IBM DB2 for i Offers New Ways to Enhance Your BI Performance

Published September 1, 2008 by System i Network

Written by Mike Cain – mcain@us.ibm.com

Business intelligence (BI) is an umbrella term or concept that describes the storing of data, processing of data, and ultimately the presentation of useful and insightful information to a set of users. Data warehousing is another concept under that umbrella and describes techniques for building up, storing, and accessing data as part of a BI solution. In a BI environment that writes, stores, and accesses vast quantities of data, database management, system performance, and scalability are crucial to achieving a good return on investment and meeting end-user goals. In this article, I discuss some best practices for storing, accessing, and processing data in support of BI.

The Data Model

IBM DB2 for i (formerly DB2 for i5/OS) is an integrated part of the IBM i operating system (formerly i5/OS) running on IBM Power servers (formerly System i). Because the IBM i operating system handles all the storage management for DB2, the database architect or engineer is free of certain responsibilities, such as creating table spaces, developing physical storage allocations, and ensuring parallelism at the database-object level. Although these are serious considerations for good performance when you use other database management systems, they are not part of the agenda when it comes to DB2 for i.

However, with any data-centric application, a proper data model is important not only for query ease of use but also for performance. With BI applications, the data model is usually normalized. In many environments, we use a specific data model known as a star schema or snowflake schema. This data model usually manifests itself as a central fact table surrounded by various dimension tables. We can think of the fact table as a repository where we add facts, metrics, or measures, such as sales figures. We can think of the dimension tables as descriptive information for various subjects, such as customers, retail outlets, and dates. The dimension table rows are related to the fact table rows by a key column (one to many), and a fact table row is related to each dimension table by a set of key columns. You should use only one key column to represent the relationship between any two tables. In other words, when you join a fact table to a dimension table, the join condition is only one column equal to another column (e.g., a.join_column = b.join_column). Furthermore, the respective key columns should be of the same type and have the same attributes. Many people ask me whether a certain column type is best for joining. DB2 for i can efficiently make the join by using any column type. If you are using meaningless surrogate keys to identify rows, you can use a simple integer column type effectively.

Using constraints is best practice when implementing the data model. Letting DB2 manage and enforce the business rules is a good way to give more of the work to the database management system and can pay dividends during query optimization. Using primary and unique key constraints lets the database engine ensure that no duplicates exist, and lets the query optimizer understand the query selectivity and column cardinality in this regard. Referential integrity (RI) constraints let DB2 manage and enforce the relationship between tables. During query optimization, RI constraints are used to better understand the potential behavior of various join orders, as well as to let the query optimizer possibly eliminate tables from the query. This form of query rewrite helps eliminate I/O operations. When you specify the primary key, unique key, or RI constraints, DB2 creates radix indexes to support these constraints. These indexes are available to the query optimizer and database engine for use with queries.

Another type of constraint is the check constraint. When implemented, this feature guarantees that the contents of a column meet a given business rule. Again, constraints relieve the application programmer from building and remembering to specify the business rule and, more important, the constraint lets the query optimizer understand the data and consequently help with query rewrites. A simple example to illustrate this idea involves a check constraint specifying that the column Color cannot be equal to RED, which means that DB2 does not allow any RED values in the column. If a query specifies local selection Where Color = 'RED', the query optimizer can rewrite the query and add logic for early exit. In other words, the check constraint guarantees that the query optimizer will not run the query, because no rows match the value 'RED', guaranteed by the check constraint.

On many other database management systems, local or distributed table partitioning is an important and sometimes vital component to achieving good performance. The database administrator spends a lot of time and effort designing, implementing, and managing the partitioning scheme. Although DB2 for i certainly has the ability to support partitioned tables, partitioning is primarily recommended to overcome growth limitations. There are a few reasons why we generally don't recommend table partitioning as means of improving performance:

- DB2 for i has many state-of-the-art methods and strategies for fast data access and data processing.
- DB2 for i can use parallelism without partitioning.
- Table partitioning requires a lot of database administration and engineering effort.

If a non-partitioned table is expected to exceed the current limit of 1.7 TB or 4.2 billion rows, both range and hash partitioning are available to effectively raise the limit by a factor of 256. Otherwise, keeping the implementation simple and productive is best.

When the time comes to implement and populate the data model on other database management systems, many database administrators get involved with physical storage allocation. With DB2 for i, you don't need to create table spaces or specify storage parameters. As a matter of fact, the IBM i operating system handles the creation, persistence, and storage of objects, including DB2 tables and indexes. In effect, the database objects are spread across all available disk units to minimize contention and

performance hotspots and maximize throughput. This technique also lets DB2 make the most of I/O parallelism, such as reading from and writing to multiple disk units simultaneously.

Indexing

After the data model is implemented, the single most important item regarding great query performance is adequate and proper indexing. Indexes are database objects that facilitate efficient reference, letting the database engine quickly identify and access rows of interest. Within DB2 for i, indexes also play an important role with query optimization. Indexes not only provide the query optimizer with more choices for data access, but they also provide information about the data. The index object acts as an efficient and accurate source of statistics, and good statistics help the optimizer make better choices.

For a BI data model, you should provide indexes that help maintain integrity and relationships. Constraints and the indexes associated with them do this. For queries, indexes are created to support local selection, joining, grouping, and ordering.

With a star schema or snowflake schema data model, multidimensional queries constrain the rows in the fact table by selecting rows of interest from one or more dimension tables and then joining those rows to the corresponding rows in the fact table. Given this behavior, indexes created for the local selection columns of the dimension tables help in avoiding full table scans. Indexes created explicitly or implicitly (via constraints) on the join columns help facilitate the best join order and join implementation. Because the fact table usually has no columns referenced with local selection, the indexing strategy is focused on the foreign key or join columns. This is where some unique DB2 for i optimization technology comes into play.

First, we need to understand the concept of index ANDing/ORing. Basically, this means that we can use more than one index to identify a set of rows in a table. By using more than one index, we can eliminate physical I/O and speed up the query. For index ANDing/ORing, we use one index to satisfy one part of the local selection criteria and to identify a set of rows by relative row number (RRN). Another index is used to satisfy another part of the local selection, resulting in another set of RRNs, and so on. The sets of RRNs are then merged with the appropriate logic from the Where clause (ANDed/ORed together) to produce a final list of RRNs. This list represents all or most of the local selection and is used to access a much smaller set of rows in the table. This technique is particularly effective for larger tables.

As I mentioned earlier, when it comes to querying a star schema, the rows of interest in the fact table are identified by inner joins with one or more dimension tables. Inner joins act like local selections in that they can eliminate rows from the result set. To test the row for inclusion, you need to have I/O performed. This requirement can potentially make a query against a large fact table time-consuming. To avoid this situation, DB2 employs a clever technique known as "look ahead predicate generation" (LPG). This technique uses the join criteria between the fact table and the dimension tables to generate local selection predicates on the fact table itself. After the optimizer rewrites the query to have local selection predicates, DB2 can apply the index ANDing technique to identify the relatively narrow set of rows in the

fact table. What does this have to do with indexing? Everything! Simply put, it is best practice to create an encoded vector index (EVI) on each foreign key of the fact table. Because of LPG, DB2 can use these DB objects with index ANDing. Having both EVIs and radix indexes (via constraints) on these columns is perfectly acceptable and advantageous.

Unlike many other database management systems (DBMSs), DB2 handles much of the data structure management and integrity for you. For example, indexes are maintained, balanced, and protected automatically. If an index object is damaged or otherwise unusable, DB2 recognizes this and attempts to fix the object. Whether the index is totally rebuilt or incrementally adjusted depends on the threshold setting of System Managed Access Path Protection (SMAPP). The threshold determines which indexes should be rebuilt and which indexes need to be protected by means of journaling. Journaling introduces additional workload yet provides faster recoverability should an index become unusable. When you're initially loading data into a data model, best practice is to drop unnecessary indexes and rebuild them after the data model is fully populated. Another best practice is to review your availability and recovery requirements and set the SMAPP threshold to a suitable level. When you're initially loading data, setting the SMAPP threshold to *NONE will avoid the potential overhead of journaling the index changes and increase performance. If any indexes are damaged during the process, you can rebuild them before you go live with the data model. Be sure to change the SMAPP threshold setting back to a value that supports your requirements.

When you have the data model in production, and it's being queried, you should use the systemwide index advice. DB2 provides this advice automatically for every query, and you can access it through Navigator for i5/OS. Analyzing the advice provides you with additional insight and opportunities to help you fine-tune your queries.

Materialized Query Tables

If there's one way to cheat and produce the answer to the query before the user issues a request, it has to be the Materialized Query Table (MQT). This object is a true database table with special characteristics. The MQT is based on one or more tables with a signature query and the results of this query. The magic occurs when the query optimizer takes the user's SQL request and matches it to the MQT. The user's query is rewritten internally to use the MQT instead of the base table or tables. Given that the MQT holds the precomputed query results, the user experiences faster query response time, and DB2 uses fewer computing and I/O resources to deliver the result. It is beneficial to design, build, and maintain MQTs in a BI environment to provide "point at a time" query results fast and effectively.

Database Parallelism via DB2 Symmetric Multiprocessing

When it comes to query performance, the laws of physics prevail. But maybe we can use them to our advantage. Accessing and processing data takes time. Queries typically march along in linear fashion, performing work synchronously — one phase after another. What if this stream of work was broken up into multiple parts, and these parts simultaneously executed parallel to one another? We would be doing more in less time. We would also be trading resources for time. In other words, using more

resources (i.e., more CPUs, more I/O bandwidth) to go faster and finish sooner. Database parallelism embodies this concept.

With the installation of the optional OS feature DB2 Symmetric Multiprocessing (SMP), the query optimizer is free to break up all or part of the query, and the database engine is free to run the multiple parts of the query in parallel. For example, a full table scan can be divided into multiple ranges of rows in which different threads process a respective range of data in parallel, which results in a faster completion of the table scan. The same idea applies to creating indexes. DB2 for i has sophisticated technology that makes judicious use of SMP and multiple processors by running multiple tasks simultaneously. This technology speeds up the creation of indexes and is especially important in very large database environments or when very fast data access and processing is paramount.

A Programmatic Approach to Database Parallelism

For index creation and queries, the DB2 for SMP feature can help. For other types of workloads, such as non-SQL data processing, parallelism is achieved with a programmatic approach. For example, during the extraction/transformation/loading (ETL) process, we can achieve higher throughput by letting the ETL tool or program execute multiple concurrent streams of processing with multiple database connections. Keep in mind that SQL Insert, Update, and Delete operations are not parallel-enabled. So again, if you are operating on large numbers of rows, it is best to take control and programmatically identify a range of data to process and issue multiple concurrent operations, one for each range of data.

Isolation and Locking

Database commitment control and isolation levels provide a means to determine when a transaction is complete and when rows associated with a given transaction are available to other users. DB2 for i uses row-level locking. In other words, depending on the isolation level specified within the job or database connection and the SQL operation, rows are locked and unavailable to other users. As the number of row locks increases into the millions, more effort and time is required to track and manage these locks — not to mention that the locked rows are unavailable for the duration of the transaction. You should fully understand and use the appropriate isolation level. It is always best to use the lowest level of isolation required to obtain the proper application behavior. If you're using journaling and commitment control to support recoverability, set up and configure the journal and journal receiver objects for best performance, and issue commit operations at the appropriate intervals to balance the number of locks held with the bundling of I/O operations.

Query and Reporting Tools

SQL is the most strategic and advanced interface for issuing data-definition and data-manipulation operations. Over the past five releases of DB2 for i, SQL itself, the query optimizer, and the database engine have undergone significant enhancement. The latest functional, performance, and tooling enhancements are available only when you're using the SQL Query Engine (SQE). DB2 for i has multiple query interfaces (e.g., SQL, OPNQRYF, QUERY/400, query API), and only SQL requests can use SQE and

take advantage of the enhancements. You should always ensure that your data-centric applications and query and reporting tools use SQL when interfacing with DB2 for i. If you're not using SQE, you're not getting the most out of DB2 for i, and you're not getting the greatest BI performance.

Balanced Configuration — Optimal Set of Resources

When you're setting up a system for business intelligence, be sure to configure the proper set of computing and storage resources. You must balance this set of resources in terms of the relationship between CPUs, memory, and I/O subsystem. The fastest part of the system is the CPU, and all CPUs wait at the same speed. You should always provide a properly sized and configured system for performance. Depending on the actual query usage and response time expectations, a properly balanced configuration consists of dedicated POWER6 processors, 6 GB to 12 GB of memory per CPU, and 10 to 20 disk units per CPU. These are minimums based on general observations and experience. Of course, your requirements and configuration might be different. Proper analysis, planning, and testing are crucial success factors. Always avoid a lopsided configuration in which you have plenty of CPU power but not enough I/O subsystem to support it.

If you use DB2 SMP to accomplish database parallelism, multiple CPUs must be available to support the execution of multiple concurrent threads or tasks. Furthermore, it is generally best to set up a dedicated system or logical partition (LPAR) for the BI environment. Trying to support both online transaction processing (OLTP) and your BI workload on the same system is problematic and can be difficult to tune effectively. A separate BI system or LPAR provides a greater return on investment and lets you meet the differing and sometimes diverging requirements of OLTP and query/reporting.

Given that IBM's Power servers and supporting operating systems offer some of the best virtualization features available, you should consider using a logical partition for the data warehousing or reporting infrastructure. By having the data warehouse LPAR next to the transaction LPAR(s), you can use private high-speed LANs to facilitate the data transportation process. The system can dynamically move and share CPU and memory resources as business objectives ebb and flow throughout the day or week. Furthermore, if the need arises to run BI solutions or middleware in AIX or Linux, you can configure a suitable LPAR next to IBM i or i5/OS LPAR.

Work Management

DB2 for i work management and nondatabase workloads have some similarities and differences. It's the same in that the IBM i operating system manages database work just like any other work. Unlike other database management systems, DB2 doesn't need you to set up and configure separate subsystems, memory areas, and buffer pools. Jobs or threads doing DB2 work are initiated and managed by OS subsystems, and they have a run priority and a time slice. Given that the query optimizer's job is to identify and assemble the best (and fastest) plan, what stops it from building a plan that consumes all the computing and I/O resources? In other words, we need to understand what's different about work management when it comes to query optimization.

Two of the most important items to consider are the parallel degree setting for the job running the query, and the job's fair share of memory that the query optimizer can consider. The SMP or parallel degree values of *NONE, *IO, and *OPTIMIZE (aka ANY) tell the optimizer to calculate and adhere to the job's fair share of memory. The *MAX setting lets the query optimizer consider and use all the memory in the pool in which the job is running. The fair share of memory is roughly calculated as memory pool size divided by the maximum active value for the pool. Thus, depending on the pool size and the max active value, the query optimizer can build an aggressive plan (i.e., more fair share) or an anemic plan (i.e., less fair share). The query optimizer should allocate a fair share of memory per job to ensure maximization of the number of options available for running the query.

Fair allocation has the added benefit of letting the query plans be more CPU bound, consequently minimizing or eliminating waiting for I/O. The keys to this practice are proper capacity and performance planning as well as setting the max active for the pool to a value indicative of how many active jobs are executing queries concurrently. In other words, how many jobs do you expect to be truly running query plans versus just sitting idle? You should also carefully consider the behavior of the auto performance adjuster. Many systems use this feature, which is controlled by the system value QPFRADJ. Changes that the performance adjuster makes to the pool size and max active setting affect the fair share that the optimizer calculates. If your job's fair share decreases, the query plans might become less aggressive and ultimately slower. Lower overall CPU utilization and less database throughput are indications of this phenomenon.

PWRDWNSYS — to IPL or Not

With SQL queries, you tell DB2 what to do but not how to do it. In effect, the query optimizer is "a little programmer in a box" who writes the logic to fulfill your request. This logic (or query plan) is saved for future use, which eliminates the need for repeating work when the same query is issued. These query plans are saved in temporary storage and don't persist across IPL of the system or LPAR.

The DB2 for i query optimizer and database engine use temporary data structures when no permanent objects, such as indexes, exist or, more important, to let the query become CPU bound and go faster. With SQE, these temporary data structures can be reused and even shared across queries and jobs. As SQE executes queries, learning and adaptation are occurring. One byproduct of this behavior is the creation, saving, and reusing of data structures — and even the ability to save a query's final result and reuse it for the same query run against the same underlying data. In some respects, this functioning represents autonomic self-tuning. The temporary data structures don't persist across IPL of the system or LPAR.

From a database perspective, best practice is to avoid frequent IPLs of the system or LPAR. By powering down less often, you can take advantage of the self-tuning support and minimize the "Monday-morning slowdown" of queries. One suggestion is to consider monthly or quarterly scheduled IPLs and use the opportunity to apply PTFs and make any hardware adjustments.

Monitoring and Tuning Facilities

With V5R4, you have new and improved SQL monitoring and tuning facilities. With V6R1, you have even more capabilities available to help keep an eye on the queries. These capabilities are primarily delivered through Navigator for i5/OS client support and DB2 for i server support. Additionally, value-added capabilities are available through ISVs.

You can continually monitor and appropriately tune queries with features such as the SQL Plan Cache, snapshots, and monitors. The plan cache represents a repository for all SQE plans and contains a lot of useful information about the query plans and query runtime behavior, as well as advice for creating indexes. Although the plan cache is live and volatile, the SQL Plan Cache Snapshot feature resolves the query plan and query runtime information and stores it in a persistent database table, giving the database-performance analyst the opportunity to collect, store, and compare query plans from different points in time. Unlike the SQL Plan Cache, the SQL Performance Monitor (or database monitor) provides the ability to trace all the SQL statements for a job or set of jobs. The information collected is stored in a persistent database table, and you can analyze it by using the built-in Navigator for i5/OS reporting, or you can query directly to produce custom reports. The SQL Performance Monitor has many filtering options, letting you focus on specific queries, users, jobs, and database clients.

Another powerful feature of the DB2 for i cost-based optimizer is the predictive query governors. Both a time-limit governor and a temporary-storage-limit governor are available. During optimization, the query's estimated runtime and estimated amount of temporary storage are calculated. Before running the query, the system compares these values with the user's threshold. If the query's estimated runtime or temporary storage exceeds the threshold, the system can either end the query or ignore the limit and let the query run, depending on how you've configured the values. In addition, for any query that exceeds the threshold, the system can capture detailed query optimization information in an SQL performance-monitor table.

To reduce the number of long-running queries, you must determine and set appropriate time limits and temporary storage limits. You can pursue more-detailed investigation and tuning by using the detailed monitor data captured for any out-of-bounds SQL requests. Remember, the optimizer's runtime and temporary storage figures represent only an estimate and might not reflect actual query execution behavior. The query optimizer's estimates primarily exist to help you compare and choose the best methods and strategies. The better the statistics (from permanent indexes and column statistics), the more accurate the estimates are.

Test, Tune, Go Live

Query optimization is based on many factors, all of which come together to form a plan. A given query plan is directly dependent on the SQL request, the underlying data, and the computing environment. You should test and tune your SQL-based applications in the same environment in which they're running in production and with the same data. Also, test, monitor, and tune the ETL and end-user queries before going live. As part of building a successful quality assurance and performance test plan, you need to fully understand the query response time expectations, the number of concurrent users, and the data

growth. Teaming up with knowledgeable power users from the business can go a long way toward ensuring a successful BI implementation.

Keep a Watchful Eye

When you deliver a BI solution to the business, you should continue the monitoring process. This process can include some data profiling. Data profiling is the collection, analysis, and understanding of the data in the data warehouse and data marts. For example, how large are the tables? How fast are they growing? What is the cardinality of column values? What is the distribution of a given set of column values? Understanding your data will help you understand and thereby properly configure DB2's behavior. By keeping a watchful eye on database use and user behavior, you can anticipate and resolve problems more quickly and with less stress and frustration. Another aspect of monitoring is that it lets you ascertain new or changing requirements so that you can plan for future enhancements to the query and reporting environment. For example, you might discover that you need to add more fact tables and dimensions to support new subject areas. Or you might notice organizational changes that require you to change dimensions in the proper way. Remember, a BI solution is valuable only if it's used, and used efficiently.

Broaden Your Horizons

With DB2 for i 6.1, today's powerful IBM servers, and the best practices I describe here, great BI performance is at your fingertips. More information about many of these topics is available on the DB2 for i website (ibm.com/systems/i/software/db2). Click the Getting Started tab, and then click the links to White Papers or Online Presentations. If you want to delve into these topics and become proficient at the science and art of DB2 for i, consider enrolling in one of our SQL Performance Workshops. More information about the venues is available at ibm.com/systemi/db2/db2performance.html.