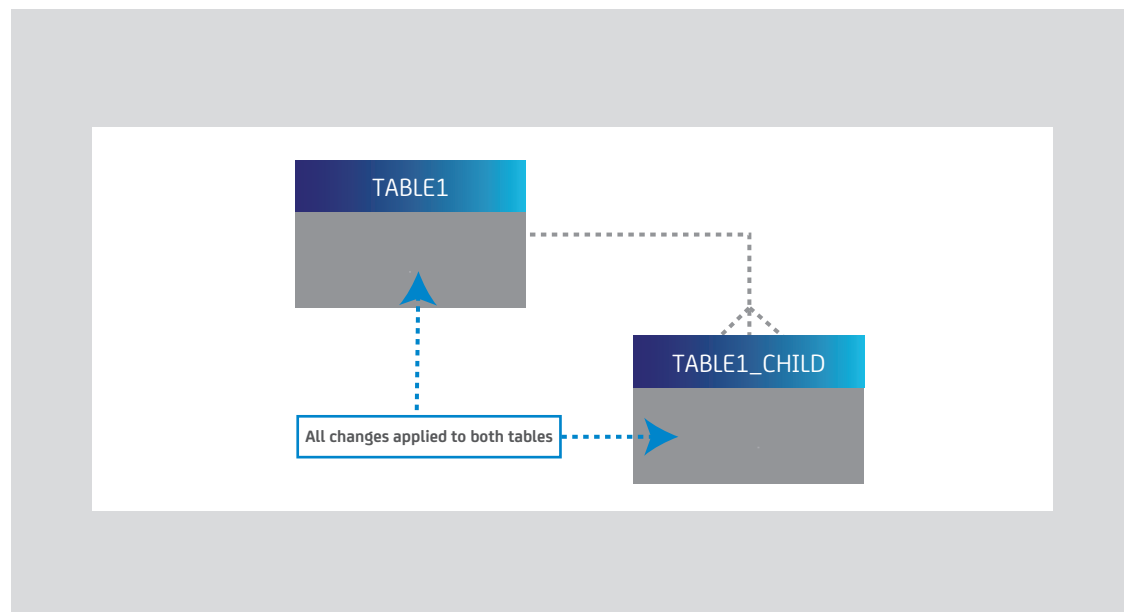
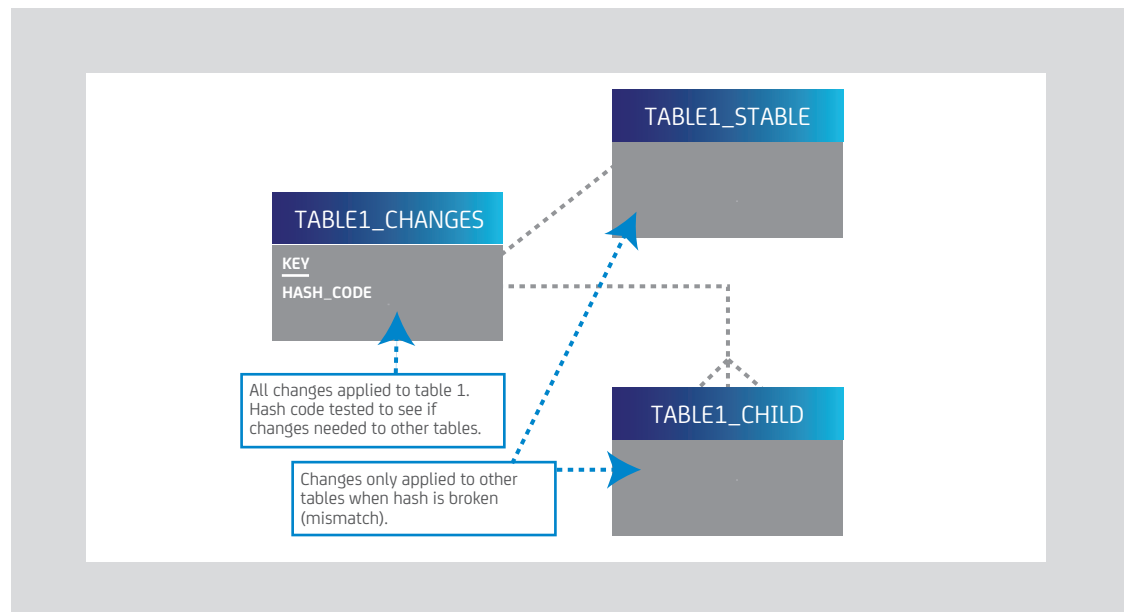


Figure F.

Table design with hash code to avoid changes to more stable data.



Reducing the Impact of Unknown Change

I like to think of today's service-oriented systems as large scale message processing systems. A request comes into a service to retrieve some data for a specific set of data. That data is retrieved in full whether or not all or a portion of the data is needed. Eventually the data may or may not come back in another request to change the data. Of course, there is no identification as to what data is needed and what data has changed. Typically in these types of implementations there is a vast quantity of SQL being issued as well as many unnecessary updates to the database. A combination of a change detection process along with some reduction in data access will go a long way in reducing the quantity of SQL issued. I also occasionally employ a method of caching commonly accessed, as well as recently accessed, data in an effort to avoid rereading of data. When incoming data is returned to a service for update, a comparison may be possible with cached values, or a change detection process (discussed above) using either hash codes or a direct comparison (especially when using a cache) can be employed to avoid rereading of data and unnecessary application of changes. This can also be combined with some "denormalization light" as is discussed in the next section below.

Denormalization

In the relational world, denormalization equals death. I'm not joking! I cannot count the number of times I have seen DBAs gathered in a room convinced that a design will not work due to too many table accesses and then spend months on a denormalization strategy without running any tests. Then when the application is deployed, access is terrible, and the data is a mess, they all throw up their arms and say "but we thought of everything"! A proper relational design can be a performance advantage in that it can be used to avoid update anomalies, reduce redundancy of data storage, minimize the impact of data change, and reduce the quantity of physical data access required for a process. You should use the relational design unless it is proven to introduce performance problems via a proof of concept test (Section 9).

I have at times subscribed to denormalization, but only when all other attempts at managing performance have failed. The types of denormalization I have applied, however, "does not subscribe" to the normal concepts.

Denormalization Light

This technique has been useful in large scale environments when there is a single parent table (a primary entity) surrounded by one or more level of child tables that contain optional data. In situations where a process most often requests all data for a given primary key, there may be many unnecessary table accesses to child tables that contain no data for the key presented. In a situation where the pattern of requests is truly random, there can be excessive random requests to many tables that most often return no data. If a test proves that this type of access will not meet a given service level agreement, I may elect to employ a technique that utilizes indicator columns in the primary entity that can be used to avoid access to child tables that contain no data. The downside is that any update process will have to maintain the indicator columns. The good news is that access to child tables with no data for a key can be avoided. The great news is that full denormalization can be avoided!

Figure G.

Original table design

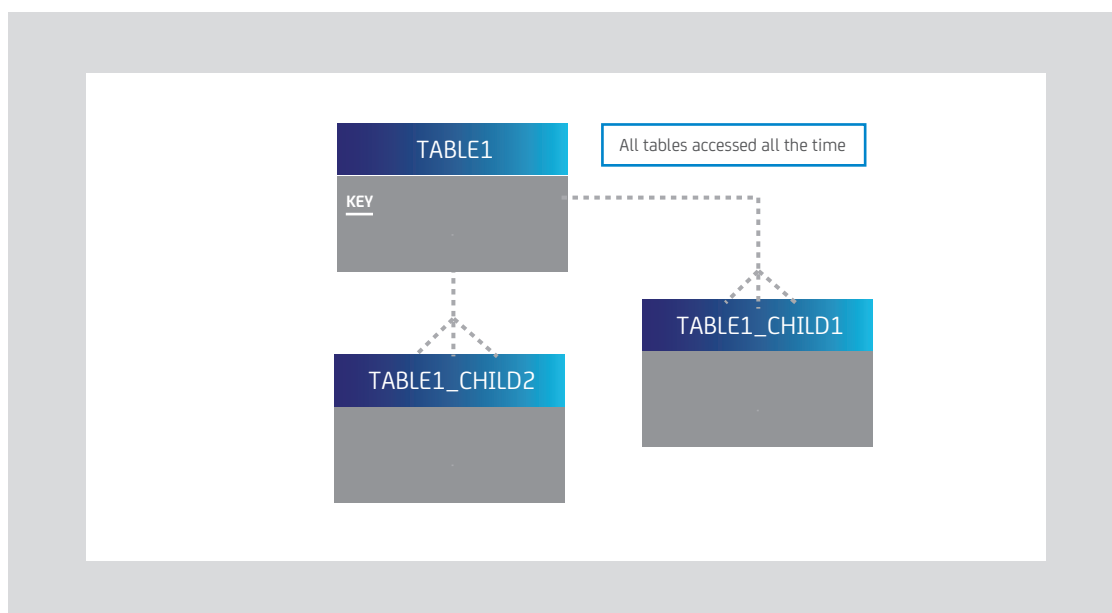
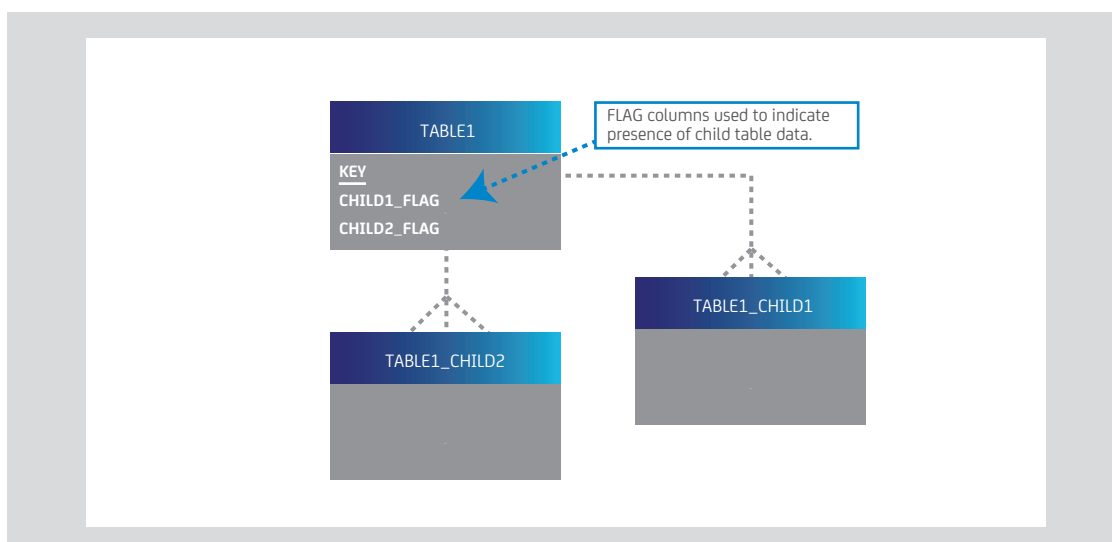


Figure H.

Design with indicator columns



In many cases, you can actually code for table access avoidance by utilizing DB2 during join predicates. In this case, DB2 will not join to the child tables unless the indicator column has a value that indicates the child data is present.

Original query that accesses all tables.

```
SELECT *
FROM   TABLE1 P
LEFT OUTER JOIN
      TABLE1_CHILD1 TC1
ON P.KEY = TC1.KEY
LEFT OUTER JOIN
      TABLE1_CHILD2 TC2
ON P.KEY = TC2.KEY
WHERE P.KEY = ?
```

Query that uses the indicator columns to avoid access to child tables with no data for the key.

```
SELECT *
FROM   TABLE1 P
LEFT OUTER JOIN
      TABLE1_CHILD1 TC1
ON P.KEY = TC1.KEY
AND P.CHILD1_FLAG = 'Y'
LEFT OUTER JOIN
      TABLE1_CHILD2 TC2
ON P.KEY = TC2.KEY
AND P.CHILD2_FLAG = 'Y'
WHERE P.KEY = ?
```

Full Partial Denormalization

One of the biggest challenges in complex environments is the performance of search queries. This book contains many recommendations that can be applied to improve the performance of search queries. However, sometimes the cost of a complex search that involves access to several tables can be just a little too expensive to meet an SLA. While denormalization is a possible solution, it can present many challenges and can actually increase costs rather than reduce them. In situations in which my back has been against a wall, I have successfully employed a technique that I like to call “full partial denormalization”. This technique has several components:

- An analysis of the data will hopefully show the source of the performance challenge. This is typically in the form of more common data values. These more common data values may actually only represent a portion of the full volume of data. For example, the name “Smith” versus the name “Luksetich”. I’m going to bet there are more of the first than the latter when searching on name. This common set of data can be combined into a set of common search key values that may or may not be spread across different tables. This set of keys are then identified and placed into a special “common key only” table.

- The “common key only” table is used to pull all of the related search data from all the other tables and is denormalized into a special “early out” search table. Remember that in order for this to be effective it has to represent only a subset of the entire quantity of data being searched. In one implementation I employed, there were 600,000 keys in the common key only table and 30,000,000 rows in the denormalized search table. The full quantity of search data represented about 1.1 billion rows. This is a very good ratio!
- Establish a set of triggers to maintain the “early out” search table values. The triggers use the “common key only” table in a WHEN condition and then perform the denormalization on the fly as the underlying data changes.
- Modify the search process to use an early out strategy. The input arguments are used to match against the “common key only” table. If there is a match then the denormalized table is searched. If there is no match then the normal search query is executed.

Always test such design ideas to see if they will perform in your specific situation! I have also employed more simplified “early out” techniques for searches where one query was used to satisfy the requests using the most common input arguments, and a second more versatile query was executed for all other requests.

Section 7:

Memory, Subsystem Parameters and Performance

I’m a huge subscriber to the fact that 80% of the performance to be gained is at the application and database level. However, you at least need to have a minimally decent subsystem configuration, so this section will address some of the “big button” subsystem settings. As with any database configuration, available memory is a big issue and its proper configuration and use is a key factor to overall system health. There is a huge amount of system performance tuning information available in the DB2 Administration Guide, the DB2 Installation Guide, and the DB2 Performance Guide, as well as in various articles, blogs, and presentations on the internet.

Buffers

The DB2 buffer pools might be the single biggest subsystem level performance tuning button that you can push. When in doubt and you have the memory and not the time to spare, bigger is better. If you have the time to spare, then an intelligent buffer configuration is definitely something that will benefit both application and subsystem health. Here are some buffer recommendations:

- **DB2 system catalog and directory.** These are not in their own buffer pool sized such that most or all of it will be memory resident? Really? Then make it so! Do not mix anything with the catalog and directory!
- **Workfile database.** I always put the workfile database in its own buffer pool. In addition I recommend that you separate the workfiles from the temporary table storage and place them each in their own buffer pool. The sequential percentage should remain at 80% due to the fact that there will be some random access to workfiles via sparse indexes and when there are indexes defined on the temporary tables. Deferred write thresholds should be set high because you want to keep as

much of your workfiles and temporary tables in memory as possible. This pool should be relatively large, but you may want to monitor just how much you are writing to DASD and balance the memory consumption here with support for the occasional large sort.

- **Indexes and tables.** You should always separate indexes from tables in buffer pools since the access patterns are quite different.
- **Random and sequential.** Objects that are accessed primarily sequentially should be in their own buffer pools since one large sequential read could wipe out a lot of cached objects. In addition, a very frequently accessed table or index that has an extremely random access pattern should be separated from other objects that have a high level of re-referenced pages. You can use the DISPLAY BUFFERPOOL command to watch these access patterns and do the proper separation.
- **Code tables.** These should be in their own buffer pool with enough memory allocated as to avoid any I/O. Monitor the access to this pool carefully to see if there is any opportunity for applications to cache code values locally if it appears they are going to these tables way too often. Monitoring this pool closely will also help you identify tables that really shouldn't be in this pool (sequential access or frequent updates) as they can seriously impact the efficiency of this special cache.
- **High volume update objects.** These objects most certainly need to be in their own buffer pool. The vertical deferred write threshold should be set to 0,40 in order to trickle the writes to DASD. You do not want writes to be triggered on system checkpoints for these objects.

Page fixing is highly recommended and you should use it whenever possible. If the DB2 subsystem is the primary subsystem on an LPAR, then you should not be sharing the memory it uses with anything else. Therefore page fixing should be a non-issue as it will save significant CPU resources used to manage the I/O for your database objects.

Although monitoring the buffer hit ratio can be a productive way of tuning the buffer pools, you really should be monitoring the number of I/O's per second on a pool-by-pool basis. The goal of buffer tuning is to reduce the I/O activity. Monitoring the I/O rate is the true way to measure success or failure of buffer pool tuning. High performance DASD can be our friend and our enemy when it comes to buffer pool tuning. It is great to throw tons of memory at the DB2 buffers, but only if it really is going to make a difference. If the memory can be used elsewhere, it should be. With that in mind, you should not only be monitoring the I/O rate when tuning the buffers but also the random and sequential response wait time per I/O. You can reduce the I/O's, but if the response time per I/O is going up while the I/O rate is going down, then you've probably reached a balance between your DB2 buffer caching and the DASD cache. At that point it may be time to use the memory for other buffer pools or for other DB2 (other subsystems on the LPAR) resources.

Logging

The DB2 logs are a central resource and are critical for performance. The active logs should be sized large enough to map to the available logical DASD devices (I prefer 2GB active log size, but am not committed to it) and the logs should be split (you are dual logging of course) onto separate DASD subsystems. I have never heard of anyone getting any benefit from striping the logs so don't even consider doing it. The default log buffer size, in my opinion, is way too small so make sure to increase it. There should be enough active logs such that rollbacks (and replication tools) never have to go to an archive log. You should also have a policy in place such that archive logs initially go to DASD and move to tape once everything, especially the system catalog and directory, have been image copied.

DASD and I/O

High performance DASD subsystems are making us all look like geniuses these days. If your enterprise is serious about performance, then you have invested in large scale, high performance DASD subsystems with the largest amount of cache available and parallel access volumes enabled. If this is the case in your environment, then you should spend a minimal amount of time worrying about physical storage. While separating DB2 logs, bootstrap database, and system catalog and directory from the other database objects is important, there is little concern in separating indexes from table spaces and high performance objects from low performance objects except for extreme cases. The logical devices can be made large and should be extended addressability enabled. I typically set PRIQTY and SECQTY to -1 except for very high performance objects and no longer worry much about extents unless the number gets very large (see my RTS queries in Section 2 of this book for an example).

Big Impact Subsystem Parameters

What follows is a list of subsystem parameters that you need to consider in a high performance environment.

Name: ACCUMACC

Description: DB2 accounting records are rolled up per user for DDF & RRSAF threads.

Possible Values: No, 2 - 65535

Default Value: 10

Performance Implications: Since a DB2 accounting record is produced whenever a DDF thread goes inactive (or with sign-on for RRSAF threads), less overhead is incurred if this option is used. Using the default of 10 will roll up every 10 accounting records into 1 SMF 101 record - this can be helpful in reducing the number of DB2 accounting records (SMF 101). A typical production environment may set ACCUMACC to 10, or 100, or higher to reduce the number of accounting records produced. Non-prod environments may set ACCUMACC to 100 or higher since detail accounting records aren't regularly reviewed in these environments. There is a trade-off here between the quantity of SMF records produced and the ability to produce a thread-by-thread accounting trace report. I like 1,000 as a production environment value for high volume environments. However, in a data sharing environment you can set one member to 1 in order to have the ability to get a detailed trace report if desired.

Name: AUTHCACH

Description: Default authorization cache size per plan if CACHESIZE is not specified on the BIND PLAN command.

Possible Values: 0 - 4096

Default Value: 3072

Performance Implications: If you specify 0, or a number too small to contain all the authorization ids which can concurrently execute the PLAN, a security check (or read) to the DB2 catalog (SYSPLANAUTH) will be required for each plan execution per user. The default size should be sufficient, but to determine the storage you will need, use this formula: 8 bytes times the number of concurrent users (userids) plus 32 bytes of overhead.

Name: CACHEDYN

Description: Whether prepared dynamic SQL statements are to be cached for future use

Possible Values: YES, NO

Default Value: YES

Performance Implications: If a dynamic SQL statement is already (prepared) in the statement cache, a prepare is not performed on the statement, improving performance. Also, the SQL statements need to use parameter markers instead of host variables in order to be eligible for cache statement reuse.

Name: CACHEDYN_FREELOCAL

Description: Allows DB2 to free cached dynamic statements if DBM1 is using too much below the bar storage

Possible Values: 0,1

Default Value: 1

Performance Implications: The default should be used just in case DBM1 uses too much storage below the bar for cached dynamic SQL statements – DB2 will free some cached dynamic SQL statements.

Name: CACHEPAC

Description: Storage needed in DBM1 for caching package authorizations.

Possible Values: 0 – 10M

Default Value: 5M

Performance Implications: Caching authorization ids, which are authorized to execute a package, improves performance by eliminating reads to the DB2 catalog (SYSPACKAUTH).

Name: CACHERAC

Description: Storage needed in DBM1 for caching routine (Stored Procedures, User-defined functions) authorizations.

Possible Values: 0 – 10M

Default Value: 5M

Performance Implications: Caching authids, which are authorized to execute routines (Stored Procedures or User-defined Function), improves performance by eliminating reads to the DB2 catalog (SYSROUTINEAUTH).

Name: CDSSRDEF

Description: This is the default value for the CURRENT DEGREE special register for dynamically prepared SQL statements when SET CURRENT DEGREE is not specified.

Possible Values: 1, ANY

Default Value: 1

Performance Implications: In almost all cases, the default of 1 should be used to disable DB2 query parallelism for dynamic SQL statements. Use query parallelism only in cases where you are sure parallelism will help – for example, with long running, CPU-intensive, data warehouse type queries. This value can be overridden with SET CURRENT DEGREE statement.

Name: INDEX_IO_PARALLELISM

Description: For tables with multiple indexes, allow insert processing on multiple indexes in parallel

Possible Values: YES, NO

Default Value: YES

Performance Implications: Can reduce in-DB2 elapsed time for insert operations on tables with multiple indexes. The insert processing will be performed concurrently on the indexes.

Name: CHKTYPE, CHKREQ, CHKLOGR, CHKMINS

Description: These subsystem parameters are used to control the frequency of system checkpoints.

Possible Values: LOGRECS, MINUTES, BOTH, or SINGLE

Default Value: MINUTES

Performance Implications: For subsystem efficiency, log checkpoints should not be too frequent or too far apart. If your log checkpoint is too frequent, DB2 can consume unnecessary resources. If log checkpoints are too far apart, DB2 recovery can be elongated. Checkpointing every 5 to 15 minutes is a pretty good idea. Remember that there is quite a bit of activity that is triggered upon a checkpoint so I lean more towards the 15 minutes and adjust the vertical deferred write threshold in the buffer pools to trickle writes for high volume update objects.

Name: CMTSTAT

Description: Allows DBATs to be marked INACTIVE after commit (or rollback) if they hold no cursors, or temp tables.

Possible Values: ACTIVE, INACTIVE

Default Value: INACTIVE

Performance Implications: CMTSTAT enables INACTIVE DBATs to be returned to the pool for reuse and disconnects them from the active connection. This allows the best use of DBATs and remote connections to support large numbers of distributed connections. The default should be used in 99% of situations. The only reason to set it to ACTIVE would be in super high volume, remote batch environments and only if high performance DBATs are not possible.

Name: COMPACT

Description: Compresses data written to archive logs on tape

Possible Values: NO, YES

Default Value: NO

Performance Implications: Compressing archive log data on tape reduces tape usage, but your tape devices need to support IDRC (improved data recording capability). In a high performance environment it is quite typical to archive to DASD and then eventually move the logs to a tape or virtual tape device.

Name: CONDBAT

Description: Maximum number of concurrent DDF connections

Possible Values: 0 - 150000

Default Value: 10000

Performance Implications: CONDBAT needs to be greater than or equal to MAXDBATs. If CONDBAT is reached, connection requests will be rejected, subsequently impacting your distributed applications.

Name: CONTSTOR

Description: Contract thread storage after commit instead of at thread deallocation.

Possible Values: YES, NO

Default Value: NO

Performance Implications: If your DBM1 address space is under virtual storage constraints, set CONTSTOR to YES to free up unused thread storage. If DBM1 is not storage constrained, the default (NO) is recommended. Setting this value to YES will consume more CPU, so it really should be set to NO unless you have a real memory problem. In addition, in DB2 10 for z/OS, almost all thread storage is above the 2 gig bar making this parameter even less necessary.

Name: CTHREAD

Description: Total number of concurrent allied threads to a DB2 subsystem.

Possible Values: 1 - 20000

Default Value: 200

Performance Implications: If CTHREAD is exceeded, all new thread requests are queued, causing applications to wait. CTHREAD is the total of all concurrent TSO users, batch jobs, CICS & IMS threads, utilities, and DDF connections. Keep in mind utilities can create multiple threads due to parallel tasks which count towards total CTHREADs. Also, query parallelism creates parallel tasks for queries which are counted toward CTHREAD totals. It's human nature to open the door and let everyone in, but this parameter really needs to be balanced with available resources.

Name: EDM_SKELETON_POOL

Description: Minimum storage needed in the EDM pool at DB2 start up to hold skeleton package tables.

Possible Values: 5120 – 2097152 KB

Default Value: 10240

Performance Implications: Since the EDM Skeleton pool is storage above the 2 gig bar, the setting can be liberal. However, if your DB2 subsystem uses all the EDM_SKELETON_POOL storage, package allocations will fail with a -904 resource unavailable.

Name: EDMDBDC

Description: Minimum storage needed in the EDM pool at DB2 start up to hold DBDs (Database Descriptors).

Possible Values: 5000 – 2097152 KB

Default Value: 23400

Performance Implications: The DBD cache in the EDM pool is storage above the 2 gig bar which allows you to be generous with the size. If your DB2 runs out of EDM pool DBD cache (EDMDBDC), DBDs will not load into memory resulting in -904 resource unavailable errors. In DB2 Data Sharing, EDMDBDC should be increased on each member due to DB2 storing duplicate copies of DBDs in each member. A member can have multiple copies of a DBD loaded in its EDMDBDC until all threads using that DBD are deallocated.

Name: EDMPOOL

Description: Maximum storage needed for the EDM pool below the 2 gig bar.

Possible Values: 0 – 2097152 KB

Default Value: 0

Performance Implications: EDM pool storage is critical for the operation & performance of a DB2 subsystem. In DB2 10 for z/OS, the EDM subpools (DBD, Skeleton, and Statement Cache storage pools) are above the 2 gig bar. However, EDM pool storage below the 2 gig bar is needed for plan and package allocations that were bound in a prior release until all plans and packages are rebound under DB2 10 for z/OS. The recommendation is to use the same EDM pool size defined for your DB2 9 for z/OS subsystem. Once rebound under DB2 10 for z/OS, plans and packages will be loaded above the 2 gig bar upon allocation. Also, after all plans & packages have been rebound under DB2 10 for z/OS, the EDMPOOL size can be reduced. Moving these structures about the bar frees up thread storage below the bar in DBM1 allowing for larger numbers of concurrent threads.

If any of the EDM pools become full, page stealing of unused pages from any of the pools begins in order to satisfy new requests. Sizing these pools correctly allows DB2 to be more efficient and reduces application elapsed time. Whichever structures can't be found in these pools forces reads (I/Os) to the DB2 catalog and directory in order to load them. In addition, costly and time consuming prepares of dynamic SQL statements can result if your EDM Statement Cache is too small.

Name: EDMSTMTC

Description: Minimum storage needed in the EDM pool at DB2 start up for (dynamic) prepared SQL statements.

Possible Values: 5000 – 1048576 KB

Default Value: 113386

Performance Implications: The EDM Statement Cache (also called Dynamic Statement Cache – DSC) is above the 2 gig bar allowing for generous allocation sizes. The best practice is to make the EDM Statement Cache large enough to hold most of the frequently executed dynamic SQL statements. If the dynamic SQL statement is found in the cache, DB2 can avoid costly and time consuming SQL statement prepares. Be aware that if you make this pool extremely large, it can increase the cost of searching for a statement, which will impact the performance over all dynamic statements, so there really needs to be a balance between the hit rate and overall performance. Make changes slowly.

Name: IDBACK and IDFORE

Description: Number of concurrent foreground (or TSO) and background (or batch) allied threads connected to a DB2

Possible Values: 1 - 20000

Default Value: 50

Performance Implications: If this number is exceeded, the thread allocation request is rejected. This is something to balance between the other parameters that control the number of threads and probably only an issue in very high volume environments.

Name: IDTHTOIN

Description: Idle thread timeout value for distributed access threads (DBATs)

Possible Values: 0 - 9999

Default Value: 120

Performance Implications: This value determines how long DB2 (server) waits for a client response after sending a result before it cancels the clients DBAT (thread). In most cases, DB2 is waiting for a request from the client for DB2 to send more results (rows) or DB2 is waiting for the client to signal it is ready to commit. But if the client isn't responding, a DBAT is being held and can be holding locks on DB2 resources. This reduces concurrency and limits the number of DBATs available to other clients. Once the Idle Thread Timeout value is reached, the client's thread is cancelled.

Name: MAXDBAT

Description: Maximum number of distributed access threads connected to a DB2

Possible Values: 0 - 19999

Default Value: 200

Performance Implications: If MAXDBATs is reached, the next DBAT request is queued and will wait for an available DBAT. It is a good practice for distributed applications to connect to the database (data source), perform its work, issue a commit (don't rely on Autocommit), and disconnect from the database. When the application issues the disconnect, the DBAT becomes an Inactive DBAT (provided CMTSTAT=INACTIVE) and releases all DB2 resources. In addition, the DBAT becomes disconnected from the connection and the DBAT is returned to the pool for reuse. The DBAT will stay inactive in the pool for the time specified on the POOLINAC subsystem parameter, or until reuse. Some distributed applications may require a constant (distributed) connection which will reduce the number of DBATs available for reuse.

Many organizations classify their DDF workload in WLM Service Classes which have service periods. A DDF Service Class with service periods means that after a DBAT uses 'x' number of z/OS Service Units, the priority of the work through that DBAT (and Service Class) will be lowered. The result is that the workload running through that DBAT will run slower in DB2, holding locks longer and reducing concurrency. Be careful with DBAT reuse in this manner since the WLM classification doesn't get reset until the DBAT is marked Inactive (at disconnect time).

DB2 10 for z/OS introduced High Performance DBATs which is the equivalent of DBAT using RELEASE(DEALLOCATE). High Performance DBATs have the same characteristics as CICS or Batch plans bound with RELEASE(DEALLOCATE), which means concurrent DDL changes, Binds, and Utilities could encounter locking contention. See Section 3 for more information on high performance DBATs.

Name: MAXKEEPD

Description: Number of prepared dynamic SQL statements kept past a commit in the dynamic statement cache for Plans & Packages bound with KEEPDPYDYNAMIC(YES).

Possible Values: 0 - 65535

Default Value: 5000

Performance Implications: Keeping prepared dynamic SQL statements in cache past commits helps keep prepare time and costs down for local (static packaged with dynamic SQL) and distributed applications. Distributed applications using CURSOR WITH HOLD benefit from this option since their prepared SQL statement will still be in cache after the commit.

Name: MAXRBLK

Description: The amount of storage needed for RID processing

Possible Values: 9, 128 – 10000000 KB

Default Value: 400000

Performance Implications: Access paths which require RID processing use this storage area. Since this storage is above the 2 gig bar, allocations can be generous. If too many RIDs qualify for RID processing, or too much of the RID pool is needed for RID processing of an access path, DB2 may change the access path to a table space scan at run time or overflow the RID list to a workfile. You can specify that DB2 uses workfiles to continue RID list processing when the RID pool is not large enough by specifying a MAXTEMPS_RID value.

Name: MAXTEMPS_RID

Description: The maximum amount of temporary storage in the workfile database that a single RID list can use at a time. Possible Values: NONE, NOLIMIT, or 1 to 329166

Default Value: NOLIMIT

Performance Implications: I'm a fan of the default, but you may want to adjust this down if you have an extreme amount of large RID list processing.

Name: MAXRTU

Description: Maximum number of tape units DB2 can use to read archive log tapes

Possible Values: 1 - 99

Default Value: 2

Performance Implications: Choosing the proper setting for this parameter can be involved. If you are in Data Sharing, other members can't use the tape drives allocated to another member until the DEALLOC is reached. So you may want to submit all recovery jobs from the same member. Try to set the number as high as possible, leaving a few tape drives available for other work. For example, if there are 10 tape drives available to DB2, set this parameter to 6 or 8. You don't want to impact archive log processing if all the drives are taken by recovery processes reading archive logs. Also, if your installation has Change Data Capture or other replication products that can read DB2 archive logs if it falls behind, a higher number may be necessary for MAXRTU.

Name: MAXTEMPS

Description: Maximum amount of temporary storage in the workfile database any one user (thread) can use.

Possible Values: 0 – 2147483647 MB

Default Value: 0

Performance Implications: MAXTEMPS limits the impact when just a few users consume a large amount of the temp workfiles for large sorts, temp tables, or scrollable cursors. By limiting how much workfile storage a user can use, normal transaction workloads are less likely to encounter contention for workfiles.

Name: WFDBSEP

Description: Specifies whether DB2 should provide an unconditional separation of table spaces in the workfile database based on the allocation attributes of the table spaces.

Possible Values: YES, NO

Default Value: NO

Performance Implications: I like separating the workfile utilization for sorts from temporary table usage. If you specify YES, DB2 will separate the use of the workfile table spaces depending upon whether or not you specify SECQTY in the table space allocation. This will enable you to split the usage, allocate different sizes and buffer pools for temp table usage versus pure workfile usage which is a definite plus for a high volume environment.

Name: WFSTGUSE_AGENT_THRESHOLDP

Description: This parameter determines the percentage of available space in the workfile database on a DB2 subsystem or data sharing member that can be consumed by a single agent before a warning message is issued.

Possible Values: 0 to 100

Default Value: 0

Performance Implications: I can't tell you the number of times people ask about how much workfile is being consumed by a particular process. Setting this to something other than the default will help point out the processes that are over utilizing the workfile database.

Name: WFSTGUSE_SYSTEM_THRESHOLD

Description: This parameter determines the percentage of available space in the workfile database on a DB2 subsystem or data sharing member that can be consumed by all agents before a warning message is issued.

Possible Values: 0 to 100

Default Value: 90

Performance Implications: 90 may be too small and you may want to set this lower if you have many out of control sorts and want to get information about when they are running and when your workfile database is under stress.

Name: MINSTOR

Description: Have DB2 invoke storage management techniques to reduce a thread's storage.

Possible Values: YES, NO

Default Value: NO

Performance Implications: I would only recommend turning this on if your DBM1 is under storage constraints, or if IBM recommends it. Otherwise, DBM1 can consume more CPU trying to manage/ reduce thread storage.

Name: NUMLKTS

Description: Total number of locks a user can have on one table space before lock escalation occurs

Possible Values: 0 - 104857600

Default Value: 2000

Performance Implications: Applications want to avoid hitting this threshold. When lock escalation occurs on an object, concurrency goes down and timeouts and/or deadlocks can occur. If an application hits this threshold consistently, it could be that the application isn't committing

frequently enough, or they are using the wrong Isolation level, or lock size (page or row). In addition, the storage usage increases (540 bytes per lock) due to all the locks being held. Some (CRM) applications may require a higher number because of many tables in one table space, or because they have very large units of work. Increasing the NUMLKTS may require increasing IRLM region size.

Name: NUMLKUS

Description: Total number of locks a user can hold across all table spaces concurrently.

Possible Values: 0 - 104857600

Default Value: 10000

Performance Implications: Some applications may acquire more than 10,000 locks in a unit of work. Therefore, it may be necessary to raise the NUMLKUS value. But, increasing this number has its risks, too. More storage will be required as more locks are obtained. More locking could reduce concurrency and increase timeouts and/or deadlocks. In Data Sharing, more locks mean more lock propagation to the CF, which increases overhead and path length.

Name: OPTHINTS

Description: Enables the use of optimization hints during the bind process to influence access path select for static and dynamic SQL statements.

Possible Values: YES, NO

Default Value: NO

Performance Implications: Sometimes it is necessary to “help” the DB2 engine choose a more desirable access path. These situations should be less and less frequent due to improvements in DB2, but are certainly needed from time to time. The recommendation is to use OPTHINTS only in special situations and not as a standard. OPTHINTS is best used when the DBA knows exactly what access path is best and how to specify the proper hints to get the Optimizer to generate that access path. It is up to the DBA to manage the optimization hints once they are used.

Name: OUTBUFF

Description: Size of the log output buffers.

Possible Values: 400 – 400000 KB

Default Value: 4000

Performance Implications: DB2 page fixes the log output buffers in real memory. The larger the log output buffer the greater the likelihood a log record remains in memory, which can reduce physical log reads. DB2 statistics trace records can show how many times a log write had to wait due to a full log buffer. Too many waits due to log buffer full conditions would be an indication the log buffer is too small. I personally feel that the default is too small and recommend making it larger as this is a critical shared resource.

Name: PARA_EFF

Description: Set the parallelism efficiency DB2 assumes when determining parallelism access paths

Possible Values: 0 - 100

Default Value: 50

Performance Implications: Since most installations are not set up for achieving optimally balanced parallel tasks, the recommendation is to use the default setting. Many variables come into play when DB2 is determining parallelism access paths, and unless you have a very controlled environment, changing this setting may or may not provide the results you want.

Name: PARAMDEG

Description: Maximum number of parallel tasks (degree) DB2 will allow at run time.

Possible Values: 0 - 254

Default Value: 0

Performance Implications: Because this sets the limit of parallel tasks in a parallel group, it overrides the degree parameter specified at Bind time. This can be helpful for controlling the number of parallel tasks in a query instead of allowing DB2 to determine the number of tasks. The number of available general processors (GPs) and available zIIPs (Specialty Engines – SEs) needs to be considered when setting this number. A good rule of thumb is 2 times the number of engines (GPs + SEs). Notice that the default is zero. If you want to take advantage of parallelism, you're going to have to set this parameter first.

Name: POOLINAC

Description: Time inactive DBATs stay in the pool before being destroyed.

Possible Values: 0 – 9999 seconds

Default Value: 120

Performance Implications: It is important for distributed applications to disconnect from the database (data source) after committing so the DBAT becomes inactive and is returned to the pool for reuse.

Name: PTASKROL

Description: Roll up parallel task accounting data into the parent task accounting record.

Possible Values: YES, NO

Default Value: YES

Performance Implications: This will reduce the number of SMF 101 accounting records produced when using parallelism. All of the parallel task information will be rolled up into the parent task accounting record.

Name: SMFCOMP

Description: Tells DB2 to compress trace records (SMF100, SMF101, & SMF102) written to SMF.

Possible Values: OFF, ON

Default Value: OFF

Performance Implications: This will reduce the size of DB2 SMF records and save SMF record processing time. It's a pretty good idea to turn this on in a high performance environment.

Name: SRTPOOL

Description: This is the size of the Sort Pool storage.

Possible Values: 240 – 128000 KB

Default Value: 10000

Performance Implications: The larger this pool is, the more efficient sort processing will be, and more concurrent sorts can be supported. When an order by or distinct is needed in a query, sorting the data in memory (or the Sort Pool) is much faster than if workfiles need to be used. If you have the memory to spare, I recommend increasing this value, as it can have a serious impact on moderately sized sorts.

Name: STORTIME

Description: Time to wait for a Stored Procedure or UDF to start.

Possible Values: NOLIMIT, 5 - 1800

Default Value: 180

Performance Implications: The recommendation is not to use NOLIMIT because your application could wait indefinitely if the SP address space is down. There could be a variety of reasons causing your app to wait and the longer you wait the more likely you are to cause other timeouts and/or deadlocks within DB2.

Name: TCPKPALV

Description: Overrides the TCP/IP value for keeping a TCP/IP session alive.

Possible Values: DISABLE, ENABLE, or 1 - 65534

Default Value: 120

Performance Implications: You don't want TCP/IP to cancel your communication sessions between your clients and DB2. This value should be close to, or a little higher, than the Idle Thread Timeout (IDTHTOIN) value. Too small a number causes excessive network traffic as TCP/IP would probe more often to check if the connections are still alive.

Name: UTSORTAL

Description: Should the DB2 utilities dynamically allocate the sort work datasets.

Possible Values: YES, NO

Default Value: YES

Performance Implications: Allowing the DB2 utilities to allocate the sort work dataset increases the chances of parallelism in the utilities (CHECK, LOAD, REBUILD, REORG, and RUNSTATS). UTSORTAL can be set to YES if IGSORTN is set to YES. Different combinations of UTSORTAL, IGSORTN subsystem parameters, along with the SORTNUM utility control card will determine if the Sort utility or the DB2 utility will allocate the sort workfiles.

Name: IGSORTN

Description: Should the DB2 utilities ignore the SORTNUM control card.

Possible Values: YES, NO

Default Value: NO

Performance Implications: UTSORTAL can be set to YES if IGSORTN is set to YES. Different combinations of UTSORTAL, IGSORTN subsystem parameters, along with the SORTNUM utility control card will determine if the Sort utility or the DB2 utility will allocate the sort workfiles.

zIIP Engines

In brief, please keep in mind that the zIIP secondary engines are a capacity and licensing issue, not a performance feature. CPU time is CPU time and you should be tuning the queries to reduce CPU time. When monitoring, be sure to add the secondary engine CPU time from the accounting reports to the CPU time as they are accounted separately and the proportion of time spent on each type of processor will vary from execution to execution.

Section 8:

Monitoring Subsystem and Application Performance

Monitoring is really the key to designing, understanding and maintaining a high performance implementation in DB2 for z/OS. Without monitoring you have nothing! There has to be proof of the cost of an application, and there needs to be a way to compare the before and after impact of making a change in an effort to improve performance. While EXPLAIN gives us a great way to predict access patterns and cost, it should never be the “be all end all” for performance analysis. Monitoring is the only definitive proof of performance.

There are many tools and techniques available for monitoring. What you have at your disposal is dependent upon the tools on hand and management rules as to what level of monitoring is allowed. These tools can be online and batch operated and I have found that no matter how many tools you have at your disposal, there is always something useful in each tool, even if there are redundancies.

- DB2 accounting traces
- DB2 statistics traces
- DB2 dynamic statement cache
- Online database and application monitors
- Batch trace reporting tools

The cost of running the recommended statistics and accounting traces is quite low and they pay for themselves many times over in the savings realized by interrogating the output. Of course, if you have no policy for reviewing trace output then you don't have to collect it! Then again, if you don't have a policy of reviewing trace output, you have no concern for performance and probably aren't reading this book anyway!

DB2 Traces

Without DB2 traces we have limited ability to collect valuable performance information unless we have very specialized tools. DB2 traces can monitor all sorts of activities within a subsystem depending on the types of traces and levels of traces set. The basic component of a trace is something called an instrumentation facility component identifier (IFCID). IFCIDs are grouped into classes and classes can be grouped into trace types. There are several types of traces, but I'll only talk about statistics, accounting, and performance traces in this section.

Traces can be started by default via subsystem parameters, but they can also be turned on and off, as well as adjusted, via DB2 commands.

Statistics Trace

A statistics trace collects performance metrics at a subsystem level. The trace records produced are typically directed towards system management facility (SMF) datasets and hopefully your friendly system programmer is externalizing these SMF records for DB2 subsystems into separate datasets for analysis. Statistics traces are relatively very low cost and the data these days is typically externalized

every 1 to 10 minutes. By default DB2 starts statistics trace classes 1,3,4,5, and 6. The information collected is subsystem-wide and contains the following information:

- Class 1 = DB2 basic statistics
- Class 3 = exceptions like deadlock, timeout, datasets extents
- Class 4 = distributed activity
- Class 5 = data sharing related statistics
- Class 6 = storage usage statistics

Accounting Trace

The accounting trace collects thread level performance metrics and is really an essential key to understanding and improving application performance. Like the statistics trace records, the accounting trace records produced are typically directed towards system management facility (SMF) datasets and again, hopefully your friendly system programmer is externalizing these SMF records for DB2 subsystems into separate datasets for analysis. Accounting traces are very low cost traces which may add up to 15% overall CPU overhead to a subsystem (your results can vary dramatically). The accounting trace data these days is typically externalized upon the termination of each individual thread. So, you get one record per thread. This means that for large scale batch jobs, CICS transactions that use protected threads, distributed threads with ACCUMAC subsystem parameter set, or high performance DBATs, you can get multiple transactions bundled together into the same accounting trace record. By default, DB2 only starts accounting trace class 1, which I believe is seriously not enough to accurately diagnose and track the performance of applications. My recommended accounting trace classes to set are 1,2,3,7, and 8.

- Class 1 = Total elapsed time and CPU used by a thread while connected to DB2 at the plan level.
- Class 2 = Total elapsed time and CPU used by a thread within DB2 at the plan level. This is a subset of class 1. The elapsed time is broken into suspension time and CPU time.
- Class 3 = Total time waiting for resources in DB2 at the plan level. This is a subset of the class 2 time and is equal to the class 2 suspension time.
- Class 7 and 8 = These are, respectively, the class 2 and 3 times divided between packages utilized during thread execution.

Performance Trace

Performance traces are not set by default and can be used to track specific threads or sets of threads at an extremely detailed level. I typically start performance traces at a very specific level using a DB2 command to collect information for a problem application for which other methods of diagnosis and tuning has fallen short. You run a performance trace in situations in which you have to look for specific events within the execution of an application. I am typically using a performance trace to get detailed performance metrics for individual SQL statements in static embedded applications, buffer access patterns, excessive index access, and other very detailed metrics. I also use a performance trace in conjunction with a performance monitor tool. Given the advances in buffer pool monitoring, advanced tooling, and the capabilities to externalize the dynamic statement cache, I do find myself relying on performance traces less and less these days.

Running a performance trace is a very expensive proposition and should only be done in a controlled and/or understood situation. You'll want to limit the trace as much as possible to the thread of interest.

IFCID 316 and 318

These are the IFCIDs that need to be set in order to collect performance metrics within the global dynamic statement cache. Given the proliferation of dynamic SQL these days it is quite critical in my opinion to collect this data routinely, not as an exception. These traces are not turned on by default but can be turned on via the following command.

```
-START TRACE(ACCTG) IFCID(316,318)
```

There has been some debate over the overhead of running this trace. The fact of the matter is that a number of years ago a redbook was published that stated a number, something like 4% CPU overhead subsystem wide. I can guarantee that this number has been greatly exaggerated. In every customer site where I have turned this on, as well as every DBA I have interviewed, there has been no discernible increase in CPU consumption as a result of turning on this trace! Besides, the benefits of the trace far outweigh any potential impact, if an impact can be detected. If you don't believe me, please listen to a podcast I did with two of the most famous IBM DB2 professionals in the world and see what they have to say!

<http://www.db2expert.com/index.php?page=podcasts#S1P6>

Once this trace is activated you can use the techniques described in Section 4 of this book to externalize the information generated.

Understanding and Reporting Trace Output

In order to interpret trace output you must either be a really good programmer, or use a tool. If you are a clever programmer, you can find the macros that lay out the DB2 SMF record formats in the SDSNMACS DB2 library in members DSNDQWST for statistics trace records type 100, and members starting with DSNDQW for accounting trace records. Good luck! If you really don't have a tool that will read and format DB2 accounting and statistic trace records there is a "poor man's" tool that I have never used, but have recently heard about called DSN1SMFP. Information can be found at this link.

<http://www-01.ibm.com/support/docview.wss?uid=swg1PM94545>

With a tool, various reports can be produced and analyzed in order to determine whether or not there are resource constraints or bottlenecks at the subsystem, data sharing group, or application level.

In a statistics report, I generally start with a one-day summary and watch for daily usage, as well as accumulating the summaries in order to look for trends in certain metrics. If I am viewing an accounting report, I may get a statistics report for the same time period as the thread that I'm interested in was executing. In that way, I can correlate some of the accounting metrics with the statistics metrics.

Some statistics to monitor:

- **Threads, commits, signons. SQL frequency statistics.** These are great numbers to collect and monitor over time because they are a good way to gauge relative subsystem activity. That is, if management approaches you and says that CPU consumption is way up and capacity planning is

identifying DB2 as a potential reason, you can review resource consumption and use these numbers to see whether or not overall activity is up corresponding to the increase, or if not, then maybe some rogue processes have been introduced that may require tuning.

- **Checkpoints taken.** It is possible to set the system checkpoint frequency by time or the number of checkpoints written. You don't really want to checkpoint once per day, nor do you want to checkpoint every 5 seconds. Remember, checkpoints are used for crash recovery as well as normal subsystem restart. However, they also trigger system events including the launching of asynchronous buffer write requests. I like checkpointing in the area of once every 5 to 10 minutes depending upon overall system activity.
- **Reads from archived logs.** If there are processes failing and rolling back and those processes end up reading log records from archived logs, then it is quite likely there is a problem that needs to be corrected. There are either unstable processes with excessively long units of work, or the active logs are too small to accommodate the normal amount of overall logging that contains a typical unit of work. Take action!
- **Log writes suspended.** Log records are written to an output buffer. Suspended log writes may indicate a log I/O bottleneck. If this number appears high relative to log records written, then you should consider increasing the size of the log output buffer. Be generous - the log is a central point of contention.
- **DRDA requests.** This is an indication of remote request activity, and it's good to monitor this as a way of determining whether or not you have enough DBATs to accommodate the inbound requests.
- **Deadlocks and timeouts.** You don't want deadlocks and timeouts, although they may not be entirely avoidable in normal applications. These statistics should be routinely monitored and tracked for trends. Any increase should require investigation into which applications are responsible. If you look at the console log for the subsystem in which the deadlocks occurred over the time period represented by the statistics report, you should see messages that identify the applications involved in locking conflicts.
- **Lock escalations.** These are not typically desired and are an indication that one or more applications are acquiring more locks than allowed and causing larger locks to be taken. You should be able to look at the console log for the subsystem in which the escalations occurred, over the time period represented by the statistics report, and see a message for the application that experienced the escalation.
- **RID list and list prefetch.** These are numbers that reflect the quality of processing in support of SQL operations that utilize RID processing and the memory reserved for this processing including list prefetch, multi-index access, hybrid join, and other access paths. These numbers will indicate whether or not RID processing overflowed into workfiles or failed. A response to any overflows or failures should require first an investigation into which applications were involved (accounting reports) and a potential adjustment to the SQL or database objects involved (index changes, clustering changes, better matching predicates). You could also consider increasing the size of RID storage if the applications are considered well behaved.
- **Parallelism.** These values indicate how much parallelism was active on behalf of SQL running with parallelism, as well as how effective the parallelism was. If there were serious reductions in

parallelism, or failure to get parallelism at all, consider increasing buffer resources or scheduling the queries to run at times when more CPU is available. If you are getting parallelism when it is not expected, you may need to see if packages are being bound with parallelism that really shouldn't be.

- **Global locks false contention.** A large number of false contentions may indicate a need to increase the memory and/or resources available to the coupling facility.
- **Dynamic prepare statements found in cache.** If you are processing a lot of dynamic SQL then you'll want to see a high percentage of statements found in-cache versus the total number of dynamic statements. If the number is low, consider increasing the size of the statement cache or look into the types of SQL being issued. If the SQL is constructed with embedded literal values, consider changes to the SQL or the use of literal concentration.
- **Buffer pool statistics.** This is one of the most valuable measurements presented in a statistics report. Watch for extremely busy buffer pools, as well as pools that have high proportions of sequential versus random activity. The subsystem level buffer information should be used together with the detailed information provided by accounting traces and by using the DISPLAY BUFFERPOOL command. Remember that the ultimate goal of buffer tuning is to reduce the number of I/O's executed. You can use the information here to monitor buffer hitratio's, but more importantly to monitor overall activity and especially I/O activity.

I spend the vast majority of performance analysis looking at DB2 accounting reports. These reports are definitely the key to understanding application performance and are not only extremely useful for monitoring production performance, but are also important in evaluating proposed performance changes tested via benchmark tests or POCs. I also accumulate accounting information over time to look for trends in application performance for specific critical applications. In this way patterns can be detected that may indicate the need to perform REORGs, collect RUNSTATS, or look at potential application changes that may have introduced bad access paths or changes to transaction input patterns.

Some accounting fields to monitor.

- **Elapsed time and CPU time.** Ever get the call that says, "Hey my application is not responding, what's wrong with DB2?" Well, get your hands on an accounting report! The first thing you should look at is the class 1 and class 2 elapsed and CPU times. Is there a constraint? Is the application spending the majority of class 1 time doing class 2 things? If yes, it's a DB2 issue and if no, then it's not a DB2 issue. What portion of the class 2 time is elapsed time versus CPU time? If class 2 CPU is a minor portion of the elapsed time, then there are class 3 suspensions that need to be investigated. If the majority is CPU time, then you need to look at the quantity of SQL issued and perhaps even divide the number of statements into the CPU time to get an average. If the average is relatively high, you can tune some statements, but if the CPU time per statement is very low, you may have a quantity of SQL issue.
- **Secondary engine CPU time.** If you have any zIIP processors you may be interested to see how much an application is using them. Remember zIIPs are not a performance feature but a cost feature.
- **Time spent in routines.** If the application is utilizing stored procedures, user-defined functions, and triggers, then the time spent in these routines is broken out from the remaining DB2 and application time. This is important if you are using these routines for extraneous processing such that you can attach a cost to that processing. Also, if one or more routines are having a performance issue, these

numbers can indicate as much and you can then direct your attention to the package level information if it exists for the routines (always for triggers and stored procedures).

- **Lock and latch suspension time.** This indicates the number of locking and latching events that incurred a suspension, as well as the time spent in suspension. A little bit of time spent here is quite normal, but if locking represents a majority of the elapsed time then you have a concurrency issue that needs to be addressed. Look for additional information such as deadlocks and timeouts, and check the console log for messages related to logging conflicts. Sometimes excessive latch time can be an indication of limited CPU resources, so don't be fooled by this. Double check against unaccounted for time (class 2 time elapsed time – class 2 CPU time – class 2 suspension time) and if there is significant unaccounted for time associated with excessive latch wait time then there is likely a CPU shortage during the thread execution.
- **Global lock and latch suspension time.** Any significant values here are going to indicate a concurrency problem with a resource across members of a data sharing group. Look for things such as identity columns and sequence objects (increasing cache will help) or frequently updated tables shared across applications (system affinities or alternate locking sizes or strategies may be required). Large values here could also indicate undersized coupling facility resources.
- **Synchronous I/O suspension time.** Death by random I/O! If time spent here is excessive then you need to look at the clustering of the data in your tables and the input to your transactions. Buffer tuning can help, as well as additional indexes. There are many opportunities here for SQL and database tuning. The first thing to do is identify the objects and SQL statements involved.
- **Other I/O wait time.** While asynchronous I/O's are more expensive than synchronous I/O's, applications typically do not wait due to the fact that the asynchronous I/O's are independent of the application process. If there is excessive time spent here, there may be a DASD performance issue or some buffer pool separation (separating sequentially accessed objects from pools that contain sequentially accessed objects), and size increases may be required.
- **Log writes.** Logging should not be a bottleneck until you achieve extremely high rates of change in the database. However, you should add up the amount of data logged and see how well it matches the amount of data processed by the transactions. It could be an indication of excessively modifying data.
- **SQL activity and commits.** These are good counts to gauge how much work an application is doing and how that work changes over time. I typically divide elapsed and CPU time by SQL and commits to get a "per unit" measurement that can be used to direct me to specific tuning of statements, REORG opportunities, or adjustments to the size of the units of work. CPU trending over time up for a relative stable number of statements and commits is a good indication that a REORG is needed somewhere.
- **Deadlocks and timeouts.** These counters should be zero, but if they're not you can go to the console log for the period of time the thread was active to look for messages that will detail the conflicts.
- **Parallelism.** Are you expecting parallelism for one or more queries in your application? Did you actually get it? The only way to really know is here!
- **Buffer pool statistics.** The buffers are one of the most important resources in a DB2 subsystem, and buffer pool usage statistics are delivered in an accounting trace at the thread level. It is also

possible to gather these statistics at the package level if the appropriate trace class and IFCIDs are set (although there may be a higher trace cost). The information presented here is extremely valuable to see how an individual application is using (and impacting) the buffers. These statistics can be used together with the statistics trace buffer information and the information derived from using a DISPLAY BUFFERPOOL command to get a really concise picture of the buffer activity!

- **Package level statistics.** When accounting trace classes 7 and 8 are active various elapsed, CPU, SQL, and suspension time is gathered at the package level. This is extremely valuable information for static embedded applications in that you can identify which programs should get the most tuning attention (the programs that use the most resources). These package level statistics will also report on triggers, stored procedures, and some UDFs used by the application.

DB2 Performance Database

Most performance analysis tools that utilize DB2 accounting and statistics trace records are capable of formatting the information in those records into something that can be loaded into a DB2 database. Many of these tools include the database definitions for tables to populate with this performance data. I'm happy to load the performance data into a DB2 table and write a series of queries to analyze it. If no tool is available to format the data I'll actually write a REXX routine that will "scrape" accounting reports and format the data into records that can easily be loaded into a performance database. I find these databases to be extraordinarily useful for reporting trends on an application level. One way to make this information really meaningful is to establish separate authorization ids for different applications. In that way individual applications are easy to identify.

A performance database can become really large really quickly. Frequent purging can help, but moving the performance data off the mainframe can be beneficial as well. If you take a look at the PC you are using today there is a very good chance that it has a very large hard drive of which very little is in use. Why not download DB2 Express-C, install it on your computer, and load it with some performance data¹¹.

I used to offer all types of performance reports to management and most were ignored. Then the day came when I discovered the performance database, compiled trending using SQL, and then displayed the query results in Microsoft Excel with various charts. Suddenly management took notice and tuning projects were launched resulting in significant savings! Do not underestimate the power of trending and pretty pictures to make your point!

Displaying Buffer Statistics

The DISPLAY BUFFERPOOL command can also be used as a very powerful reporting tool. This is my favorite DISPLAY BUFFERPOOL command.

```
-DIS BUFFERPOOL(ACTIVE) DETAIL(INTERVAL) LSTATS(*)
```

This command will display the accumulated buffer pool statistics since the last time it was issued or since the subsystem was started if no previous command was issued. In addition, it will produce a report for every open dataset assigned to the pool that will include statistics since the last time the command was issued, or since the subsystem was started if no previous command was issued.

The total buffer pool statistics can be monitored and compared to previous reports to see how recent activity has impacted the pool. In addition the dataset list, which is listed by table space or index space name, gives detailed access information for each of the objects in the pool. In this way you can see which objects are dominating the pool as well as both synchronous and asynchronous I/O delays for each object.

Section 9:

Testing Performance Designs and Decisions

How many times can it be said that you should not be spending your time sitting in a meeting room trying to figure out a physical database design? Throw that away and spend the time instead having the database tell you how to create the database. Use the ideas presented in this book and test them in your situation. Are you tuning? Did you run an EXPLAIN before and after a change and the EXPLAIN shows an obvious improvement based upon the change to the statement or database? It all really doesn't mean much unless you are doing a proof of concept (POC) or at least running a before and after performance test. If you don't have a database or application to test with, it does not matter. You can make each in a matter of minutes.

The primary tools for testing:

- Recursive SQL to generate data.
- SQL to generate additional data.
- Recursive SQL to generate statements.
- SQL to generate additional statements.
- SPUFI for statement and data generation, as well as for running some test scenarios.
- DSNTDP2 or DSNTIAUL for running test scenarios.
- REXX for performance benchmarks and test application simulation.
- SQLPL stored procedures for performance benchmarks and test application simulation.
- Shell scripts and db2batch for remote access simulation.

Establishing a Test Database

In a situation in which you are making modifications or enhancements to existing databases and applications it can be assumed that there is a test environment to work with. However, what if there is a brand new design? Fear not! Go ahead and create a simple test database. All you really need are the key tables, primary keys, and at least one data column. Of course it depends on the strategies you are trying to test, whether it be partitioning or clustering or various alternate table designs. Go ahead and make all of the table variations so that the same sets of data can be loaded into the tables and test scenarios are easy to choose and execute. Representative catalog statistics do present somewhat of a challenge, but if you can generate an appropriate amount of test data, which is a key factor, then a normal RUNSTATS execution may be enough to get representative statistics.

Generating Data and Statements

When in a situation where there is a new database being proposed, and some POCs are required, but no existing application or data exists, then I'm typically going to generate my own data and statements. As another option I will also use SQL to generate key values that can be placed into a file and used as input to a test process. The primary tool for this generation is SQL. The primary statement to start with is the most simple of recursive SQL statements, which is a statement that generates a result set from nothing.

```
WITH GEN_ROWS(N) AS
(SELECT 1 FROM SYSIBM.SYSDUMMY1
 UNION ALL
 SELECT N+1 FROM GEN_ROWS
 WHERE N < 100)
SELECT N
FROM GEN_KEYS
ORDER BY N
```

Generating Data

In the following scenario, a test is required against a proposed employee table. This proposal is for a new application and database, so no current tables or data exists. Once we decide on the various design options we'd like to test, we're going to have to create a table, data and statements. As long as the table definition contains the appropriate structure for the test, a primary key, any potential additional indexes, and some non-indexed columns, it is good enough to test against.

```
CREATE TABLE DANL.EMPT
(EMPNO          BIGINT NOT NULL
,WORKDEPT      CHAR(3) NOT NULL
,LASTNAME      CHAR(50)
,PRIMARY KEY (EMPNO)
)
IN DANLDB.DANEMPT
CCSID EBCDIC;
CREATE UNIQUE INDEX DANL.XEMP2T
ON DANL.EMPT
(EMPNO ASC)
USING STOGROUP DANLSG
CLUSTER
BUFFERPOOL BP8
CLOSE NO;
```

In this example, we need to generate primary key values, presumed foreign key values, and some data. We can use a common table expression to make the foreign key values, an increasing value to create the primary key values, and just a simple string to make the data. The following statement will generate 100 rows of data.

```

INSERT INTO DANL.EMPT
WITH GEN_ROWS(N) AS
(SELECT 1 FROM SYSIBM.SYSDUMMY1
 UNION ALL
 SELECT N+1 FROM GEN_ROWS
 WHERE N < 100)
,GENDEPT (WORKDEPT) AS
(SELECT 'D01' FROM SYSIBM.SYSDUMMY1
 UNION ALL
 SELECT 'E01' FROM SYSIBM.SYSDUMMY1
 UNION ALL
 SELECT 'E02' FROM SYSIBM.SYSDUMMY1
 UNION ALL
 SELECT 'F01' FROM SYSIBM.SYSDUMMY1
 UNION ALL
 SELECT 'G04' FROM SYSIBM.SYSDUMMY1
 UNION ALL
 SELECT 'X01' FROM SYSIBM.SYSDUMMY1
)
SELECT A.N
      ,X.WORKDEPT
      , 'BOBRADY'
FROM GEN_ROWS A
INNER JOIN TABLE
  (SELECT WORKDEPT, RAND() AS RANDOM_VALUE
   FROM   GENDEPT
   WHERE  A.N > 0
   ORDER BY RANDOM_VALUE
   FETCH FIRST 1 ROW ONLY) AS X(WORKDEPT, RV)
ON 1=1
;

```

The first recursive common table expression generates 100 rows and also supplies our primary key. The second common table expression presents six different possible values for the WORKDEPT column. The final SELECT statement retrieves the primary key and non-indexed data from the generation table, and then it joins that to a nested table expression that contains a nonsense correlated reference to force materialization and a RAND function with an ORDER BY and FETCH FIRST in an effort to randomly supply a WORKDEPT value to each generated row. Now there are 100 rows in the test EMPT table with generated primary key values and randomly selected foreign key values.

Data can also be generated from data. This allows for the generation of more variations in values that are more difficult to generate randomly or by using a sequence value. In the following example we use the 100 rows just generated from nothing to generate 100 more rows using the same WORKDEPT values, but introducing a new data column value.

```
INSERT INTO DANL.EMPT
SELECT EMPNO + 100, WORKDEPT, 'LUKSETICH'
FROM DANL.EMPT;
```

This type of generation can be repeated over and over with ever increasing quantities of data in order to fully populate a test table in a manner that is suitable for the test at hand. Clustering can be changed and REORG executed and then the clustering changed back in an effort to test organized and disorganized data scenarios. Any variation in table design can be made and the original generated data used to populate the various table variations.

Generating Statements

Just as with generating data it is possible to generate statements in the same manner. You can generate statements from nothing using recursive SQL, or use the data in the table to generate statements that act against the table. In the following example, the data in the test table is used to generate random SELECT statements that will test random primary key index access with table access.

```
SELECT STMT
FROM (
SELECT 'SELECT LASTNAME FROM DANL.EMPT WHERE EMPNO = ' ||
      STRIP(CHAR(EMPNO)) ||
      ';'
      ,RAND() AS BOB
FROM DANL.EMPT) AS T(STMT, BOB)
ORDER BY BOB
```

The generated statements can then be used as input to SPUFI, DSNTEP2, or DSNTIAUL to run a performance test. One of the drawbacks of the above statement is that, while it will test random access to the test table, every statement will find data. What if you wanted to test misses as well as hits? One choice would be to randomly delete data once the statements are generated, and another choice would be to generate the statements independent of the data in the table. Once again, recursive SQL is used to generate something from nothing.

```
WITH GEN_ROWS(N) AS
(SELECT 1 FROM SYSIBM.SYSDUMMY1
 UNION ALL
 SELECT N+1 FROM GEN_ROWS
 WHERE N < 100)
SELECT 'SELECT LASTNAME FROM DANL.EMPT WHERE EMPNO = ' ||
      CHAR(INT(RAND() * POWER(10, INT(RAND() * 6)))) ||
      ';'
FROM GEN_ROWS
```

Many times I'll use a simple REXX program to test a SQL statement in a loop. One version of a REXX test is used to test access to a table or tables based upon a set of input keys read from a file. I'll use statements similar to the ones above to generate the key values. So, instead of generating statements they simply generate keys that are then placed in a file.

```
SELECT STRIP(CHAR(EMPNO))
      ,RAND() AS BOB
FROM   DANL.EMPT
ORDER BY BOB;
```

Simple Performance Benchmarks

There are several different tools I have used to perform performance benchmarks. My primary tool of choice is REXX. REXX is an extremely powerful and easy to learn scripting language that allows for the extremely fast development of simple applications that can include embedded dynamic SQL statements. The vast majority of tests are performed using two simple REXX routines. One tests the same statement with the same values over and over, and is used to test for CPU consumption. The other uses an input file to supply key values, either sequential or random, in an effort to test a query for response time. I also use SPUFI to test queries that process large values of data, as well as large numbers of SQL statements. DSNTEP2 and DSNTIAUL are also both effective benchmarking tools when using a SQL script containing large single statements or large volumes of individual SQL statements.

If a remote connection is possible from a PC or UNIX-based system then it is possible to also use shell scripts, windows batch files, and the db2batch system command if you have installed at least a DB2 client on the PC or other system. I usually have DB2 Express-C installed on my PC which includes mainframe connectivity (depends on when you download and who from IBM you ask) or IBM Data Studio. Both of these software products are free downloads from IBM. If you have a PC, you could use Cygwin to simulate a UNIX environment and then use shell scripting. Db2batch use, from my experience, works really great for running benchmarks against a DB2 for LUW database, but is somewhat limited when running against a DB2 for z/OS subsystem. It is, however, worth a try if you have the time.

REXX Routine Examples

The following REXX routine example is used to test for CPU usage. It runs the same statement repeatedly dependent upon the loop limit you set.

```
/*REXX*/
CALL INITIALIZE
/* SET A VARIABLE EQUAL TO THE DESIRED SQL STATMENT.
*/
/*
*/
SQLSTMT="SELECT A.LASTNAME, A.WORKDEPT
",
" FROM DANL.EMP A
",
" WHERE A.EMPNO = '000010'
"
```

```

/* PREPARE THE STATEMENT FOR EXECUTION BY DECLARING THE
CURSOR, */
/* DOING A PREPARE OF THE STATEMENT, AND OPENING THE CURSOR.
*/
/*
*/
ADDRESS DSNREXX
"EXECSQL DECLARE C1 CURSOR FOR S1"
IF SQLCODE \= "0" THEN
DO
SAY "DECLARE CURSOR FAILED WITH SQLCODE = " SQLCODE
EXIT 8
END
"EXECSQL PREPARE S1 FROM :SQLSTMT"
IF SQLCODE < "0" THEN
DO
SAY "PREPARE FAILED WITH SQLCODE = " SQLCODE
EXIT 8
END
/* OPEN/FETCH/CLOSE 50 TIMES
*/
DO 50
"EXECSQL OPEN C1 "
IF SQLCODE \= "0" THEN
DO
SAY "OPEN FAILED WITH SQLCODE = " SQLCODE
EXIT 8
END
DO UNTIL SQLCODE \= "0"
"EXECSQL FETCH C1 INTO :TESTMSG, :t2 "
IF SQLCODE = "0" THEN
DO
/* SAY TESTMSG */
END
END
/* HERE THE FETCH LOOP HAS ENDED (SQLCODE <> 0), AND WE CHECK
WHY.*/
SELECT
WHEN SQLCODE = 0 THEN DO
END
WHEN SQLCODE = 100 THEN DO
END
OTHERWISE
SAY "FETCH FAILED WITH SQLCODE = " SQLCODE

```

```

        SAY SQLERRMC
    END
    /* ALL IS WELL, SO THE CURSOR IS CLOSED.
    */
    "EXECSQL CLOSE C1"
    IF SQLCODE \= "0" THEN
        DO
            SAY "CLOSE FAILED WITH SQLCODE = " SQLCODE
        END
    /* END THE DO LOOP HERE */
    END
    /* TRACE ADDED TO CAPTURE INFO IN ONLINE MONITOR
    */
    trace ?I
    /* CLEANUP.
    */
    ADDRESS MVS "DELSTACK"
    ADDRESS DSNREXX "DISCONNECT"
    S_RC = RXSUBCOM('ADD','DSNREXX','DSNREXX')
    EXIT 0

INITIALIZE:
    /* CHECK TO SEE IF THE REXX/DB2 COMMAND ENVIRONMENT IS
    AVAILABLE */
    /* IF IT IS NOT, THEN ESTABLISH IT.
    */
    /* THEN CONNECT TO THE APPROPRIATE DATABASE (SUBSYSTEM)
    */
    'SUBCOM DSNREXX'
    IF RC THEN
        S_RC = RXSUBCOM('ADD','DSNREXX','DSNREXX')
    /* CHANGE 'DSN1' IN THE LINE BELOW TO YOUR TARGET SUBSYSTEM
    */
    ADDRESS DSNREXX "CONNECT" "DSN1"
    IF SQLCODE \= "0" THEN
        DO
            SAY "FAILURE TO CONNECT TO DATABASE"
            EXIT 8
        END
    /* INITIALIZE STACK AND FREE ANY */
    ADDRESS MVS "DELSTACK"
    ADDRESS MVS "NEWSTACK"
    RETURN;

```

The following REXX routine example is used to test for I/O dependent upon the sequence of values in the associated input file.

```

/*REXX*/
CALL INITIALIZE
/* SET A VARIABLE EQUAL TO THE DESIRED SQL STATMENT.
*/
" FROM DANL.EMP A
",
" WHERE A.EMPNO = ?
"
/* PREPARE THE STATEMENT FOR EXECUTION BY DECLARING THE
CURSOR, */
/* DOING A PREPARE OF THE STATEMENT, AND OPENING THE CURSOR.
*/
ADDRESS DSNREXX
"EXECSQL DECLARE C1 CURSOR FOR S1"
IF SQLCODE \= "0" THEN
DO
SAY "DECLARE CURSOR FAILED WITH SQLCODE = " SQLCODE
EXIT 8
END
"EXECSQL PREPARE S1 FROM :SQLSTMT"
IF SQLCODE < "0" THEN
DO
SAY "PREPARE FAILED WITH SQLCODE = " SQLCODE
EXIT 8
END
/*
DO 50
PARSE PULL empcusr.
"EXECSQL OPEN C1 USING :empcusr "
IF SQLCODE \= "0" THEN
DO
SAY "OPEN FAILED WITH SQLCODE = " SQLCODE
EXIT 8
END
/* FETCH LOOP.
*/
DO UNTIL SQLCODE \= "0"
"EXECSQL FETCH C1 INTO :TESTMSG, :t2 "
IF SQLCODE = "0" THEN
DO
END
END
END

```

```

/* HERE THE FETCH LOOP HAS ENDED (SQLCODE <> 0), AND WE CHECK
WHY.*/
SELECT
    WHEN SQLCODE = 0 THEN DO
        END
    WHEN SQLCODE = 100 THEN DO
        END
    OTHERWISE
        SAY "FETCH FAILED WITH SQLCODE = " SQLCODE
        SAY SQLERRMC
    END
/* ALL IS WELL, SO THE CURSOR IS CLOSED.
*/
"EXECSQL CLOSE C1"
IF SQLCODE \= "0" THEN
    DO
        SAY "CLOSE FAILED WITH SQLCODE = " SQLCODE
    END
/* END THE DO LOOP HERE */
END
/* TRACE PLACED AT END TO CATCH DATA IN ONLIN MONITOR */
trace ?I
/* CLEANUP.
*/
/* DISCONNECT FROM THE DATABASE, AND RELEASE ALLOCATIONS.
*/
ADDRESS MVS "DELSTACK"
ADDRESS DSNREXX "DISCONNECT"
S_RC = RXSUBCOM('ADD', 'DSNREXX', 'DSNREXX')
EXIT 0

INITIALIZE:
/* CHECK TO SEE IF THE REXX/DB2 COMMAND ENVIRONMENT IS
AVAILABLE */
/* IF IT IS NOT, THEN ESTABLISH IT.
*/
/* THEN CONNECT TO THE APPROPRIATE DATABASE (SUBSYSTEM)
*/
'SUBCOM DSNREXX'
IF RC THEN
    S_RC = RXSUBCOM('ADD', 'DSNREXX', 'DSNREXX')
/*
*/

```

```

/* CHANGE 'DSN1' IN THE LINE BELOW TO YOUR TARGET SUBSYSTEM
*/
/*
*/
ADDRESS DSNREXX "CONNECT" "DSN1"
IF SQLCODE \= "0" THEN
DO
    SAY "FAILURE TO CONNECT TO DATABASE"
    EXIT 8
END
/* INITIALIZE STACK AND FREE ANY */
/* LEFT OVER ALLOCATED FILES      */
ADDRESS MVS "DELSTACK"
ADDRESS MVS "NEWSTACK"
X = OUTTRAP(TRASH.) /* SUPPRESS MESSAGES */
ADDRESS TSO "FREE F(SYSUT1,SYSUT2)"
X = OUTTRAP(OFF) /* SUPPRESS MESSAGES */
/* ALLOCATE THE INPUT FILE */
/*                          */
ADDRESS TSO "ALLOC F(SYSUT1) DA(idinpt.text) SHR "
IF RC /=0 THEN DO
    SAY "FAILURE TO ALLOCATE INPUT DATASET"
    EXIT 8
END
/* READ THE INPUT FILE INTO THE STACK */
ADDRESS TSO "EXECIO * DISKR SYSUT1 (FINIS"
IF RC /=0 THEN DO
    SAY "FAILURE TO READ INPUT DATASET"
    EXIT 8
END
ADDRESS TSO "FREE F(SYSUT1)"
RETURN;

```

Running a Benchmark

There has to be a certain level of control and coordination when running one of these simple benchmark tests. Regardless of the tool you are using to execute the test, there must also be a certain level of preparation to guarantee repeatability, as well as control after the test in order to collect the performance measurements. Obviously, you are going to need some sort of performance monitoring tool. In the very least a DB2 accounting trace must be set with the appropriate classes, and there must be a means of reporting from the trace records generated. It's also possible to use the dynamic statement cache and EXPLAIN STMTCACHE ALL to collect performance metrics for the statements you are testing. Be sure to use an authorization id that is unique to the testing and document the exact start and end time of each test so that the report generated from whatever tool you are using can be associated to a particular test.

- Prepare the data for the test. This would involve making sure that the test tables have the correct data in them, are backed up if the test will modify data, and are either organized or disorganized based upon what you are testing for.
- Prepare any input files for the test. This may involve generation of the input files and/or sorting of the file to simulate a particular input pattern. Make sure that there is a base copy on the input data that can be used to recreate the input if you make modifications and eventually need to go back to the original.
- Make sure the appropriate traces are set and any monitoring tools are active.
- Flush the dynamic statement cache if you plan on using the metrics stored there as test results. To flush the statement cache run a RUNSTATS UPDATE NONE REPORT NO against each of the database objects involved in the test.
- Check to make sure the machine is not currently overloaded. A busy LPAR is going to heavily impact test results. Make sure your tests are run when there is some headroom so machine stress is not a significant variable. Always pay attention to in-DB2 times only when reviewing test results, but make sure to check for “not accounted for” time in the results to make sure machine stress was not a factor.
- Flush the DB2 buffers. If you are testing for I/O you especially should flush the buffers before running a test. Also, make sure that your objects and buffers are sized in such a way that you will be somewhat reflecting reality when running the tests. To flush the buffers all you need to do is to stop and start all of the database objects involved in the test prior to executing the test.
- Execute the benchmark test. Be sure to mark the exact start and stop time of the test so that you can correlate the results with the test. Set up a Microsoft Excel document to help you document the tests along with the key metrics.
- Collect the results. Put the results in a Microsoft Excel spreadsheet with full description and metrics. Include a paragraph that explains the reason for the test. When you present the reports in a design meeting, people will grill you!

When interpreting benchmark results, make sure that you pay attention primarily to the class 2 and class 3 measurements. There can be great variations in REXX performance, as well as any batch or online testing you perform due to the workload manager settings for your test jobs or TSO address space. Repeat an identical test several times to make sure that a test that should result in consistent performance metrics actually gets those consistent performance metrics. Make sure to keep very detailed and careful documentation, including any pretty bar charts or graphs, so that the people you present the results to will understand the purpose and the result. Hopefully then everyone is making an educated guess based upon some real numbers when making those critical design decisions!

I test EVERYTHING! You should as well. Cheers!

Section 10:

Top Tuning Tips

These are random thoughts that come to mind whenever I am talking to people about DB2 performance monitoring and tuning. There are probably a hundred of them and these are some of my favorites, but I'm sure, if you catch me in a hallway somewhere, there may be a few more that I insist are my favorites.

Start with the Given Design

It is completely senseless to sit in a room with a bunch of other people and debate how you think a database should be designed for performance. The people doing a logical model have done so for a specific purpose. So, you have two choices. The first is to completely abandon what has already been done and begin testing new designs based upon various hypotheses. The second is to implement the vision that has been presented and immediately begin a POC to prove where it may or may not be performance challenged. There better be time reserved in the project plan for the performance testing and any adjustments that may have to be made, but testing in a POC can happen while the application is being developed. If there is no reserved time for proving performance, then just implement the logical model and fix the problems when they happen in production just like normal people do.

Deploy Indexes Slowly

It's really easy to deploy an index as a potential solution to a performance problem. Did it really solve the problem? What is the cost of inserts and deletes to the index? Hopefully you have performed thorough performance tests prior to deploying the index in production. Once indexes are deployed it's not so easy to get rid of them, especially if they are primarily accessed by dynamic SQL. The real-time statistics table SYSIBM.SYSINDEXSPACESTATS does contain a LAST_USED column which is somewhat helpful, but not perfect, as there might be a subtle enough difference to pick one index over another when the benefit is next to nothing.

Develop a Table Access Sequence

When developing an application, establish a prescribed sequence of table access for transactions that access multiple tables. If every transaction accesses tables in the same sequence, then the worst that will ever happen is a transaction waits for a lock, but will never deadlock.

Update at the End

Retain all changes to tables in a transaction until the end of the transaction and then apply all of the changes at once before issuing a commit. In that way, locks are held for the least amount of time possible.

Always Test Performance Changes

Every database or SQL change in the name of performance should be proven by doing both an EXPLAIN and by conducting a representative performance test. You cannot rely on just the cost comparison of two EXPLAINS. A benchmark test tells the truth, and can also be used to predict actual cost savings in production.

Put Some Data in That Table

Do you have specialized tables that are used to control the flow of an application? These might be exception tables or tables that indicate one-time processing or implement bypass processing for a given key value. If these tables are empty most of the time, but accessed quite frequently, then you may want to consider putting some dummy data in the table. This is only important if there is an index on the table that is used for access (make the table VOLATILE). Index lookaside appears to be dependent upon checking key values in a leaf page. If there is no data in the index, there are no upper bound keys to test and so for some reason index lookaside is not activated. Therefore, the application can incur millions of getpages for an index that contains no keys. So, put some phony data in the table that will never be retrieved in a query and make sure the key values are higher than anything that will ever be requested. Then you can get index lookaside. For high volume transactions the savings can actually be quite large!

Get a Sponsor

You are not going to be able to effectively make application developers change code in support of performance unless you have management backing you up. Make sure to collect performance metrics and make some bar charts based upon some performance benchmarks to demonstrate the potential savings. If you can get the formula from your capacity planner to convert CPU times to dollars and put that in the chart, you may be able to get some sponsorship and be able to pressure development into making changes to support better performance.

Take Advantage of SQL Features

There are many SQL features that are designed specifically to reduce the number of calls to the data server. Multi-row operations, select from table expression, nested table expressions, common table expressions, triggers, stored procedures, and many more. Use them! Follow the directions in this book to learn how to use them to reduce the number of calls being made. This is the most important directive for performance.

Use One Authid per Application

A really easy way to separate DB2 accounting records by application is to assign one unique authorization id per application. In that way, you can easily create summary reports by application, regardless of whether it is static or dynamic SQL, or from a variety of sources. It also makes it much easier to identify which statements in the dynamic statement cache are from a particular application.

Stick a Comment in that Statement

If you place a comment in a dynamic SQL statement that contains the name of the method or program that contains the statement, then it is much easier to communicate to a developer which statement and which method it came from when you retrieve the statement text from the dynamic statement cache. Easy!

Use Query Numbers

Similar to sticking a comment in a statement you can add your own query numbers to statements. This is especially good for dynamic statements, but is generally good for all statements. It makes it easier to identify statements, but also easier to employ statement level hints if needed.

Take Advantage of Cache

Cache stuff locally in the application. Cache common code values. How often do they change?

I'd also recommend that you cache some more commonly used reference data that changes much less frequently. You can place counters in the cache that cause it to break and refresh at regular intervals. If there are four warehouses in your shipping chain why query the warehouse information table for every order?

Grab Those Resources

One of the most ridiculous things I ever heard was when a system programmer said that they don't like to run the LPARs at any more than 70% utilized at any given time. What? I feel that the machine is meant to be utilized and that idle time is a good time to conduct experiments or run some great report queries that utilize parallelism. Buffers need memory. Take your system programmer to lunch and ask real nicely about available resources. It may come in handy some day!

Understand the Trade Offs

If you ask a manager how fast he or she wants an application to respond and they say "as fast as possible", then tell them to buy an infinite number of machines. Hopefully then they will come up with a realistic SLA. Make sure you understand that flexibility and performance are at odds, and often availability is as well. Perform POCs with various configurations that favor one design option versus another and document the performance trade-offs. It's really surprising sometimes how wrong you can be when it comes to guessing performance!

Section 11:

About the Author

Dan Luksetich is an independent DB2 consultant specializing in high performance database solutions, DB2 performance evaluations, and SQL training. Dan works on some of the largest and most complex database systems in the world on the z/OS, UNIX, Linux, and Windows platforms. Dan has taken the knowledge he has gained over the years and shared it at various International DB2 User Group (IDUG) technical conferences, IBM Information on Demand technical conferences, and various local user groups in North America, Europe, Africa, and the Middle East. Dan is an IBM Champion for Information Management, chairman of the IDUG Content Committee, co-author of two DB2 for z/OS certification guides, co-author of several DB2 certification tests, and author or co-author of numerous DB2 articles published over the course of many years. He graduated with honors from the University of Illinois, and has held positions in programming and database management including COBOL, assembler, DB2 and IMS.



Connect with CA Technologies at ca.com



Agility Made Possible: The CA Technologies Advantage

CA Technologies (NASDAQ: CA) provides IT management solutions that help customers manage and secure complex IT environments to support agile business services. Organizations leverage CA Technologies software and SaaS solutions to accelerate innovation, transform infrastructure and secure data and identities, from the data center to the cloud. CA Technologies is committed to ensuring our customers achieve their desired outcomes and expected business value through the use of our technology. To learn more about our customer success programs, visit ca.com/customer-success. For more information about CA Technologies go to ca.com.