

SECTION 2: CHAPTERS 4-6

CA Performance Handbook

for DB2 for z/OS

About the Contributors from Yevich, Lawson and Associates Inc.

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

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

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

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

About CA

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

Table of Contents

About This Handbook

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

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

Who Should Read This Handbook

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

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

Page intentionally left blank

Table and Index Design for Performance

It is very important to make the correct design choices when designing physical objects such as tables, table spaces, and indexes — once a physical structure has been defined and implemented, it is generally difficult and time-consuming to make changes to the underlying structure. The best way to perform logical database modeling is to use strong guidelines developed by an expert in relational data modeling, or to use one of the many relational database modeling tools supplied by vendors. But it is important to remember that just because you can ‘press a button’ to have a tool migrate your logical model into a physical model, does not mean that the physical model is the most optimal for performance. There is nothing wrong with twisting the physical design to improve performance as long as the logical model is not compromised or destroyed.

DB2 objects need to be designed for availability, ease of maintenance, and overall performance, as well as for business requirements. There are guidelines and recommendations for achieving these design goals, but how each of these is measured will depend on the business and the nature of the data.

General Table Space Performance Recommendations

Here are some general recommendations for table space performance.

Use Segmented or Universal Table Spaces

Support for simple table spaces is going away. Although you can create them in DB2 V7 and DB2 V8, they cannot be created in DB2 9 (although you can still use previously created simple table spaces). There are several advantages to using segmented table spaces. Since the pages in a segment will only contain rows from one table, there will be no locking interference with other tables. In simple table spaces, rows are intermixed on pages, and if one table page is locked, it can inadvertently lock a row of another table just because it is on the same page. When you only have one table per table space, this is not an issue; however, there are still several benefits to having a segmented table space for one table. If a table scan is performed, the segments belonging to the table being scanned are the only ones accessed; empty pages will not be scanned. If a mass delete or a DROP table occurs, segment pages are available for immediate reuse, and it is not necessary to run a REORG utility. Mass deletes are also much faster for segmented table spaces, and they produce less logging (as long as the table has not been defined with DATA CAPTURE CHANGES). Also, the COPY utility will not have to copy empty pages left by a mass delete. When inserting records, some read operations can be avoided by using the more comprehensive space map of the segmented table space. Whatever version you are on, you should be using a segmented table space over a simple table space.

DB2 9 introduces a new type of table space called a universal table space. A universal table space is a table space that is both segmented and partitioned. Two types of universal table spaces are available: the partition-by-growth table space and the range-partitioned table space.

A universal table space offers the following benefits:

- Better space management relative to varying-length rows: A segmented space map page provides more information about free space than a regular partitioned space map page.
- Improved mass delete performance: Mass delete in a segmented table space organization tends to be faster than table spaces that are organized differently. In addition, you can immediately reuse all or most of the segments of a table.

Before DB2 9, partitioned tables required key ranges to determine the target partition for row placement. Partitioned tables provide more granular locking and parallel operations by spreading the data over more data sets. Now, in DB2 9, you have the option to partition according to data growth, which enables segmented tables to be partitioned as they grow, without the need for key ranges. As a result, segmented tables benefit from increased table space limits and SQL and utility parallelism that were formerly available only to partitioned tables, and you can avoid needing to reorganize a table space to change the limit keys.

You can implement partition-by-growth table space organization in several ways:

- You can use the new MAXPARTITIONS clause on the CREATE TABLESPACE statement to specify the maximum number of partitions that the partition-by-growth table space can accommodate. The value that you specify in the MAXPARTITIONS clause is used to protect against run-away applications that perform an insert in an infinite loop.
- You can use the MAXPARTITIONS clause on the ALTER TABLESPACE statement to alter the maximum number of partitions to which an existing partition-by-growth table space can grow. This ALTER TABLESPACE operation acts as an immediate ALTER.

A range-partitioned table space is a type of universal table space that is based on partitioning ranges and that contains a single table. The new range-partitioned table space does not replace the existing partitioned table space, and operations that are supported on a regular partitioned or segmented table space are supported on a range-partitioned table space. You can create a range-partitioned table space by specifying both SEGSIZE and Numparts keywords on the CREATE TABLESPACE statement. With a range-partitioned table space, you can also control the partition size, choose from a wide array of indexing options, and take advantage of partition-level operations and parallelism capabilities. Because the range-partitioned table space is also a segmented table space, you can run table scans at the segment level. As a result, you can immediately reuse all or most of the segments of a table after the table has been dropped or a mass delete has been performed.

Range-partitioned universal table spaces follow the same partitioning rules as for partitioned table spaces in general. That is, you can add, rebalance, and rotate partitions. The maximum number of partitions possible for both range-partitioned and partition-by-growth universal table spaces, as for partitioned table spaces, is controlled by the DSSIZE and page size.

Clustering and Partitioning

The clustering index is NOT always the primary key. It generally is not the primary key but a sequential range retrieval key and should be chosen by the most frequent range access to the table data. Range and sequential retrieval are the primary requirement, but partitioning is another requirement and can be the more critical requirement, especially as tables get extremely large in size. If you do not specify an explicit clustering index DB2 will cluster by the index that is the oldest by definition (often referred to as the first index created). If the oldest index is dropped and recreated, that index will now be a new index and clustering will now be by the next oldest index. The basic underlying rule to clustering is that if your application is going to have a certain sequential access pattern or a regular batch process you should cluster the data according to that input sequence.

Clustering and partitioning can be completely independent, and we're given a log of options for organizing our data in a single dimension (clustering and partitioning are based on the same key) dual dimensions (clustering inside each partition by a different key) or multiple dimensions (combining different tables with different partitioning unioned inside a view). You should choose a partitioning strategy based upon a concept of application controlled parallelism, separating old and new data, grouping data by time, or grouping data by some meaningful business entity (e.g. sales region, office location). Then within those partitions you can cluster the data by your most common sequential access sequence.

There is a way to dismiss clustering for inserts. See the section in this chapter on append processing.

There are several advantages to partitioning a table space. For large tables, partitioning is the only way to store large amounts of data, but partitioning also has advantages for tables that are not necessarily large. DB2 allows us to define up to 4096 partitions of up to 64 GB each (however, total table size is limited depending on the DSSIZE specified). Non-partitioned table spaces are limited to 64 GB of data. You can take advantage of the ability to execute utilities on separate partitions in parallel. This also gives you the ability to access data in certain partitions while utilities are executing on others. In a data-sharing environment, you can spread partitions among several members to split workloads. You can also spread your data over multiple volumes and need not use the same storage group for each data set belonging to the table space. This also allows you to place frequently accessed partitions on faster devices.

Free Space

The FREEPAGE and PCTFREE clauses are used to help improve the performance of updates and inserts by allowing free space to exist on table spaces. Performance improvements include improved access to the data through better clustering of data, faster inserts, fewer row overflows, and a reduction in the number of REORGs required. Some tradeoffs include an increase in the number of pages, fewer rows per I/O and less efficient use of buffer pools, and more pages to scan. As a result, it is important to achieve a good balance for each individual table space and index space when deciding on free space, and that balance will depend on the processing requirements of each table space or index space. When inserts and updates are performed, DB2 will use the free space defined, and by doing this it can keep records in clustering sequence as much as possible. When the free space is used up, the records must be located elsewhere, and this is when performance can begin to suffer. Read-only tables do not

require any free space, and tables with a pure insert-at-end strategy (append processing) generally don't require free space. Exceptions to this would be tables with VARCHAR columns and tables using compression that are subject to updates. When DB2 attempts to maintain cluster during inserting and updating it will search nearby for free space and/or free pages for the row. If this space is not found DB2 will exhaustively search the table space for a free place to put the row before extending a segment or a data set. You can notice this activity by gradually increasing insert CPU times in you application (by examining the accounting records) as well as increasing getpage counts and relocated row counts. When this happens it's time for a REORG, and a perhaps a reevaluation of your free space quantities.

Allocations

The PRIQTY and SECQTY clauses of the CREATE TABLESPACE and ALTER TABLESPACE SQL statements specify the space that is to be allocated for the table space if the table space is managed by DB2. These settings influence the allocation by the operating system of the underlying VSAM data sets in which table space and index space data is stored. The PRIQTY specifies the minimum primary space allocation for a DB2-managed data set of the table space or partition. The primary space allocation is in kilobytes, and the maximum that can be specified is 64 GB. DB2 will request a data set allocation corresponding to the primary space allocation, and the operating system will attempt to allocate the initial extent for the data set in one contiguous piece. The SECQTY specifies the minimum secondary space allocation for a DB2-managed data set of the table space or partition. DB2 will request secondary extents in a size according to the secondary allocation. However, the actual primary and secondary data set sizes depend upon a variety of settings and installation parameters.

You can specify the primary and secondary space allocations for table spaces and indexes or allow DB2 to choose them. Having DB2 choose the values, especially for the secondary space quantity, increases the possibility of reaching the maximum data set size before running out of extents. In addition, the MGEXTSZ subsystem parameter will influence the SECQTY allocations, and when set to YES (NO is the default) changes the space calculation formulas to help utilize all of the potential space allowed in the table space before running out of extents.

You can alter the primary and secondary space allocations for a table space. The secondary space allocation will take immediate effect. However, since the primary allocation happens when the data set is created, then that allocation will not take affect until a data set is added (depends upon the type of table space) or until the data set is recreated via utility execution (such as a REORG or LOAD REPLACE).

Column Ordering

There are two reasons you want to order your columns in specific ways; to reduce CPU consumption when reading and writing columns with variable length data, and to minimize the amount of logging performed when updating rows. Which version of DB2 you are using will impact how you, or how DB2, organizes your columns.

For reduced CPU when using variable length columns you'll want to put your variable length columns after all of your fixed length columns (DB2 V7 and DB2 V8). If you mix the variable length columns and your fixed length columns together then DB2 will have to search for any fixed or variable length column after the first variable length column, and this will increase CPU consumption. So, in DB2 V7 or DB2 V8 you want to put the variable length columns after the fixed length columns when defining your table. This is especially true for any read-only applications. For applications in which the rows are updated, you may want to organize your data differently (read on). Things change with DB2 9 as it employs something called reordered row format. Once you move to new function mode in DB2 9 any new tablespace you create will automatically have its variable length columns placed after the fixed length columns physically in the table space, regardless of the column ordering in the DDL. Within each grouping (fixed and variable) your DDL column order is respected. In addition to new table spaces any table spaces that are REORGed or LOAD REPLACEd will get the reordered row format.

For reduced logging you'll want to order the rows in your DDL a little differently. For high update tables you'll want the columns that never changed placed first in the row, followed by the columns that change less frequently, then followed by the columns that changed all the time (e.g. an update timestamp). So, you'll want your variable length columns that never change in front of the fixed length columns that do change (DB2 V7 and DB2 V8) in order to reduce logging. This is because DB2 will record the first byte changed to last byte changed for fixed length rows, and first byte changed to end of the row for variable length rows if the length changes (unless the table has been defined with DATA CAPTURE CHANGES which will cause the entire before and after image to be logged for updates). This all changes once you've moved to DB2 9, and the table space is using the reordered row format. In this case you have no control over the placement of never changing variable length rows in front of always changing fixed length rows. This can possibly mean increased logging for your heavy updaters. To reduce the logging in these situations you can still order the columns such that the most frequently updated columns are last, and DB2 will respect the order of the columns within the grouping. You can also contact IBM about turning off the automatic reordered row format if this is a concern for you.

Utilizing Table Space Compression

Using the COMPRESS clause of the CREATE TABLESPACE and ALTER TABLESPACE SQL statements allows for the compression of data in a table space or in a partition of a partitioned table space. In many cases, using the COMPRESS clause can significantly reduce the amount of DASD space needed to store data, but the compression ratio achieved depends on the characteristics of the data.

Compression allows us to get more rows on a page and therefore see many of the following performance benefits, depending on the SQL workload and the amount of compression:

- Higher buffer pool hit ratios
- Fewer I/Os
- Fewer getpage operations
- Reduced CPU time for image copies

There are also some considerations for processing cost when using compression, but that cost is relatively low.

- The processor cost to decode a row using the COMPRESS clause is significantly less than the cost to encode that same row.
- The data access path DB2 uses affects the processor cost for data compression. In general, the relative overhead of compression is higher for table space scans and less costly for index access.

Some data will not compress well so you should query the PAGESAVE column in SYSIBM.SYSTABLEPART to be sure you are getting a savings (at least 50% is average). Data that does not compress well includes binary data, encrypted data, and repeating strings. Also you should never compress small tables/rows if you are worried about concurrency issues as this will put more rows on a page.

Keep in mind when you compress the row is treated as varying length with length change when it comes to updates. This means there is a potential for row relocation causing high numbers in NEARINDREF and FARINDREF. This means you are now doing more I/O to get to your data because it has been relocated and you will have to REORG to get it back to its original position.

Utilizing Constraints

Referential integrity (RI) allows you to define required relationships between and within tables. The database manager maintains these relationships, which are expressed as referential constraints, and requires that all values of a given attribute or table column also exist in some other table column.

In general DB2 enforced referential integrity is much more efficient than coding the equivalent logic in your application program. In addition, have the relationships enforced in a central location in the database is much more powerful than making it dependent upon application logic. Of course, you are going to need indexes to support the relationships enforced by DB2.

Remember that referential integrity checking has cost associated with it and can become expensive if used for something like continuous code checking. RI is meant for parent/child relationship, not code checking. Better options for this include check constraints, or even better to put codes in memory and check them there.

Table check constraints will enforce data integrity at the table level. Once a table-check constraint has been defined for a table, every UPDATE and INSERT statement will involve checking the restriction or constraint. If the constraint is violated, the data record will not be inserted or updated, and a SQL error will be returned.

A table check constraint can be defined at table creation time or later, using the ALTER TABLE statement. The table-check constraints can help implement specific rules for the data values contained in the table by specifying the values allowed in one or more columns in every row of a table. This can save time for the application developer, since the validation of each data value can be performed by the database and not by each of the applications accessing the database. However, check constraints should, in general, not be used for data edits in support of data entry. It's best to cache code values locally within the application and performs the edits local to the application. This will avoid numerous trips to the database to enforce the constraints.

Indexing

Depending upon you application and the type of access, indexing can be a huge performance advantage or a performance bust. Is your application a heavy reader, or perhaps even a read-only application? Then lots of indexes can be a real performance benefit. What if your application is constantly inserting, updating, and deleting from your table. Then in that case maybe lots of indexes can be a detriment. When does it matter, well of course it depends. Just remember this simple rule; if you are adding a secondary index to a table then for inserts and deletes, and perhaps even updates, you are adding another random read to these statements. Can you application afford that in support of queries that may use the index? That's for you to decide.

The Holy Grail of Efficient Database Design

When designing your database you should set out with a single major goal in mind for your tables that will contain a significant amount of data, and lots of DML activity. That goal is one index per table. If you can achieve this goal then you've captured the holy grail of efficient database design. These indexes would need to support:

- Insert strategy
- Primary key
- Foreign key (if a child table)
- SQL access path
- Clustering

You can head on this path by respecting some design objectives:

- Avoid surrogate keys. Use meaningful business keys instead
- Let the child table inherit the parent key as part of the child's primary key
- Cluster all tables in a common sequence
- Determine the common access paths, respect them, and try not to change them during design
- Never entertain a "just in case" type of design mentality

General Index Design Recommendations

Once you've determined your indexes you need to design them properly for performance. Here are some general index design guidelines.

Index Compression

As of DB2 9 an index can be defined with the COMPRES YES option (COMPRESS NO is default). Index compression can be used where there is a desire to reduce the amount of disk space an index consumes. Index compression is recommended for applications that do sequential insert operations with few or no delete operations. Random inserts and deletes can adversely effect compression. An Index compression is also recommended for applications where the indexes are created primarily for scan operations.

A bufferpool that is used to create the index must be 8K, 16K, or 32K in size. The physical page size for the index on disk will be 4K. The reason that the bufferpool size is larger than the page size is that index compression only saves space on disk. The data in the index page is expanded when read into the pool. So, index compression can possibly save you read time for sequential operations, and perhaps random (but far less likely).

Index compression can have a significant impact on the REORGs and index rebuilds resulting in significant savings in this area. Keep in mind, however, that if you use the copy utility to back up an index that image copy is actually uncompressed.

Index Free Space

Setting the PCTFREE and FREEPAGE for your indexes depends upon how much insert and delete activity is going to occur against those indexes. For indexes that have little or no inserts and deletes (updates that change key columns are actually inserts and deletes) then you can probably use a small PCTFREE with no free pages. For indexes with heavy changes you should consider larger amounts of free space. Keep in mind that adding free space may increase the number of index levels, and subsequently increase the amount of I/O for random reads. If you don't have enough free space you could get an increased frequency of index page splits. When DB2 splits a page it's going to look for a free page in which to place one of the split pages. If it does not find a page nearby it will exhaustively search the index for a free page. This could lead to CPU and locking problems for very large indexes. The best thing to do is to set a predictive PCTFREE that anticipates growth over a period of time such that you will never split a page. Then you should monitor the frequency of page splits to determine when to REORG the index, or establish a regular REORG policy for that index.

Secondary Indexes

There are two types of secondary indexes, non-partitioning secondary indexes and data partitioned secondary indexes.

NON-PARTITIONING SECONDARY INDEXES NPSIs are indexes that are used on partitioned tables. They are not the same as the clustered partitioning key, which is used to order and partition the data, but rather they are for access to the data. NPSIs can be unique or non-unique. While you can have only one clustered partitioning index, you can have several NPSIs on a table if necessary. NPSIs can be broken apart into multiple pieces (data sets) by using the PIECESIZE clause on the CREATE INDEX statement. Pieces can vary in size from 254 KB to 64 GB — the best size will depend on how much data you have and how many pieces you want to manage. If you have several pieces, you can achieve more parallelism on processes, such as heavy INSERT batch jobs, by alleviating the bottlenecks caused by contention on a single data set. As of DB2 V8 and beyond the NPSI can be the clustering index.

NPSIs are great for fast read access as there is a single index b-tree structure. They can, however, grow extremely large and become a maintenance and availability issue.

DATA PARTITIONED SECONDARY INDEXES The DPSI index type provides us with many advantages for secondary indexes on a partitioned table space over the traditional NPSIs (Non-Partitioning Secondary Indexes) in terms of availability and performance.

The partitioning scheme of the DPSI will be the same as the table space partitions and the index keys in 'x' index partition will match those in 'x' partition of the table space. Some of the benefits that this provides include:

- Clustering by a secondary index
- Ability to easily rotate partitions
- Efficient utility processing on secondary indexes (no BUILD-2 phase)
- Allow for reducing overhead in data sharing (affinity routing)

DRAWBACKS OF DPSIS While there will be gains in furthering partition independence, some queries may not perform as well. If the query has predicates that reference columns in a single partition are therefore are restricted to a single partition of the DPSI it will benefit from this new organization. The queries will have to be designed to allow for partition pruning through the predicates in order to accomplish this. This means that the at least leading column of the partitioning key has to be supplied in the query in order for DB2 to prune (eliminate) partitions from the query access path. However if the predicate references only columns in the DPSI it may not perform very well because it may need to probe several partitions of the index. Other limitations to using DPSIs include the fact that they cannot be unique (some exceptions in DB2 9) and they may not be the best candidates for ORDER BYs.

Rebuild or Recover?

As of DB2 V8 you can define an index as COPY YES. This means, as with a table space, you can use the COPY and RECOVER utilities to backup and recover these indexes. This may be especially useful for very large indexes. Be aware, however, that large NPSI's cannot be copied in pieces, and can get very large in size. You'll need to have large data sets to hold the backup. This could mean large quantities of tapes, or perhaps even hitting the 59 volume limit for a data set on DASD. REBUILD will require large quantities of temporary DASD to support sorts, as well as more CPU than a RECOVER. You should carefully consider whether your strategy for an index should be backup and recover, or rebuild.

Special Types of Tables Used for Performance

Here are some table designs that are part of the DB2 product offering and are intended to help you boost the performance of the applications.

Materialized Query Tables

Decision support queries are often difficult and expensive. They typically operate over a large amount of data which may have to scan or process terabytes of data and possibly perform multiple joins and complex aggregations. With these types of queries traditional optimization and performance is not always optimal.

As of DB2 V8, one solution can be with the use of MQTs — Materialized Query Tables. This allows you to precompute whole or parts of each query and then use computed results to answer future queries. MQTs provide the means to save the results of prior queries and then reuse the common query results in subsequent queries. This helps avoid redundant scanning, aggregating and joins. MQTs are useful for data warehouse type applications.

MQTs do not completely eliminate optimization problems but rather move optimizations issues to other areas. Some challenges include finding the best MQT for expected workload, maintaining the MQTs when underlying tables are updated, ability to recognize usefulness of MQT for a query, and the ability to determine when DB2 will actually use the MQT for a query. Most of these types of problems are addressed by OLAP tools, but MQTs are the first step.

The main advantage of the MQT is that DB2 is able to recognize a summary query against the source table(s) for the MQT, and rewrite the query to use the MQT instead. It is, however, your responsibility to move data into the MQT, with via a REFRESH TABLE command, or by manually moving the data yourself.

Volatile Tables

As of DB2 V8, volatile tables are a way to prefer index access over table space scans or non-matching index scans for tables that have statistics that make them appear to be small. They are good for tables that shrink and grow allowing matching index scans on tables that have grown larger without new RUNSTATS.

They also improve support for cluster tables. Cluster tables are those tables that have groups or clusters of data that logically belong together. Within each group rows need to be accessed in same sequence to avoid lock contention during concurrent access. The sequence of access is determined by primary key and if DB2 changes the access path lock contention can occur. To best support cluster tables (volatile tables) DB2 will use index only access when possible. This will minimize application contention on cluster tables by preserving the access sequence by primary key. We need to be sure indexes are available for single table access and joins.

The keyword VOLATILE can be specified on the CREATE TABLE or the ALTER TABLE statements. If specified you are basically forcing an access path of index accessing and no list prefetch.

Clone Tables

In DB2 9 you can create a clone table on an existing base table at the current server by using the ALTER TABLE statement. Although ALTER TABLE syntax is used to create a clone table, the authorization granted as part of the clone creation process is the same as you would get during regular CREATE TABLE processing. The schema (creator) for the clone table will be the same as for the base table. You can create a clone table only if the base table is in a universal table space.

To create a clone table, issue an ALTER TABLE statement with the ADD CLONE option.

```
ALTER TABLE base-table-name ADD CLONE clone-table-name
```

The creation or drop of a clone table does not impact applications accessing base table data. No base object quiesce is necessary and this process does not invalidate plans, packages, or the dynamic statement cache.

You can exchange the base and clone data by using the EXCHANGE statement. To exchange table and index data between the base table and clone table issue an EXCHANGE statement with the DATA BETWEEN TABLE table-name1 AND table-name2 syntax. This is in essence a method of performing an online load replace!

After a data exchange, the base and clone table names remain the same as they were prior to the data exchange. No data movement actually takes place. The instance numbers in the underlying VSAM data sets for the objects (tables and indexes) do change, and this has the effect of changing the data that appears in the base and clone tables and their indexes. For example, a base table exists with the data set name *I0001.*. The table is cloned and the clone's data set is initially named *.I0002.*. After an exchange, the base objects are named *.I0002.* and the clones are named *I0001.*. Each time that an exchange happens, the instance numbers that represent the base and the clone objects change, which immediately changes the data contained in the base and clone tables and indexes. You should also be aware of the fact that when the clone is dropped and an uneven number of EXCHANGE statements have been executed, the base table will have an *I0002.* data set name. This could be confusing.

Some Special Table Designs

In situations where you are storing large amounts of data or need to maximize transaction performance there are many creative things you can do. Here are a few alternatives to traditional table design.

UNION in View for Large Table Design

The amount of data being stored in DB2 is increasing in a dramatic fashion. Availability is also an increasingly important feature of our databases. So, as we build these giant tables in our system, we have to make sure that they are built in a way that makes them available 24 hours a day, 7 days per week. These high demands are pushing database administrators to the edge. They have to build large tables that are easy to access, hold significant quantities of data, are easy to manage, and available all of the time.

Traditionally, our larger database tables have been placed into partitioned tablespaces. Partitioning helps with database management because it's easier to manage several small objects versus one very large object. There are still some limits to partitioning. For example, each partition is limited to a maximum size of 64GB, a partitioning index is required (DB2 V7 only), and if efficient alternate access paths to the data are desired then non-partitioning indexes (NPSIs) are required. These NPSIs are not partitioned, and exist as single large database indexes. Thus NPSIs can present themselves as an obstacle to availability (i.e. a utility operation against a single partition may potentially make the entire NPSI unavailable), and as impairment to database management as it is more difficult to manage such large database objects.

A UNION in a view can be utilized as an alternative to table partitioning in support of very large database tables. In this type of design, several database tables can be created to hold different subsets of the data that would have otherwise be held in a single table. Key values, similar to what may be used in partitioning, can be used to determine which data goes into which of the various tables. Take, for example, the following view definition:

```
CREATE VIEW V_ACCOUNT_HISTORY
  (ACCOUNT_ID, PAYMENT_DATE, PAYMENT_AMOUNT,
   PAYMENT_TYPE, INVOICE_NUMBER)
AS
SELECT ACCOUNT_ID, PAYMENT_DATE, PAYMENT_AMOUNT,
        PAYMENT_TYPE, INVOICE_NUMBER
FROM   ACCOUNT_HISTORY1
WHERE  ACCOUNT_ID BETWEEN 1 AND 100000000
UNION ALL
SELECT ACCOUNT_ID, PAYMENT_DATE, PAYMENT_AMOUNT,
        PAYMENT_TYPE, INVOICE_NUMBER
FROM   ACCOUNT_HISTORY2
WHERE  ACCOUNT_ID BETWEEN 100000001 AND 200000000
UNION ALL
SELECT ACCOUNT_ID, PAYMENT_DATE, PAYMENT_AMOUNT,
        PAYMENT_TYPE, INVOICE_NUMBER
FROM   ACCOUNT_HISTORY3
WHERE  ACCOUNT_ID BETWEEN 200000001 AND 300000000
UNION ALL
SELECT ACCOUNT_ID, PAYMENT_DATE, PAYMENT_AMOUNT,
        PAYMENT_TYPE, INVOICE_NUMBER
FROM   ACCOUNT_HISTORY4
WHERE  ACCOUNT_ID BETWEEN 300000001 AND 999999999;
```

By separating the data into different table, and creating the view over the tables we can create a logical account history table with these distinct advantages over a single physical table:

- We can add or remove tables with very small outages, usually just the time it takes to drop and recreate the view.
- We can partition each of the underlying tables, creating still smaller physical database objects.
- NPSIs on each of the underlying tables could be much smaller and easier to manage then they would be under a single table design.
- Utility operations could execute against an individual underlying table, or just a partition of that underlying table. This greatly shrinks utility times against these individual pieces, and improves concurrency. This truly gives us full partition independence.
- The view can be referenced in any SELECT statement in exactly the same way as a physical table would be.
- Each underlying table could be as large as 16TB, logically setting the size limit of the table represented by the view at 64TB.
- Each underlying table could be clustered differently, or could be a segmented or partitioned tablespace.
- DB2 will distribute predicates against the view to every query block within the view, and then compare the predicates. Any impossible predicates will result in the query block being pruned (not executed). This is an excellent performance feature.

There are some limitations to UNION in a view that should be considered:

- The view is read-only. This means you'd have to utilize special program logic and possibly even dynamic SQL to perform inserts, updates, and deletes against the base tables. If you are using DB2 9, however, you can possible utilize INSTEAD OF triggers to provide this functionality.
- Predicate transitive closure happens after the join distribution. So, if you are joining from a table to a UNION in a view and predicate transitive closure is possible from the table to the view then you have to code the redundant predicate yourself.
- DB2 can apply query block pruning for literal values and host variables, but not joined columns. For that reason if you expect query block pruning on a joined column then you should code a programmatic join (this is the only case). Also, in some cases the pruning does not always work for host variables so you need to test.
- When using UNION in a view you'll want to keep the number of tables in a query to a reasonable number. This is especially true for joining because DB2 will distribute the join to each query block. This multiplies the number of tables in the query, and this can increase both bind time and execution time. Also, you could in some cases exceed the 225 table limit in which case DB2 will materialize the view.
- In general, you'll want to keep the number of tables UNIONed in the view to 15 or under.

Append Processing for High Volume Inserts

In situations in which you have very high insert rates you may want to utilize append processing. When append processing is turned on, DB2 will use an alternate insert algorithm whose intention it is to simply add data to the end of a partition or a table space. Append processing can be turned off for DB2 V7 or DB2 V8 by setting the table space options PCTFREE 0 FREEPAGE 0 MEMBER cluster. Make sure to have APARS PQ86037 and PQ87381 applied. In DB2 9 append processing is turned on via the APPEND YES option of the CREATE TABLE statement. When append processing is turned on, DB2 will not respect cluster during inserts and simply put all new data at the end of the table space.

If you have a single index for read access, then having append processing may mean more random reads. This may require more frequent REORGs to keep the data organized for read access. Also, if you are partitioning, and the partitioning key is not the read index then you will still have random reads during insert to your non-partitioned index. You'll need to make sure you have adequate free space to avoid index page splits.

Append processing can also be used to store historical or seldom read audit information. In these cases you would want to partition based upon an every ascending value (e.g. a date) and have all new data go to the end of the last partition. In this situation all table space maintenance, such as copies and REORGs, will be against the last partition. All other data will be static and will not require maintenance. You will possibly need a secondary index, or each read query will have to be for a range of values within the ascending key domain.

Building an Index on the Fly

In situations in which you are storing data via a key designed for a high speed insert strategy with minimal table maintenance you'd also like to try to avoid secondary indexes. Scanning billions of rows typically is not an option.

The solution may be to build a look-up table that acts as a sparse index. This look up table will contain nothing more than your ascending key values. One example would be dates, say one date per month for every month possible in our database. If the historical data is organized and partitioned by the date, and we have only one date per month (to further sub-categorize the data), then we can use our new sparse index to access the data we need. Using user-supplied dates as starting and ending points the look up table can be used to fill the gap with the dates in between. This gives us the initial path to the history data. Read access is performed by constructing a key during SELECT processing. So, in this example we'll access an account history table (ACCT_HIST) that has a key on HIST_EFF_DTE, ACCT_ID, and our date lookup table called ACCT_HIST_DATES, which contains one column and one row for each legitimate date value corresponding to the HIST_EFF_DTE column of the ACCT_HIST table.

CURRENT DATA ACCESS Current data access is easy; we can retrieve the account history data directly from the account history table.

```
SELECT {columns}
FROM ACCT_HIST
WHERE ACCT_ID = :ACCT-ID
AND HIST_EFF_DTE = :HIST-EFF-DTE;
```

RANGE OF DATA ACCESS Accessing a range of data is a little more complicated than simply getting the most recent account history data. Here we need to use our sparse index history date table to build the key on the fly. We apply the range of dates to the date range table, and then join that to the history table.

```
SELECT {columns}
FROM ACCT_HIST HIST
INNER JOIN
    ACCT_HIST_DATES DTE
ON HIST.HIST_EFF_DTE = DTE.EFF_DTE
WHERE HIST.ACCT_ID = :ACCT-ID
AND HIST.HIST_EFF_DTE BETWEEN :BEGIN-DTE AND :END-DTE;
```

FULL DATA ACCESS To access all of the data for an account we simply need a version of the previous query without the date range predicate.

```
SELECT {columns}
FROM ACCT_HIST HIST
INNER JOIN
    ACCT_HIST_DATES DTE
ON HIST.HIST_EFF_DTE = DTE.EFF_DTE
WHERE HIST.ACCT_ID = :ACCT-ID;
```

Denormalization “Light”

In many situations, especially those in which there is a conversion from a legacy flat file based system to a relational database, there is a performance concern (or more importantly a performance problem in that an SLA is not being met) for reading the multiple DB2 tables. These are situations in which the application is expecting to read all of the data that was once represented by a single record, but is now in many DB2 tables. In these situations many people will begin denormalizing the tables. This is an act of desperation! Remember the reason your moving your data into DB2 in the first place, and that is for all the efficiency, portability, flexibility, and faster time to delivery for your new applications. By denormalizing you are throwing these advantages away, and you may as well have stayed with your old flat file based design.

In some situations, however, the performance of reading multiple tables compared to the equivalent single record read just isn't good enough. Well, instead of denormalization you could possibly employ a “denormalization light” instead. This type of denormalization can be applied to parent and child tables, when the child table data is in an optional relationship to the parent. Instead of denormalizing the optional child table data into the parent table simply instead add a column to the parent table that basically indicates whether or not the child table has any data for that parent key. This will require some additional application responsibility in maintaining that indicator column. However, DB2 can utilize a during join predicate to avoid probing the child table when there is no data for the parent key.

Take, for example, an account table and an account history table. The account may or may not have account history, and so the following query would join the two tables together to list the basic account information (balance) along with the history information if present:

```

SELECT A.CURR_BAL, B.DTE, B.AMOUNT
FROM   ACCOUNT A
LEFT OUTER JOIN
       ACCT_HIST B
ON     A.ACCT_ID = B.ACCT_ID
ORDER BY B.DTE DESC

```

In the example above the query will always probe the account history table in support of the join, whether or not the account history table has data. We can employ our light denormalization by adding an indicator column to the account table. Then we can use a during join predicate. DB2 will only perform the join operation with the join condition is true. In this particular case the access to the account history table is completely avoided when the indicator column a value not equal to “Y”:

```

SELECT A.CURR_BAL, B.DTE, B.AMOUNT
FROM   ACCOUNT A
LEFT OUTER JOIN
       ACCT_HIST B
ON     A.ACCT_ID = B.ACCT_ID
AND    A.HIST_IND = 'Y'
ORDER BY B.DTE DESC

```

DB2 is going to test that indicator column first before performing the join operation, and supply nulls for the account history table when data is not present as indicated.

You can image now the benefit of this type of design when you are doing a legacy migration from a single record system to 40 or so relational tables with lots of optional relationships. This form of denormalizing can really improve performance in support of legacy system access, while maintaining the relation design for efficient future applications.

EXPLAIN and Predictive Analysis

It is important to know something about how your application will perform prior to the application actually executing in a production environment. There are several ways in which we can predict the performance of our applications prior to implementation, and several tools can be used. The important thing you have to ask when you begin building your application is “Is the performance important?” If not, then proceed with development at a rapid pace, and then fix the performance once the application has been implemented. What if you can’t wait until implementation to determine performance? Well, then you’re going to have to predict the performance. This chapter will suggest ways to do that.

EXPLAIN Facility

The DB2 EXPLAIN facility is used to expose query access path information. This enables application developers and DBAs to see what access path DB2 is going to take for a query, and if any attempts at query tuning are needed. DB2 can gather basic access path information in a special table called the PLAN_TABLE (DB2 V7, DB2 V8, DB2 9), as well as detailed information about predicate stages, filter factor, predicate matching, and dynamic statements that are cached (DB2 V8, DB2 9).

EXPLAIN can be invoked in one of the following ways:

- Executing the EXPLAIN SQL statement for a single statement
- Specifying the BIND option EXPLAIN(YES) for a plan or package bind
- Executing the EXPLAIN via the Optimization Service Center or via Visual EXPLAIN

When EXPLAIN is executed it can populate many EXPLAIN tables. The target set of EXPLAIN tables depends upon the authorization id associated with the process. So, the creator (schema) of the EXPLAIN tables is determined by the CURRENT SQLID of the person running the EXPLAIN statement, or the OWNER of the plan or package at bind time. The EXPLAIN tables are optional, and DB2 will only populate the tables that it finds under the SQLID or OWNER of the process invoking the EXPLAIN. There are many EXPLAIN tables, which are documented in the DB2 Performance Monitoring and Tuning Guide (DB2 9). Some of these tables are not available in DB2 V7. The following tables can be defined manually, and the DDL can be found in the DB2 sample library member DSNTESC:

- **PLAN_TABLE** The PLAN_TABLE contains basic access path information for each query block of your statement. This includes, among other things, information about index usage, and the number of matching index columns, which join method is utilized, which access method is utilized, and whether or not a sort will be performed. The PLAN_TABLE forms the basis for access path determination.

- **DSN_STATEMNT_TABLE** The statement table contains estimated cost information for the cost of a statement. If the statement table is present when you EXPLAIN a query, then it will be populated with the cost information that corresponds to the access path information for the query that is stored in the PLAN_TABLE. For a given statement, this table will contain the estimated processor cost in milliseconds, as well as the estimated processor cost in service units. It places the cost values into two categories:
 - **Category A** DB2 had enough information to make a cost estimation without using any defaults.
 - **Category B** DB2 had to use default values to make some cost calculations.

The statement table can be used to compare estimated costs when you are attempting to modify statements for performance. Keep in mind, however, that this is a cost estimate, and is not truly reflective of how your statement will be used in an application (given input values, transaction patterns, etc). You should always test your statements for performance in addition to using the statement table and EXPLAIN.

- **DSN_FUNCTION_TABLE** The function table contains information about user-defined functions that are a part of the SQL statement. Information from this table can be compared to the cost information (if populated) in the DB2 System Catalog table, SYSIBM.SYSROUTINES, for the user-defined functions.
- **DSN_STATEMENT_CACHE_TABLE** This table is not populated by a normal invocation of EXPLAIN, but instead by the EXPLAIN STMTCACHE ALL statement. Issuing the statement will result in DB2 reading the contents of the dynamic statement cache, and putting runtime execution information into this table. This includes information about the frequency of execution of these statements, the statement text, the number of rows processed by the statement, lock and latch requests, I/O operations, number of index scans, number of sorts, and much more! This is extremely valuable information about the dynamic queries executing in a subsystem. This table is available only for DB2 V8 and DB2 9.

EXPLAIN Tables Utilized by Optimization Tools

There are many more EXPLAIN tables than those listed here. These additional EXPLAIN tables are typically populated by the optimization tools that use them. However, at least some of them you can make and use yourself without having to use the various optimization tools. These additional tables are documented in the DB2 Performance Monitoring and Tuning Guide (DB2 9), however some of them are utilized in Version 8.

What EXPLAIN Does and Does Not Tell You

The PLAN_TABLE is the key table for determining the access path for a query. Here is some of the critical information it provides:

- **METHOD** This column indicates the join method, or whether or not an additional sort step is required.
- **ACCESSTYPE** This column indicates whether or not the access is via a table space scan, or via index access. If it is by index access the specific type of index access is indicated.
- **MATCHCOLS** If an index is used for access this column indicates the number of columns matched against the index.

- **INDEXONLY** A value of “Y” in this column indicates that the access required could be fully served by accessing the index only, and avoiding any table access.
- **SORTN####, SORTC####** These columns indicate any sorts that may happen in support of a UNION, grouping, or a join operation, among others.
- **PREFETCH** This column indicates whether or not prefetch may play a role in the access.

By manually running EXPLAIN, and examining the PLAN_TABLE, you can get good information about the access path, indexes that are used, join operations, and any sorting that may be happening as part of your query. If you have additional EXPLAIN tables created (those created by Visual Explain among other tools) then those tables are populated automatically either by using those tools, or by manually running EXPLAIN. You can also query those tables manually, especially if you don't have the remote accessed required from a PC to use those tools. The DB2 Performance Monitoring and Tuning Guide (DB2 9) documents all of these tables. These tables provide detailed information about such things as predicate stages, filter factor, partitions eliminated, parallel operations, detailed cost information, and more.

There is some information, however, that EXPLAIN does not tell you about your queries. You have to be aware of this to effectively do performance tuning and predictive analysis. Here are some of the things you cannot get from EXPLAIN:

- **INSERT indexes** EXPLAIN does not tell you the index that DB2 will use for an INSERT statement. Therefore, it's important that you understand your clustering indexes, and whether or not DB2 will be using APPEND processing for your inserts. This understanding is important for INSERT performance, and the proper organization of your data. See Chapter 4 for more information in this regard.
- **Access path information for enforced referential constraints** If you have INSERTS, UPDATES, and DELETES in the program you have EXPLAINed, then any database enforced RI relationships and associated access paths are not exposed in the EXPLAIN tables. Therefore, it is your responsibility to make sure that proper indexes in support of the RI constraints are established and in use.
- **Predicate evaluation sequence** The EXPLAIN tables do not show you the order in which predicates of the query are actually evaluated. Please see Chapter 3 of this guide for more information on predicate evaluation sequence.
- **The statistics used** The optimizer used catalog statistics to help determine the access path at the time the statement was EXPLAINed. Unless you have historical statistics that happen to correspond to the time the EXPLAIN was executed, then you don't know what the statistics looked like at the time of the EXPLAIN, and if they are different now and can change the access path.
- **The input values** If you are using host variables in your programs then EXPLAIN knows nothing about the potential input values to those host variables. This makes it important for you to understand these values, what the most common values are, and if there is data skew relative to the input values.
- **The SQL statement** The SQL statement is not captured in the EXPLAIN tables, although some of the predicates are. If you EXPLAINed a statement dynamically, or via one of the tools, then you know what the statement looks like. However, if you've EXPLAINed a package or plan, then you are going to need to see the program source code.

- **The order of input to your transactions** Sure the SQL statement looks OK from an access path perspective, but what is the order of the input data to the statement? What ranges are being supplied? How many transactions are being issued? Is it possible to order the input or the data in the tables, in a manner in which it's most efficient. This is not covered in EXPLAIN output. These things are discussed further, however, throughout this guide.
- **The program source code** In order to fully understand the impact of the access path that a statement has used you need to see how that statement is being used in the application program. So, you should always be looking at the program source code of the program the statement is embedded in. How many times will the statement be executed? Is the statement in a loop? Can we avoid executing the statement? Is the program issuing 100 statements on individual key values when a range predicate and one statement would suffice? Is the programming performing programmatic joins? These are questions that can only be answered by looking at the program source code. The EXPLAIN output may show a perfectly good and efficient access path, but the statement itself could be completely unnecessary. (this is where a tool — or even a trace — can be very helpful in order to verify if the SQL statements executed really are what's expected).

Optimization Service Center and Visual EXPLAIN

As the complexity of managing and tuning various workloads continues to escalate, many database administrators (DBAs) are falling farther behind in their struggle to maintain quality of service while also keeping costs in check. There are many tools currently available (such as IBM Statistics Advisor, IBM Query Advisor etc.) designed to ease the workload of DBAs through a rich set of autonomic tools that help optimize query performance and workloads. You can use your favorite tool to identify and analyze problem SQL statements and to receive expert advice about statistics that you can gather to improve the performance of problematic and poorly performing SQL statements on a DB2 subsystem. It provides:

- The ability to snap the statement cache
- Collect statistics information
- Analyze indexes
- Group statements into workloads
- Monitor workloads
- An easy-to-understand display of a selected access path
- Suggestions for changing a SQL statement
- An ability to invoke EXPLAIN for dynamic SQL statements
- An ability to provide DB2 catalog statistics for referenced objects of an access path, or for a group of statements
- A subsystem parameter browser with keyword find capabilities
- The ability to graphically create optimization hints (a feature not found in the V8 Visual Explain product)

The OSC can be used to analyze previously generated explains or to gather explain data and explain dynamic SQL statements.

The OSC is available for DB2 V8, and DB2 9. If you are using DB2 V7, then you can use the DB2 Visual Explain product, which provides a subset of the functionality of the OSC.

DB2 Estimator

The DB2 Estimator product is another useful predictive analysis tool. This product runs on a PC, and provides a graphical user interface for entering information about DB2 tables, indexes, and queries. Table and index definitions and statistics can be entered directly, or imported view DDL files. SQL statements can be imported from text files.

The table definitions and statistics can be used to accurately predict database sizes. SQL statements can be organized into transactions, and then information about DASD models, CPU size, access paths, and transaction frequencies can be set. Once all of this information is input into Estimator, then capacity reports can be produced. These reports will contain estimates of the DASD required, as well as the amount of CPU required for an application. These reports are very helpful during the initial stage of capacity planning. That is, before any actual real programs or test data is available. The DB2 Estimator product will no longer be available after DB2 V8, and so you should download it today!

Predicting Database Performance

Lots of Developer and DBA time can be spent in the physical design of a database, and database queries based upon certain assumptions about performance. A lot of this time is wasted because the assumptions being made are not accurate. A lot of time is spent in meetings where people are saying things such as “Well that’s not going to work”, “I think the database will choose this access path”, or “We want everything as fast as possible.” Then, when the application is finally implemented, and performs horribly, people scratch their heads saying “...but we thought of everything!”

Another approach for large systems design is to spend little time considering performance during the development and set aside project time for performance testing and tuning. This frees the developers from having to consider performance in every aspect of their programming, and gives them incentive to code more logic in their queries. This makes for faster development time, and more logic in the queries means more opportunities for tuning (if all the logic was in the programs, then tuning may be a little harder to do). Let the logical designers do their thing, and let the database have a first shot at deciding the access path.

If you choose to make performance decisions during the design phase, then each performance decision should be backed by solid evidence, and not assumptions. There is no reason why a slightly talented DBA or developer can't make a few test tables, generate some test data, and write a small program or two to simulate a performance situation and test their assumptions about how the database will behave. This gives the database the opportunity to tell you how to design for performance. Reports can then be generated, and given to managers. It's much easier to make design decisions based upon actual test results.

Tools that you can use for testing statements, design ideas, or program processes include, but are not limited to:

- REXX DB2 programs
- COBOL test programs
- Recursive SQL to generate data
- Recursive SQL to generate statements
- Data in tables to generate more data
- Data in tables to generate statements

Generating Data

In order to simulate program access you need data in tables. You could simply type some data into INSERT statements, insert them into a table, and then use data from that table to generate more data. Say, for example, that you have to test various program processes against a PERSON_TABLE table and a PERSON_ORDER table. No actual data has been created yet, but you need to test the access patterns of incoming files of orders. You can key some INSERT statements for the parent table, and then use the parent table to propagate data to the child table. For example, if the parent table, PERSON_TABLE, contained this data:

```
PERSON_ID    NAME
          1  JOHN SMITH
          2  BOB RADY
```

Then the following statement could be used to populate the child table, PERSON_ORDER, with some test data:

```
INSERT INTO PERSON_ORDER
(PERSON_ID, ORDER_NUM, PRDT_CODE, QTY, PRICE)
SELECT PERSON_ID, 1, 'B100', 10, 14.95
FROM YLA.PERSON_TABLE
UNION ALL
SELECT PERSON_ID, 2, 'B120', 3, 1.95
FROM YLA.PERSON_TABLE
```

The resulting PERSON_ORDER data would look like this:

PERSON_ID	ORDER_NUM	PRDT_CDE	QTY	PRICE
1	1	B100	10	14.95
1	2	B120	3	1.95
2	1	B100	10	14.95
2	2	B120	3	1.95

The statements could be repeated over and over to add more data, or additional statements can be executed against the PERSON_TABLE to generate more PERSON_TABLE data.

Recursive SQL (DB2 V8, DB2 9) is an extremely useful way to generate test data. Take a look at the following simple recursive SQL statement:

```
WITH TEMP(N) AS
  (SELECT 1
   FROM SYSIBM.SYSDUMMY1
  UNION ALL
   SELECT N+1
   FROM TEMP
  WHERE N < 10)
SELECT N
FROM TEMP
```

This statement generates the numbers 1 through 10, one row each. We can use the power of recursive SQL to generate mass quantities of data that can then be inserted into DB2 tables, and ready for testing. The following is a piece of a SQL statement that was used to insert 300,000 rows of data into a large test lookup table. The table was quickly populated with data, and a test conducted to determine the performance. It was quickly determined that the performance of this large lookup table would not be adequate, but that couldn't have been known for sure without testing:

```
WITH LASTPOS (KEYVAL) AS
  (VALUES (0)
   UNION ALL
   SELECT KEYVAL + 1
   FROM   LASTPOS
   WHERE  KEYVAL < 9)
,STALETBL (STALE_IND) AS
  (VALUES 'S', 'F')
SELECT STALE_IND, KEYVAL
,CASE STALE_IND WHEN 'S' THEN
  CASE KEYVAL WHEN 0 THEN 1
    WHEN 1 THEN 2 WHEN 2 THEN 3
    WHEN 3 THEN 4 WHEN 4 THEN 4
    WHEN 5 THEN 6 WHEN 6 THEN 7
    WHEN 7 THEN 8 WHEN 8 THEN 9
    WHEN 9 THEN 10 END
      WHEN 'F' THEN
  CASE KEYVAL WHEN 0 THEN 11
    WHEN 1 THEN 12 WHEN 2 THEN 13
    WHEN 3 THEN 14 WHEN 4 THEN 15
    WHEN 5 THEN 16 WHEN 6 THEN 17
    WHEN 7 THEN 18 WHEN 8 THEN 19
    WHEN 9 THEN 20 END
      END AS PART_NUM
FROM   LASTPOS INNER JOIN
      STALETBL ON 1=1;
```

Generating Statements

Just as data can be generated so can statements. You can write SQL statements that generate statements. Say, for example, that you needed to generate singleton select statements against the EMP table to test a possible application process or scenario. You could possibly write a statement such as this to generate those statements:

```
SELECT 'SELECT LASTNAME, FIRSTNME ' CONCAT
      'FROM EMP WHERE EMPNO = ''' CONCAT
      EMPNO CONCAT ''';'
FROM   SUSAN.EMP
WHERE  WORKDEPT IN ('C01', 'E11')
AND    RAND() < .33
```

The Above query will generate SELECT statements for approximately 33% of the employees in departments “C01” and “E01”. The output would look something like this:

```
1
-----
SELECT LASTNAME, FIRSTNME FROM EMP WHERE EMPNO = '000030';
SELECT LASTNAME, FIRSTNME FROM EMP WHERE EMPNO = '000130';
SELECT LASTNAME, FIRSTNME FROM EMP WHERE EMPNO = '200310';
```

You could also use recursive SQL statements to generate statements. The following statement was used during testing of high performance INSERTs to an account history table. The following statement generated 50,000 random insert statements:

```

WITH GENDATA (ACCT_ID, HIST_EFF_DTE, ORDERVAL) AS
  (VALUES (CAST(2 AS DEC(11,0)), CAST('2003-02-01' AS DATE), CAST(1 AS
  FLOAT))
  UNION ALL
  SELECT ACCT_ID + 5, HIST_EFF_DTE, RAND()
  FROM   GENDATA
  WHERE  ACCT_ID < 249997)
SELECT 'INSERT INTO YLA.ACCT_HIST (ACCT_ID, HIST_EFF_DTE)' CONCAT
' VALUES(' CONCAT CHAR(ACCT_ID) CONCAT ',' CONCAT ''''
CONCAT CHAR(HIST_EFF_DTE,ISO) CONCAT '''' CONCAT ');'
FROM GENDATA ORDER BY ORDERVAL;

```

Determining Your Access Patterns

If you have access to input data or input transactions then it would be wise to build small sample test programs to determine the impact of these inputs on the database design. This would involve simple REXX or COBOL programs that contain little or no business logic, but instead just simple queries that simulated the anticipated database access. Running these programs could then give you ideas as to the impact of random versus sequential processing, or the impact of sequential processing versus skip sequential processing. It could also give you an idea of how buffer settings could impact performance, as well as whether or not performance enhancers such as sequential detection or index lookaside will be effective.

Simulating Access with Tests

Sitting around in a room and debating access paths and potential performance issues is a waste of time. Each potential disagreement among DBAs and application developers should be tested, the evidence analyzed, and a decision made. Write a simple COBOL program to try access to a table to determine what the proper clustering is, how effective compression will be for performance, whether or not one index performs over another, and whether or not adding an index will adversely impact INSERTs and DELETEs.

In one situation it was debated as to whether or not an entire application interface should utilize large joins between parent and child tables, or that all access should be via individual table access (programmatically joins) for the greatest in flexibility. Coding for both types of access would be extra programming effort, but what is the cost of the programmatic joins for this application? Two simple COBOL programs were coded against a test database; one with a two table programmatic join, and the other with the equivalent SQL join. It was determined that the SQL join consumed 30% less CPU than the programmatic join.

Monitoring

It's critical to design for performance when building applications, databases, and SQL statements. You've designed the correct SQL, avoided programmatic joins, clustered commonly accessed table in the same sequence, and have avoided inefficient repeat processing. Now, your application is in production and is running fine. Is there more you can save? Most certainly!

Which statement is the most expensive? Is it the tablespace scan that runs once per day, or the matching index scan running millions of times per day? Are all your SQL statements sub-second responders, and so you don't need tuning? What is the number one statement in terms of CPU consumption? All of these questions can be answered by monitoring your DB2 subsystems, and the applications accessing them.

DB2 provides facilities for monitoring the behavior of the subsystem, as well as the applications that are connected to them. This is primarily via the DB2 trace facility. DB2 has several different types of traces. In this chapter we'll discuss the traces that are important for monitoring performance, as well as how to use them effectively for proactive performance tuning.

DB2 Traces

DB2 provides a trace facility to help track and record events within a DB2 subsystem. There are six types of traces:

- Statistics
- Accounting
- Audit
- Performance
- Monitor
- Global

This chapter will cover the statistics, accounting and performance traces as they apply to performance monitoring. These traces should play an integral part in your performance monitoring process.

Statistics Trace

The data collected in the statistics trace allows you to conduct DB2 capacity planning and to tune the entire set of DB2 programs. The statistics trace reports information about how much the DB2 system services and database services are used. It is a system wide trace and should not be used for charge-back accounting. Statistics trace classes 1, 3, 4, 5, and 6 are the default classes for the statistics trace if statistics is specified yes in installation panel DSNTIPN. If the statistics trace is started using the START TRACE command, then class 1 is the default class.

The statistics trace can collect information about the number of threads connected, the amount of SQL statements executed, and amount of storage consumed within the database manager address space, deadlocks, timeouts, logging, buffer pool utilization, and much more. This information is collected at regular intervals for an entire DB2 subsystem. The interval is typically 10 or 15 minutes per record.

Accounting Trace

The accounting trace provides data that allows you to assign DB2 costs to individual authorization IDs and to tune individual programs. The DB2 accounting trace provides information related to application programs, including such things as:

- Start and stop times
- Number of commits and aborts
- The number of times certain SQL statements are issued
- Number of buffer pool requests
- Counts of certain locking events
- Processor resources consumed
- Thread wait times for various events
- RID pool processing
- Distributed processing
- Resource limit facility statistics

Accounting times are usually the prime indicator of performance problems, and most often should be the starting point for analysis. DB2 times are classified as follows:

- Class 1: This time shows the time the application spent since connecting to DB2, including time spent outside DB2.
- Class 2: This shows the elapsed time spent in DB2. It is divided into CPU time and waiting time.
- Class 3: This elapsed time is divided into various waits, such as the duration of suspensions due to waits for locks and latches or waits for I/O.

DB2 trace begins collecting this data at successful thread allocations to DB2 and writes a completed record when the thread terminates or in some cases when the authorization ID changes. Having the accounting trace active is critical for proper performance monitoring, analysis, and tuning. When an application connects to DB2 it is executing across address spaces, and the DB2 address spaces are shared by perhaps thousands of users across many address spaces. The accounting trace provides information about the time spent within DB2, as well as the overall application time. Class 2 time is a component of class 1 time, and class 3 time a component of class 2 time.

Accounting data for class 1 (the default) is accumulated by several DB2 components during normal execution. This data is then collected at the end of the accounting period; it does not involve as much overhead as individual event tracing. On the other hand, when you start class 2, 3, 7, or 8, many additional trace points are activated. Every occurrence of these events is traced internally by DB2 trace, but these traces are not written to any external destination. Rather, the accounting facility uses these traces to compute the additional total statistics that appear in the accounting record when class 2 or class 3 is activated. Accounting class 1 must be active to externalize the information.

We recommend you set accounting classes 1,2,3,7,8. Be aware that this can add between 4% and 5% of your overall system CPU consumption. However, if you are already writing accounting classes 1,2,3, then adding 7 and 8 typically should not add much overhead. Also, if you are using an online performance monitor then it could already have these classes started. If that is the case then adding SMF as a destination for these classes should not add any CPU overhead.

Performance Trace

The performance trace provides information about a variety of DB2 events, including events related to distributed data processing. You can use this information to further identify a suspected problem or to tune DB2 programs and resources for individual users or for DB2 as a whole. To start a performance trace, you must use the `-START TRACE(PERFM)` command. Performance traces cannot be automatically started. Performance traces are expensive to run, and consume a lot of CPU. They also collect a very large volume of information. Performance traces are usually run via an online monitor tool, or the output from the performance trace can be sent to SMF and then analyzed using a monitor reporting tool, or sent to IBM for analysis.

Because performance traces can consume a lot of resources and generate a lot of data, there are a lot of options when starting the trace to balance the information desired with the resources consumed. This includes limited the trace data collected by plan, package, trace class, and even IFCID.

Performance traces are typically utilized by online monitor tools to track a specific problem for a given plan or package. Reports can then be produced by the monitor software, and can detail SQL performance, locking, and many other detailed activities. Performance trace data can also be written to SMF records, and batch reporting tools can read those records to produce very detailed information about the execution of SQL statements in the application.

Accounting and Statistics Reports

Once DB2 trace information has been collected you can create reports by reading the SMF records. These reports are typically produced by running reporting software provided by a performance reporting product. These reports are a critical part of performance analysis and tuning. You should get familiar with the statistics and accounting reports. They are the best gauge as to the health of a DB2 subsystem, and the applications using it. It is also the best way to monitor performance trends, and to proactively detect potential problems before they become critical.

Statistics Report

Since statistics records are collected typically at 10 minute or 15 minute intervals quite a few records can be collected on a daily basis. Your reporting software should be able to produce either summary reports, which can gather and summarize the data for a period of time, or detail reports, which can report on every statistics interval. Start with a daily summary report, and look for specific problems within the DB2 subsystem. Once you detect a problem then you can produce a detailed report to determine the specific period of time that the problem occurred, and also coordinate the investigation with detailed accounting reports for the same time period in an effort to attribute the problem to a specific application or process. Some of the things to look for in a statistics report:

- **RID Pool Failures** There should be a section of the statistics report that reports the usage of the RID pool for things such as list prefetch, multiple index access, and hybrid joins. The report will also indicate RID failures. There can be RDS failures, DM failures, and failures due to insufficient size. If you are getting failures due to insufficient storage you can increase the RID pool size. However, if you are getting RDS or DM failures in the RID pool then there is a good chance that the access path selected is reverting to a table space scan. In these situations it is important to determine which applications are getting these RID failures. Therefore, you need to produce a detailed statistics report that can identify the time of the failures, and also produce detailed accounting reports that will show which threads are getting the failures. Further investigation will have to be performed to determine the packages within the plan, and DB2 EXPLAIN can be used to determine which statements are using list prefetch, hybrid join, or multi-index access. You may have to test the queries to determine if they are indeed the one's getting these failures, and if they are you'll have to try to influence the optimizer to change the access path (see Chapter 3 for SQL tuning).
- **Bufferpool Issues** One of the most valuable pieces of information coming out of the statistics report is the section covering buffer utilization and performance. For each buffer pool in use the report will include the size of the pool, sequential and random getpages, prefetch operations, pages written, and number of sequential I/O's, random I/O's, and write I/O's, plus much more. Also reported are the number of times certain buffer thresholds have been hit. One of the things to watch for are the number of synchronous reads for sequential access, which may be an indication that the number of pages is too small and pages for a sequential prefetch are stolen before they are used. Another thing to watch is whether or not any critical thresholds are reached, if there are write engines not available, and whether or not deferred write thresholds are triggering. It's also important to monitor the number of getpages per synchronous I/O, as well as the buffer hit ratio. Please see Chapter 8 for information about subsystem tuning.

- **Logging Problems** The statistics report will give important information about logging. This includes the number of system checkpoints, number of reads from the log buffer, active log data sets, or archived log data sets, number of unavailable output buffers, and total log writes. This could give you an indication as to whether or not you need to increase log buffer sizes, or investigate frequent application rollbacks or other activities that could cause excessive log reads. Please see Chapter 8 for information about subsystem tuning.
- **EDM Pool Hit Ratio** The statistics report will show how often database objects such as DBD's, cursor tables, and package tables are requested as well as how often those requests have to be satisfied via a disk read to one of the directory tables. You can use this to determine if the EDM pool size needs to be increased. You also get statistics about the use of dynamic statement cache and the number of times statement access paths were reused in the dynamic statement cache. This could give you a good indication about the size of your cache, and its effectiveness, but it could also give you an indication of the potential reusability of the statements in the cache. Please see Chapter 8 for more information about the EDM pool, and Chapter 3 for information about tuning dynamic SQL.
- **Deadlocks and Timeouts** The statistics report will give you a subsystem wide perspective on the number of deadlocks and timeouts your applications have experienced. You can use this as an overall method of detecting deadlocks and timeouts across all applications. If the statistics summary report shows a positive count you can use the detailed report to find out at what time the problems are occurring. You can also use accounting reports to determine which applications are experiencing the problem.

This has only been a sample of the fields on the statistics report, and the valuable information they provide. You should be using your statistics reports on a regular basis, and using your monitoring software documentation, along with the DB2 Administration Guide (DB2 V7, DB2 V8) or DB2 Performance Monitoring and Tuning Guide (DB2 9) to interpret the information provided.

Accounting Report

An accounting report will read the SMF accounting records to produce thread or application level information from the accounting trace. These reports typically can summarize information at the level of a plan, package, correlation id, authorization id, and more. In addition, you can also have the option to produce one report per thread. This accounting detail report can give very detailed performance information for the execution of a specific application process. If you have accounting classes 1,2,3,7, and 8 turned on, then the information will be reported at both the plan and package level.

You can use the accounting report to find specific problems within certain applications, programs, or threads. Some of the things to look for in an accounting report include:

- **Class 1 and Class 2 Timings** Class 1 times (elapsed and CPU) include the entire application time, including the time spent within DB2. Class 2 is a component of class 1, and represents the amount of time the application spent in DB2. The first question to ask when an application is experiencing a performance problem is “where is the time being spent?” The first indication of the performance issue being DB2 will be a high class 2 time relative to the class 1 time. Within class 2 you could be having a CPU issue (CPU time represents the majority of class 2 time), or a wait issue (CPU represents very little of the overall class 2 time, but class 3 wait represents most of the time), or maybe your entire system is CPU bound (Class 2 overall elapsed is not reflected in class 2 CPU and class 3 wait time combined).
- **Buffer Usage** The accounting report contains buffer usage at the thread level. This information can be used to determine how a specific application or process is using the buffer pools. If you have situations in which certain buffers have high random getpage counts you may want to look at which applications are causing those high number of random getpages. You can use this thread level information to determine which applications are accessing buffers randomly versus sequentially. Then perhaps you can see which objects the application uses, and use this information to separate sequentially accessed objects from randomly accessed objects into different buffer pools (see Chapter 8 on subsystem tuning). The buffer pool information in the accounting report will also indicate just how well the application is utilizing the buffers. The report can be used during buffer pool tuning to determine the impact of buffer changes on an application.
- **Package Execution Times** If accounting classes 7 and 8 are turned on then the account report will show information about the DB2 processing on a package level. This information is very important for performance tuning because it allows you to determine which programs in a poorly performing application should be reviewed first.
- **Deadlocks, Timeouts, and Lock Waits** The accounting report includes information about the number of deadlocks and timeouts that occurred on a thread level. It also reports the time the thread spent waiting on locks. This will give you a good indication as to whether or not you need to do additional investigation into applications that are having locking issues.
- **Excessive Synchronous I/O's** Do you have a slow running job or online process? Exactly what is slow about that job or process? The accounting report will tell you if there are a large number of excessive random synchronous I/O's being issued, and how much time the application spends waiting on I/O. The information in the report can also be used to approximately determine the efficiency of your DASD by simply dividing the number of synchronous I/O's into the total synchronous I/O wait time.
- **RID Failures** The accounting report does give thread level RID pool failure information. This is important in determining if you have access path problems in a specific application.

- **High Getpage Counts and High CPU Time** Often it is hard to determine if an application is doing repeat processing when there are not a lot of I/O's being issued. You should use the accounting report to determine if your performance problem is related to an inefficient repeat process. If the report shows a very high getpage count, or that the majority of elapsed time is actually class 2 CPU time, then that may be an indication of an inefficient repeat process in one of the programs for the plan. You can use the package level information to determine which program uses the most CPU, and try to identify any inefficiencies in that program.

Online Performance Monitors

DB2 can write accounting, statistics, monitor, and performance trace data to line buffers. This allows for online performance reporting software to read those buffers, and report on DB2 subsystem and application performance in real time. These online monitors can be used to review subsystem activity and statistics, thread level performance information, deadlocks and timeouts, I/O activity, and dynamically execute and report on performance traces.

Overall Application Performance Monitoring

As much as we hate to hear it, there's no silver bullet for improving overall large system performance. We can tune the DB2 subsystem parameters, logging and storage, but often we're only accommodating a bad situation. Here the "80/20" rule applies; you'll eventually have to address application performance.

There's a way to quickly assess application performance and identify the significant offenders and SQL statements causing problems. You can quickly identify the "low-hanging fruit," report on it to your boss, and change the application or database to support a more efficient path to the data. Management support is a must, and an effective manner of communicating performance tuning opportunities and results is crucial.

Setting the Appropriate Accounting Traces

DB2 accounting traces will play a valuable role in reporting on application performance tuning opportunities. DB2 accounting traces 1, 2, 3, 7, and 8 must be set to monitor performance at the package level. Once you do that, you can further examine the most expensive programs (identified by package) to look for tuning changes. This reporting process can serve as a quick solution to identifying an application performance problem, but can be incorporated into a long-term solution that identifies problems and tracks changes.

There's been some concern about the performance impact of this level of DB2 accounting. The IBM DB2 Administration Guide (DB2 V7, DB2 V8) or the DB2 Performance Monitoring and Tuning Guide (DB2 9) states that the performance impact of setting these traces is minimal and the benefits can be substantial. Tests performed at a customer site demonstrated an overall system impact of 4.3 percent for all DB2 activity when accounting classes 1, 2, 3, 7, and 8 are started. In addition, adding accounting classes 7 and 8 when 1, 2, and 3 are already started has nominal impact, as does the addition of most other performance monitor equivalent traces (i.e. your online monitor software).

Summarizing Accounting Data

To effectively communicate application performance information to management, the accounting data will must be organized and summarized up to the application level. You need a reporting tool that formats DB2 accounting traces from System Management Facilities (SMF) files to produce the type of report you're interested in. Most reporting tools can produce DB2 accounting reports at a package summary level. Some can even produce customized reports that can filter only the information you want out of the wealth of information in trace records.

You can process whatever types of reports you produce so that a concentrated amount of information about DB2 application performance can be extracted. This information is reduced to the amount of elapsed time and CPU time the application consumes daily and the number of SQL statements each package issues daily. This highly specific information will be your first clue as to which packages provide the best DB2 tuning opportunity. The following example is from a package level report with the areas of interest highlighted (Package Name, Total DB2 Elapsed, Total SQL Count, Total DB2 TCB):

		- AVERAGE -	----	TOTAL	----
PACKAGE EXECUTIONS		HHH:MM:SS.TTT		HHH:MM:SS.TTT	
-----	DB2 TCB.....	0.008		25:18.567	
COLL ID CPG2SU01	I/O.....	0.205		010:28:50.335	
<u>PROGRAM FHPRDA2</u>	LOCK/LATCH...	0.000		43.652	
	OTHER RD I/O.	0.000		1:29.015	
	HHH:MM:SS.TTT OTHER WR I/O.	0.000		0.000	
AVG DB2 ELAPSED	0.214	DB2 SERVICES.	0.000	0.001	
<u>TOTAL DB2 ELAPSED</u>	<u>010:55:40.211</u>	LOG QUIESCE..	0.000	0.000	
		DRAIN LOCK...	0.000	0.000	
AVG SQL COUNT	86.9	CLAIM RELEASE	0.000	0.000	
<u>TOTAL SQL COUNT</u>	<u>15948446</u>	ARCH LOG READ	0.000	0.000	
AVG STORPROC EXECUTED	0.0	PG LATCH CONT	0.000	0.000	
TOT STORPROC EXECUTED	0	WT DS MSGS	0.000	0.000	
AVG UDFS SCHEDULED	0.0	WT GLBL CONT	0.000	0.000	
TOT UDFS SCHEDULED	0	GLB CHLD L-LK	0.000	0.000	
		GLB OTHR L-LK	0.000	0.000	
		GLB PSET P-LK	0.000	0.000	
		GLB PAGE P-LK	0.000	0.000	
		GLB OTHR P-LK	0.000	0.000	
		OTHER DB2....	0.000	58.196	

If you lack access to a reporting tool that can filter out just the pieces of information desired, you can write a simple program in any language to read the standard accounting reports and pull out the information you need. REXX is an excellent programming language well-suited to this type of "report scraping," and you can write a REXX program to do such work in a few hours. You could write a slightly more sophisticated program to read the SMF data directly to produce similar summary information if you wish to avoid dependency on the reporting software. Once the standard reports are processed and summarized, all the information for a specific interval (say one day) can appear in a simple spreadsheet. You can sort the spreadsheet by CPU descending. With high consumers at the top of the report, the low hanging fruit is easy to spot. The following spreadsheet can be derived by extracting the fields of interest from a package level summary report:

Package	Executions	Total Elapsed	Total CPU	Total SQL	Elaps/Execution	CPU/Execution	Elapsed/SQL	CPU/SQL
ACCT001	246745	75694.2992	5187.4262	1881908	0.3067	0.021	0.0402	0.0027
ACCT002	613316	26277.2022	4381.7926	1310374	0.0428	0.0071	0.02	0.0033
ACCTB01	8833	4654.4292	2723.1485	531455	0.5269	0.3082	0.0087	0.0051
RPTS001	93	6998.7605	2491.9989	5762	75.2554	26.7956	1.2146	0.4324
ACCT003	169236	33439.2804	2198.0959	1124463	0.1975	0.0129	0.0297	0.0019
RPTS002	2686	2648.3583	2130.2409	2686	0.9859	0.793	0.9859	0.793
HRPK001	281	4603.1262	2017.7179	59048	16.3812	7.1804	0.0779	0.0341
HRPKB01	21846	3633.5143	2006.6083	316746	0.1663	0.0918	0.0114	0.0063
HRBKB01	505	2079.5351	1653.5773	5776	4.1178	3.2744	0.36	0.2862
CUSTB01	1	4653.9935	1416.6254	7591111	4653.9935	1416.6254	0.0006	0.0001
CUSTB02	1	3862.1498	1399.9468	7971317	3862.1498	1399.9468	0.0004	0.0001
CUST001	246670	12636.0232	1249.7678	635911	0.0512	0.005	0.0198	0.0019
CUSTB03	280	24171.1267	1191.0164	765906	86.3254	4.2536	0.0315	0.0015
RPTS003	1	5163.3568	884.0148	1456541	5163.3568	884.0148	0.0035	0.0006
CUST002	47923	10796.5509	875.252	489288	0.2252	0.0182	0.022	0.0017
CUST003	68628	3428.4162	739.4523	558436	0.0499	0.0107	0.0061	0.0013
CUSTB04	2	1183.2068	716.2694	3916502	591.6034	358.1347	0.0003	0.0001
CUSTB05	563	1232.2111	713.9306	1001	2.1886	1.268	1.2309	0.7132

Look for some simple things to choose the first programs to address. For example, package ACCT001 consumes the most CPU per day, and issues nearly 2 million SQL statements. Although the CPU consumed per statement on average is low, the sheer quantity of statements issued indicates an opportunity to save significant resources. If just a tiny amount of CPU can be saved, it will quickly add up. The same applies to package ACCT002 and packages RPTS001 and RPTS002. These are some of the highest consumers of CPU and they also have a relatively high average CPU per SQL statement. This indicates there may be some inefficient SQL statements involved. Since the programs consume significant CPU per day, tuning these inefficient statements could yield significant savings.

ACCT001, ACCT002, RPTS001, and RPTS002 represent the best opportunities for saving CPU, so examine those first. Without this type of summarized reporting, it's difficult to do any sort of truly productive tuning. Most DBAs and systems programmers who lack these reports and look only at online monitors or plan table information are really just shooting in the dark.

Reporting to Management

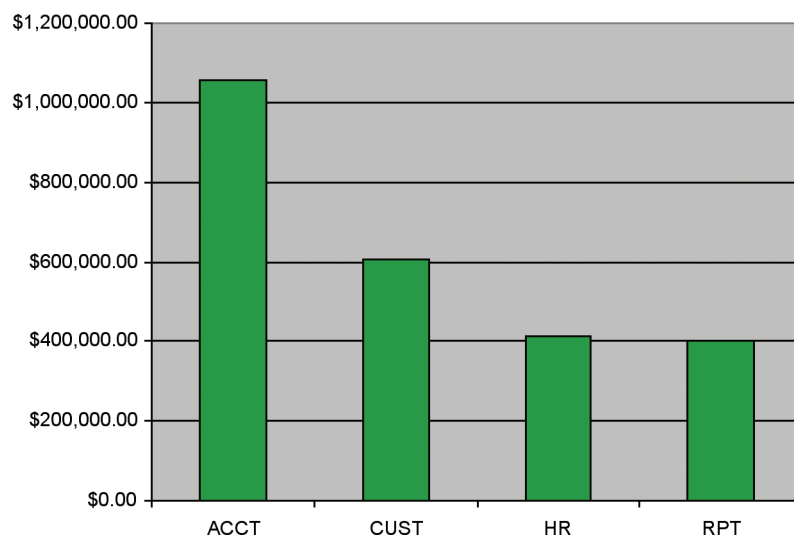
To do this type of tuning, you need buy-in from management and application developers. This can sometimes be the most difficult part because, unfortunately, most application tuning involves costly changes to programs. One way to demonstrate the potential ROI for programming time is to report the cost of application performance problems in terms of dollars. This is easy and amazingly effective!

The summarized reports can report on information on the application level. An in-house naming standard can be used to combine all the performance information from various packages into application-level summaries. This lets you classify applications and address the ones that use the most resources.

For example, if the in-house accounting application has a program naming standard where all program names begin with "ACCT," then the corresponding DB2 package accounting information can be grouped by this header. Thus, the DB2 accounting report data for programs ACCT001, ACCT002, and ACCT003 can be grouped together, and their accounting information summarized to represent the "ACCT" application.

Most capacity planners have formulas for converting CPU time into dollars. If you get this formula from the capacity planner, and categorize the package information by application, you can easily turn your daily package summary report into an annual CPU cost per application. The following shows a simple chart developed using an in-house naming standard and a CPU-to-dollars formula. Give this report to the big guy and watch his head spin! This is a really great tool for getting those resources allocated to get the job done.

ANNUAL CPU COST



Make sure you produce a "cost reduction" report, in dollars, once a phase of the tuning has completed. This makes it perfectly clear to management what the tuning efforts have accomplished and gives incentive for further tuning efforts. Consider providing a visual representation of your data. A bar chart with before and after results can be highly effective in conveying performance tuning impact.

Finding the Statements of Interest

Once you've found the highest consuming packages and obtained management approval to tune them, you need additional analysis of the programs that present the best opportunity. Ask questions such as:

- Which DB2 statements are executing and how often?
- How much CPU time is required?
- Is there logic in the program that's resulting in an excessive number of unnecessary database calls?
- Are there SQL statements with relatively poor access paths?

Involve managers and developers in your investigation. It's much easier to tune with a team approach where different team members can be responsible for different analysis.

There are many ways to gather statement-level information:

- Get program source listings and plan table information
- Watch an online monitor
- Run a short performance trace against the application of interest

Performance traces are expensive, sometimes adding as much as 20 percent to the overall CPU costs. However, a short-term performance trace may be an effective tool for gathering information on frequent SQL statements and their true costs.

If plan table information isn't available for the targeted package, then you can rebind that package with EXPLAIN(YES). If it's hard to get the outage to rebind EXPLAIN(YES) or a plan table is available for a different owner id, you could also copy the package with EXPLAIN(YES) (for example execute a BIND into a special/dummy collection-id) rather than rebinding it.

The following example shows PLAN_TABLE data for two of the most expensive programs in our example.

Q_QB_PL	PNAME	MT	TNAME	T_NO	AT	MC	ACC_NM	IX	NJ_CUJOG	PF
000640-01-01	ACCT001	0	PERSON_ACCT	1	I	1	AAIXPAC0	N	NNNNN	S
000640-01-02	ACCT001	3		0		0		N	NNNYN	
001029-01-01	RPTS001	0	PERSON	1	I	1	AAIXPRC0	N	NNNNN	S
001029-01-02	RPTS001	4	ACCOUNT	2	I	1	CCIXACC0	N	NNNNN	L
001029-01-03	RPTS001	1	ACCT_ORDER	3	I	2	CCIXAOC0	N	NNNNN	

Here, our most expensive program issues a simple SQL statement with matching index access to the PERSON_ACCT table, and it orders the result, which results in a sort (Method=3). The programmer, when consulted, advised that the query rarely returns more than a single row of data. In this case, a bubble sort in the application program replaced the DB2 sort. The bubble sort algorithm was almost never used because the query rarely returned more than one row, and the CPU associated with DB2 sort initialization was avoided. Since this query was executing many thousands of times per day, the CPU savings were substantial.

Our high-consuming reporting program is performing a hybrid join (Method=4). While we don't frown on hybrid joins, our system statistics were showing us that there were Relational Data Server (RDS) subcomponent failures in the subsystem. We wondered whether this query was causing an RDS failure, and reverted to a tablespace scan. This turned out to be true. We tuned the statement to use nested loop over hybrid join, the RDS failures and subsequent tablespace scan were avoided, and CPU and elapsed time improved dramatically.

While these statements may have not caught someone's eye just by looking at EXPLAIN results, when combined with the accounting data, they screamed for further investigation.

The Tools You Need

The application performance summary report identifies applications of interest and provides the initial tool to report to management. You can also use these other tools, or pieces of documentation, to identify and tune SQL statements in your most expensive packages:

- **Package Report** A list of all packages in your system sorted by the most expensive first (usually by CPU). This is your starting point for your application tuning exercise. You use this report to identify the most expensive applications or programs. You also use the report to sell the idea of tuning to management and to drill down to more detailed levels in the other reports.

- **Trace or Monitor Report** You can run a performance trace or watch your online monitor for the packages identified as high consumers in your package report. This type of monitoring will help to drill down to the high-consuming SQL statements within these packages.
- **Plan Table Report** Run extractions of plan table information for the high-consuming programs identified in your package report. You may quickly find some bad access paths that can be tuned quickly. Don't forget to consider the frequency of execution as indicated in the package report. Even a simple thing such as a small sort may be really expensive if executed often.
- **Index Report** Produce a report of all indexes on tables in the database of interest. This report should include the index name, table name, columns names, columns sequence, cluster ratio, clustering, first key cardinality, and full key cardinality. Use this report when tuning SQL statements identified in the plan table or trace/monitor report. There may be indexes you can take advantage of, add or change — or even drop. Indexes not used will create an overhead for Insert, Delete, Update processing as well as utilities.
- **DDL or ERD** You're going to need to know about the database. This includes relationships between tables, column data types, and knowing where data is. An Entity Relationship Diagram (ERD) is the best tool for this, but if none is available, you can print out the Data Definition Language (DDL) SQL statements used to create the tables and indexes. If the DDL isn't available, you can use a tool such as DB2LOOK (yes, you can use this against a mainframe database) to generate the DDL.

Don't overlook the importance of examining the application logic. This has to do primarily with the quantity of SQL statements being issued. The best performing SQL statement is the one that is never issued, and it's surprising how often application programs will go to the database when they don't have to. The program may be executing the world's best-performing SQL statements, but if the data isn't used, then they're really poor-performing statements.

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

HB05ESMDBM01E MP319200807