IBM DB2 for i table partitioning strategies

Version 3.0

Mike Cain
IBM DB2 for i Center of Excellence

June 2014

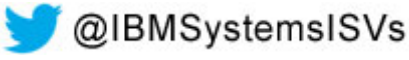


Table of contents

Abstract

This paper discusses IBM DB2 for i support for local table partitioning. This paper explains when this technology should be used and what design points should be considered before implementing partitioned tables. Both, programmers and database administrators can find information on how to deliver a successful local table partitioning implementation.

Introduction

On any platform, good database performance depends on good design. And good design includes a solid understanding of the underlying operating system and database technology, as well as the proper application of specific strategies.

This is also true for IBM® DB2® for i, which provides a robust set of technologies that assist with query optimization and performance.

Prerequisites

It is strongly recommended that database administrators, engineers and developers who are new to the IBM i platform, or new to SQL, attend the *DB2 for i SQL and Query Performance Monitoring, Analysis and Tuning* workshop. Among other things, this workshop teaches the engineer or developer the proper way to architect and implement a high-performing DB2 for i solution. For more information about delivery of this workshop, contact the author.

The points discussed in this paper assume some knowledge of DB2 for i. It is helpful to refer to, and familiarize yourself with the information contained in the IBM i Knowledge Center (at **ibm.com**/support/knowledgecenter/ssw_ibm_i/welcome) and the following publications.

Publications

- DB2 for i Multisystem
- DB2 for i SQL Reference
- DB2 for i Database Performance and Query Optimization

Papers

- Indexing and Statistics Strategies for DB2 for i

You can find the links to these and other database publications and papers through the DB2 for i website at: **ibm.com**/systems/power/software/i/db2/gettingstarted.html.

Partitioning and partition tables

Partitioning allows for data to be stored using more than one physical data space, but the table appears as one object for data-manipulation operations, such as read, insert, update, and delete. This form of local

table partitioning stores much more data in the tables, with the table physically residing on one database server.

In general, table partitioning is used as a design element within three general areas:

- Query performance and parallelism
- Bulk data operations such as save, restore, add, drop, and reorganize
- Overcoming limits to growth

Though there are some limited circumstances where query performance can be improved, the primary uses of local table partitioning are to overcome the size limitations of an individual SQL table within DB2 for i and some bulk data operations.

Without partitioning, there is a limit on the number of rows in a given table and a limit on the overall size of the table object. The limits are approximately 4.2 billion rows in a table, or a table size of 1.7 terabytes. To understand how large these limits are, the following table provides a list of various row lengths and the approximate number of rows needed to reach the 1.7 TB limit.

Row length	**Number of rows**
32766	57 million
8192	228 million
2048	912 million
1024	1.8 billion
512	3.6 billion
435	4.2 billion

Table 1: Various row lengths and number of rows to reach a 1.7 TB limit

Note: You can find all of the DB2 for i limits in Appendix A of the DB2 for i SQL Reference (**ibm.com**/systems/power/software/i/db2/docs/books.html).

Partitions are different sets of data, each with the same format, within one database table (refer to Figure 1). The physical rows of each partition are distributed automatically across all the available disk units through IBM i storage management.

A partitioned table can have up to 256 partitions, with each partition having the ability to grow to the respective maximum limit. Fully populating 256 partitions can result in a table with 1 trillion rows and a size of 435 TB.

In the world of relational database design, table partitioning is often implemented to gain the ability to use parallel methods and strategies. Unlike many other relational database management systems, DB2 for i does not require partitioned tables to fully use parallel processing. With the optional DB2 Symmetrical Multiprocessing (SMP) feature, DB2 for i can use parallel methods and strategies to run queries, whether the table is partitioned or not.

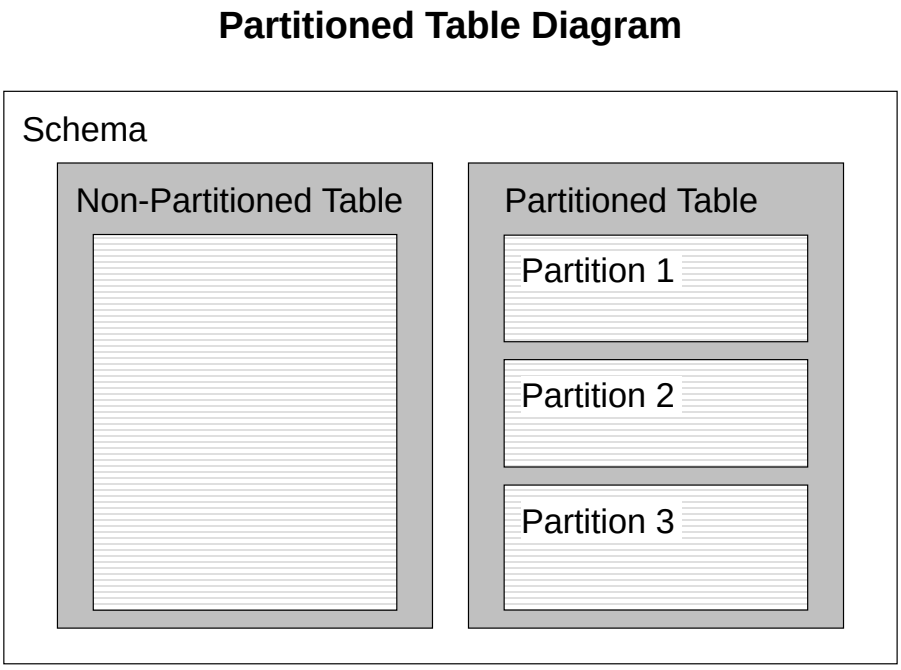


Figure 1: Partitioned table diagram

DB2 for i allows for hash partitioning or range partitioning, based on a partition key. System-generated check constraints are used to ensure that only the rows with the appropriate partition key value are inserted into the respective partition. The check constraints are also used to assist the query engine with data access.

With hash partitioning, a hashing algorithm is applied against the partition key value to determine the partition into which the row is placed (refer to the sample SQL code listing that follows and notice its correlation to **Error! Reference source not found.**).

```
CREATE TABLE MY_PARTITIONED_TABLE (
  EMPNUM INT,

  FIRSTNAME CHAR(15),

  LASTNAME CHAR(15),
  SALARY INT)

PARTITION BY HASH(EMPNUM) INTO 3 PARTITIONS;
```

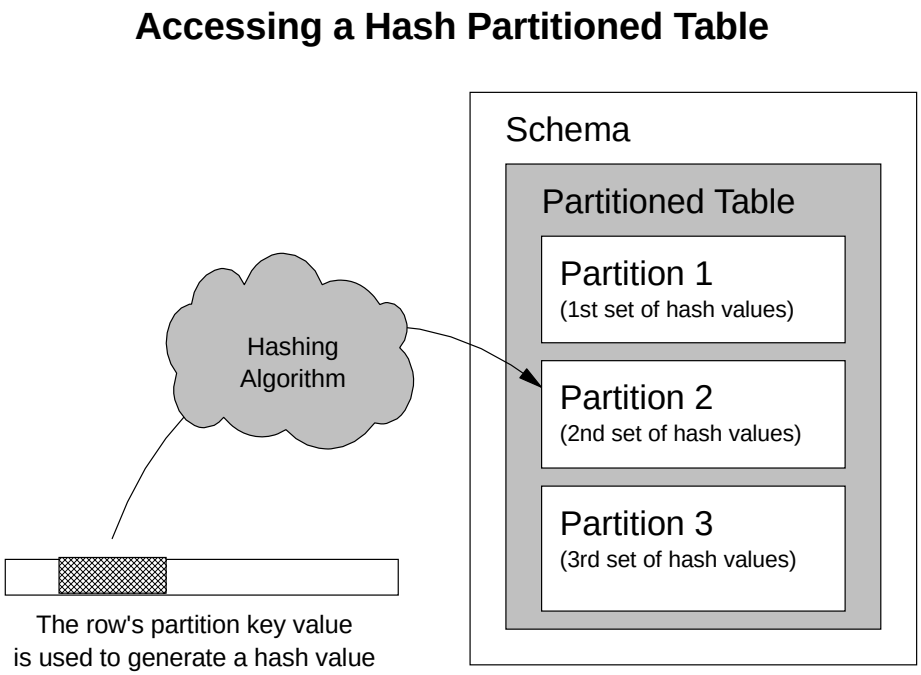


Figure 2: Accessing a hash-partitioned table

With range partitioning, the row is placed in the appropriate partition based on the partition key value as compared to the partition's defined range of values (refer to the sample SQL code listing that follows and notice its correlation to Figure 3). Be sure that the partition key ranges specified on the CREATE or ALTER statement cover the ranges expected in the data.

```
CREATE TABLE MY_PARTITIONED_TABLE (
EMPNUM INT,

FIRSTNAME CHAR(15),

LASTNAME CHAR(15),

 SALARY INT)

PARTITION BY RANGE(EMPNUM) (
STARTING FROM (MINVALUE) ENDING AT (500) INCLUSIVE,
STARTING FROM (501) ENDING AT (1000) INCLUSIVE,
STARTING FROM (1001) ENDING AT (MAXVALUE));
```

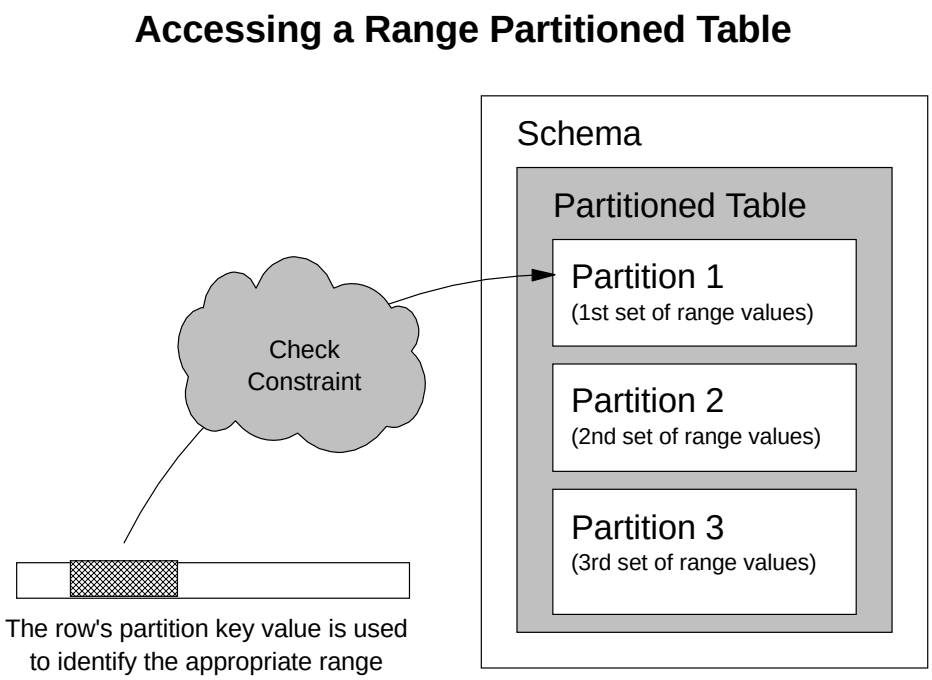


Figure 3: Accessing a range-partitioned table

Indexes on partition tables

Partitioned tables can have both radix indexes and encoded vector indexes. Radix indexes can be created as partitioned or nonpartitioned. Creating a partitioned radix index establishes an individual radix index structure for each partition in the table. Each partition of the index only contains keys for the corresponding table partition. Creating a nonpartitioned radix index creates a single radix index structure that contains keys for all the partitions. In other words, this index spans all partitions of the table. Thus, a nonpartitioned index is also known as a spanning index. Encoded vector indexes can only be created as partitioned indexes (refer to the sample SQL code listing that follows and notice its correlation to Figure 4).

```
CREATE INDEX MY_P_IX
ON MY_PARTITIONED_TABLE
(EMPNUM)
PARTITIONED;

CREATE INDEX MY_NP_IX
ON MY_PARTITIONED_TABLE
(EMPNUM)
NOT PARTITIONED;
CREATE ENCODED VECTOR INDEX MY_P_EVI
ON MY_PARTITIONED_TABLE
(EMPNUM);
```

Partitioned and Non-Partitioned Index Diagram

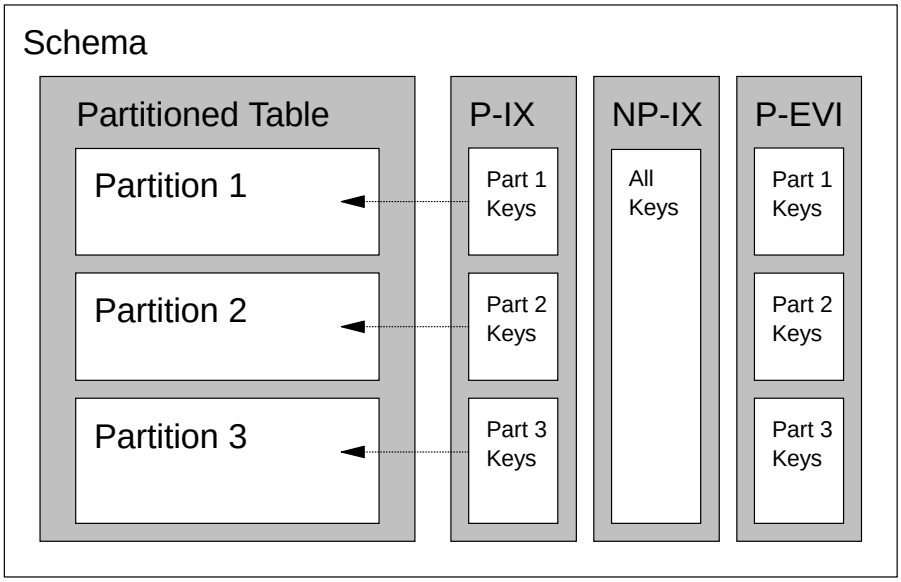


Figure 4: Partitioned and non partitioned index diagram

When creating a unique SQL index or a unique primary key constraint (which often implicitly creates a unique index) for a partitioned table, the following restrictions apply:

- An index can be partitioned if the keys of the unique index are the same or a superset of the partitioned keys.
- If a unique index is created with the default value of NOT PARTITIONED, and the keys of the unique index are a superset of the partitioned keys, the unique index is created as partitioned. If, however, the user explicitly specifies NOT PARTITIONED and the keys of the unique index are a superset of the partitioned keys, the unique index is created as a nonpartitioned index.

Given that a nonpartitioned index covers all the partitions with a single index structure, in general, adding or dropping a partition requires the re-creation of any nonpartitioned indexes. This includes any indexes, supporting primary key or unique constraints. It is possible to add a new partition that occurs at the end of the range without the spanning index being re-created. It is possible to drop an empty partition that occurs at the end of the range without the spanning index being re-created.

Re-creating indexes can be very time-consuming and resource-intensive, possibly negating any operational benefits of partitioning a table, and decreasing the availability of the index and any applications that rely on it.

Given that partitioned indexes cover each respective partition with a separate index structure, adding or dropping a partition does not affect the other partitions or their respective indexes. When adding a new partition, new index structures are created for just that partition. When dropping a partition, only the index structure over that specific partition is dropped, as well.

Requirements and limitations

Table partitioning requires the optional DB2 Multisystem licensed feature of IBM to be purchased and installed. In addition to local table partitioning, DB2 Multisystem provides the capability to create a table that is partitioned and physically distributed across multiple IBM i systems or *nodes*. Though the scope of this paper only covers local table partitioning, note that designing and implementing a data model with distributed partitioned tables is not recommended. Furthermore, implementing distributed tables has significant limitations that can prohibit the effective use of this technology. Generally speaking, the function of creating and querying a distributed table in a loosely coupled environment is deprecated.

With the DB2 SMP feature installed and enabled, queries against a partitioned table can be run using parallel-enabled methods. Currently, there is no interpartition parallelism. Only the piece of query targeting an individual partition can run in parallel. In other words, if a table has three partitions, and the query implements a table scan against each of the three partitions, the individual scan of partition number one can run in parallel, but the scans of partitions two and three wait for the scan of partition one to complete. With interpartition parallelism, scanning of all three partitions might occur simultaneously or in parallel.

Before IBM i 7.1, partitioned tables do not allow referential integrity constraints and a table with an identity column cannot be partitioned. Thus, if the data model or an application running in IBM i 6.1 requires referential integrity constraints or identity columns, then table partitioning is not available for the environment. It is recommended to upgrade IBM i to 7.1 or 7.2.

Before IBM i 7.2, the partition key cannot be changed with a row update operation. To change the partitioning key, the row must be deleted and inserted into the new partition.

You can find a list of all the restrictions in the respective DB2 Multisystem and SQL Reference manuals in the IBM i Knowledge Center.

Query optimizer considerations

Optimizing and running a query against a partitioned table is similar to optimizing and running a UNION ALL query; where each leg of the UNION ALL request represents a separate query against a particular partition.

The query:

```
SELECT *
FROM MY_PARTITIONED_TABLE
WHERE EMPNUM = 500;
```

Is roughly equivalent to:

```
SELECT *
FROM PARTITION_1
WHERE EMPNUM = 500
  UNION ALL
SELECT *
FROM PARTITION_2
WHERE EMPNUM = 500
  UNION ALL
SELECT *
FROM PARTITION_3
WHERE EMPNUM = 500;
```

If the query specifies the partition key as a local selection predicate, then the optimizer might be able to take this into consideration when choosing and building the access plan.

Another important optimization concept is the materialization of the partitioned data sets. In some cases, the individual partitions must be combined (materialized) into one data set to allow for the joining and grouping of rows. Although the optimizer attempts to push down as much local selection as possible in an effort to minimize the size of the temporary data set, this materialization can result in more I/O operations, more consumption of temporary storage, and longer query-execution time.

Given the following SQL request:

```
SELECT MAX(SALARY)
FROM MY_PARTITIONED_TABLE
WHERE LASTNAME LIKE ‘SMIT%’;
```

The query optimizer combines the rows from all three partitions into one temporary data set before implementing the MAX function. The local selection is applied (that is, pushed down) as each of the partitions are accessed and materialized.

With IBM OS/400® V5R2, a reengineered SQL Query Engine (SQE) was introduced. The original query engine is referred to as Classic Query Engine (CQE). While both SQE and CQE can optimize and run queries against partitioned tables, SQE is far better equipped to handle these types of queries. To understand the differences between the two query engines, you need to compare and contrast how each engine can optimize and run a query against a partitioned table, or, if required, a UNION ALL query.

The major differences between SQE and CQE are:

- SQE does not require the materialization of the partitioned data.
- SQE has many more methods and strategies available for optimizing queries.

CQE queries that include joining, grouping, or aggregate functions require the materialization of the partitioned data into a temporary table. If the number of rows materialized or the temporary table size exceeds the limits for a single, non-partitioned table, the query will fail. The CQE optimizer attempts to minimize the amount of data in the temporary table by performing as much local selection on each partition as possible.

Refer to “Appendix A: SQL query engine (SQE) restrictions” for a list of the query attributes and interfaces that require the use of CQE.

When CQE is used to run the query, additional restrictions are also in place. The following methods and strategies are not available:

- Use of encoded vector indexes for the creation of relative record number (RRN) lists and index ANDing and ORing
- Look ahead predicate generation (LPG)
- Use of a partitioned index (CQE can only optimize and use a nonpartitioned index, and the lack of a nonpartitioned index can result in the creation of a temporary, nonpartitioned index)

In addition to the behavior just described, if the queries against partitioned tables are optimized and run with CQE, there might be a significant and an unacceptable increase in the query’s temporary storage usage, memory footprint, and paging rates.

Whether CQE or SQE is used, there are longer overall optimization times and larger query plans, as compared to querying a nonpartitioned version of the table. To minimize the optimization time, the optimizer builds a best plan for the first partition and replicates this plan for the other partitions. For plans that make use of an index for accessing the partitions, more index pages are processed as compared to querying the same nonpartitioned table.

Without analyzing the query environment or running some benchmarks, predicting any increases in resource usage and response times is difficult.

Though the implementation of partitioning allows for up to 256 partitions for a given table, it is recommended that you choose the number of partitions wisely and judiciously.

When handling a query request, the database engine must generally go through three phases:

- Optimizing the query
- Opening of the data path (also known as: the ODP or cursor)
- Running the query plan

The larger the numbers of partitions, the higher the probability of increasing the query optimization time, open time, and possibly, the execution time. This phenomenon increases the amount of temporary storage consumed and increases the size of the query plans.

A general rule of thumb is to keep the number of partitions at or below 36. When considering more than 36 partitions, plan on performing additional analysis, testing, and benchmarking of the queries, applications and operational procedures.

Allocating more memory to the memory pools used for query workloads is likely required to avoid a significant increase in paging as the result of the larger memory footprint required for table partitioning.

Keep in mind that if the query performance of a partitioned table does not meet expectations, it might be necessary to migrate back to a nonpartitioned table. Once again, proper testing at scale is highly recommended.

Indexing and statistics strategies

The indexing and statistics strategies for partitioned tables are essentially the same as for nonpartitioned tables. That is to say, a proper indexing strategy is essential for effective query optimization and good query performance. Without indexes, the query optimizer might lack the necessary statistics and be limited in the choices it can make for the query request.

As mentioned previously, one important consideration is the fact that there are essentially two types of radix indexes for partitioned tables: partitioned indexes and nonpartitioned indexes. Generally, partitioned indexes are created and are most useful because they address a specific partition and allow for fast operations, such as adding and dropping a partition without requiring the index to be rebuilt.

If it is known or suspected that CQE or programs using native record-level access will be used to access the partitioned table, and nonpartitioned indexes must be created for local selection, joining, grouping, and ordering. Otherwise, CQE uses full-table scans or creates temporary indexes, both of which cause poor query performance. Because the nonpartitioned indexes span all the partitions in the table, adding and dropping partitions generally requires this type of index to be rebuilt. This rebuild activity uses a significant amount of resources and results in poor query performance while the index is unavailable.

Encoded vector indexes are always considered as partitioned indexes. Given that CQE can only use nonpartitioned indexes when accessing a partitioned table, creating nonpartitioned radix indexes in addition to any encoded vector index is advantageous when CQE is used. That is to say, ensure that a nonpartitioned radix index is created over any columns indexed with EVIs.

To support both SQE and CQE query processing, as well as any programs using native record-level access, you essentially need to have both the partitioned and nonpartitioned indexes created and maintained. This slows down then performance of insert, update, and delete operations; this extra work needs to be factored when considering partitioned tables.

Column statistics used by SQE are collected and stored on a partition-by-partition basis. This means that the statistical information for a column is based on the data in the respective partition, not the entire table. Given that the SQE optimizer might build a different plan for each partition, it is important to provide column statistics not only for one partition, but for all the partitions represented in the table. If SQE automatically collects a column statistic for one partition, then it is a good practice to identify those columns periodically and ensure that each partition in the table has the same statistics collected.

When and where to use partitioned tables

After careful consideration of the requirements, limitations, and specific query engine behavior; a Database Engineer can arrive at a set of environments and situations where using partitioned tables might be necessary or advantageous.

As mentioned previously, the primary use for partitioned tables is in cases where the single table limits are expected to be reached and it is not practical (or desirable) to redesign the data model, the application, or the data access technique. Partitioned tables can also be used when more efficient bulk-data operations are needed.

In a few situations, partitioned tables can be used to improve query performance. If practical and possible, creating partitions that allow all of the SQL operations to be scoped to a particular partition provides the best performance because it avoids processing multiple partitions. This can be accomplished by specifying the partition-key column as a local-selection predicate. Another example involves designing the data model and the application such that all of the database operations are explicitly directed to a specific partition with no reliance on the database engine's partition awareness. More information on how to access a partition directly is covered later in this paper.

Although the use of a partitioned table can be used in any environment to overcome limits to growth, the manner in which the table is partitioned depends on the specific environment and the type and frequency of activity against the table.

When investigating the use of table partitioning for situations other than overcoming the DB2 for i table limits, it is important to consider the intersection of all the various aspects and limitations of this new feature. If the primary interest involves the potential operational benefits, the programmer must also weigh the risk of queries that run slower and / or require higher resource utilization.

Examples in OLTP and batch-processing environments

A simple approach to overcoming the table limits in the online transaction processing (OLTP) and batch processing environments is to use two, three, or four partitions that are maintained by hash partitioning. This effectively allows for doubling, tripling, or quadrupling of the single table limits while incurring a minimum amount of performance degradation. Prior to the IBM i 7.2 release, you must choose the partition key wisely, because it is not possible to update a row's partition key value if the update results in moving the row from one partition to another. To change a partition key value, the programmer can delete the row and insert a row with the new value. A better approach to overcome the table limits might be to consider partitioning the very large table by range, where choosing the partition key column allows some natural sub-setting of data. This might be a business entity such as time periods, customers, orders, or locations. The ability to identify and access a specific subset of rows physically within a larger data set might allow some new and more efficient operations.

When an OLTP application primarily accesses a known and consistent subset of data, such as the current fiscal period's orders, there might be an opportunity to gain some benefit from partitioning the data by range. In this example, if the data is partitioned by fiscal period, only the active or current period's data must be saved. Saving only a subset of the data speeds up the operation and shortens the window of time

required for exclusive access to the database objects. If the older data sets are no longer required to be available online, they can be easily archived. In this scenario, it is important for the SQL requests to provide the fiscal period value as a local selection predicate. This allows the database engine to focus the data access to a specific partition.

When processing a batch of data on a periodic basis, perhaps nightly, table partitioning that uses ranges might also help isolate the data processing to a smaller set of rows. This can be advantageous when the new batch of data is added to an existing set of data, and only the new data needs to be accessed, perhaps with a table scan. Instead of scanning the entire historical data set, only the new partition is scanned and processed.

The periodic deletion of a batch of data is also faster. Instead of issuing delete operations for individual rows, one operation can be used to delete an entire range of data (by dropping the partition). This can have benefits in a high availability (HA) environment where the data and the atomic operations are mirrored to another system. Instead of the HA system applying each row-deletion operation, one operation (drop partition) flows to the HA system and runs.

If the query and data processing activities within the batch environment requires nonpartitioned indexes, the use of table partitions for fast adds and drops is prevented by the required re-creation of the nonpartitioned indexes.

Furthermore, the job that performs the partition operation (that is, add or drop) must obtain a lock on the entire table. This requirement restricts the level of concurrent access available and might result in an unacceptable slowdown in throughput.

Examples within a data warehouse, operational data store, and analytical environment

For business intelligence (BI) environments with time-period-based requests or operations, range partitioning might provide additional benefits. For example, if the data warehouse or operational data store (ODS) is required to contain three years of data, range partitioning that is based on a partition key of year not only allows a larger table size or more rows, but can also provide the basis for partition-based operations. These partition-based operations might save only the current year's data (thereby, dropping a year's worth of obsolete data or adding a new year's worth of data). In addition, queries that specify a particular year (with local selection against a partition key of year) might see some performance benefits, depending on the access methods available to the optimizer.

Another example assumes that the grain of the data is based on time periods, with a requirement to store three years of data. In this case, partitioning the table using a partition of year + month, results in using 36 partitions (3 years x 12 months).

Partitioning by year + month allows for a simpler, faster deletion of the old data set and quicker additions of the new data set (drop a partition, add a partition). Archiving or saving the data can also be accomplished on a month-by-month, partition-by-partition basis, with a reduction in time and resources.

If the queries can specify a given month and use SQE, then the query can be scoped to a given partition to provide query performance with a minimum of degradation. For queries that do not use the partition key as a local selection predicate, there is an increase in response time (as compared to querying the nonpartitioned table). This is because each of the 36 partitions must be queried.

In a BI environment, it is common to have large fact tables as part of a star schema or snowflake schema data model. It is also common to have the data in the fact table represented along some time dimension (such as day, week, month, or year). It is compelling to assume that partitioning this fact table based on ranges of time (week or month) is a positive thing for query performance and operational efficiency. If the multidimensional queries against the partitioned fact table contain the partition key (a specific week, month, or year) as a local selection predicate, then partitioning can yield performance benefits. But, if the queries do not contain the partition key, the queries might run significantly longer and use more resources than without table partitioning. This phenomenon is the result of identifying the candidate rows in the fact table through joining the dimensions, not necessarily through local selection directly on the fact table. Without partitions, one set of rows is identified by the query for processing. When the fact table is partitioned, multiple sets of rows are identified for processing. In other words, querying each partition might yield a set of rows to be processed. Furthermore, if the query is processed by CQE instead of SQE, then advanced optimization techniques cannot be used and the query can run for an extremely long time. Partitioned tables are not recommended in an environment where a large number of queries must use CQE.

Directly accessing partitions

By default, individual partitions are transparent to the SQL request, but with an SQL ALIAS, the programmer can explicitly reference a specific partition. By creating an SQL ALIAS, a persistent pointer to a partition is created. When the SQL ALIAS is in place, the SQL operation can be directed to a specific partition, localizing the operation to that partition only. This avoids the querying of additional partitions. Though this technique does remove some of the transparency of partitioned tables, it can be an effective way to take control of the operations and possibly to increase the speed and efficiency of an SQL request (refer to the sample SQL code listing that follows and notice its correlation to Figure 5).

```
CREATE ALIAS Alias_1
FOR MY_PARTITIONED_TABLE (Partition_1);
CREATE ALIAS Alias_2
FOR MY_PARTITIONED_TABLE (Partition_2);
CREATE ALIAS Alias_3
FOR MY_PARTITIONED_TABLE (Partition_3);
```

Partitioned Table with Aliases Defined

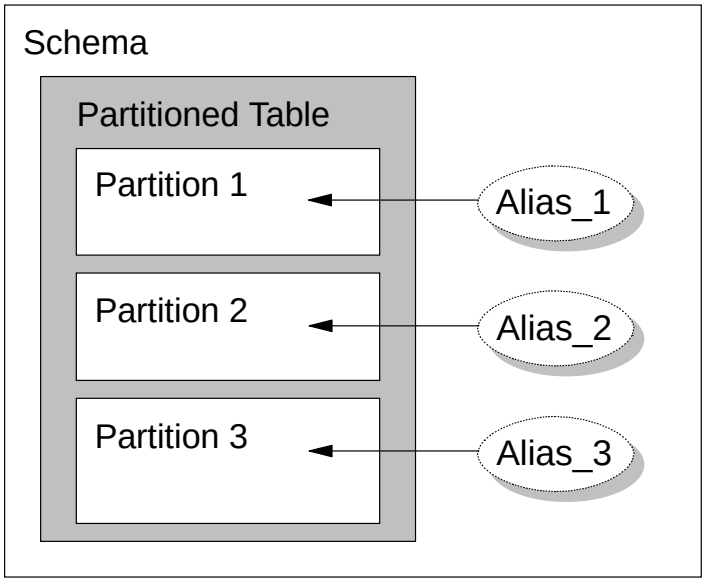


Figure 5: Partitioned table with aliases defined

The following SQL statements are examples of how to specify and use the SQL ALIAS in an SQL statement:

```
SELECT MAX(SALARY) FROM Alias_1;
UPDATE Alias_3 SET SALARY = 0;
```

Accessing partitions with the native non-SQL interfaces

Although the primary interface to partition tables is SQL, it is possible for programs to use the native record-level access to process the individual partitions within a partitioned table. Given that the partitions are implemented as physical file members, the IBM i Command Language (CL), RPG, or COBOL program can open an individual member for data processing through the IBM i Override Database File (OVRDBF) command. Furthermore, if a nonpartitioned index is created over the partitioned table, the underlying multi-member logical file can be used to access the partitioned data without needing to know about the partitioning scheme. Be aware that each physical file member has its own set of relative record numbering. RRN access must be done on a member-by-member basis.

Using native record-level access for inserting and updating rows within a partition is subject to all the same restrictions as SQL, such as the check constraints used to ensure that the rows are placed in the appropriate partition.

Using IBM i commands to operate on the individual partitions is permissible just as with other multi-member physical files, provided that the appropriate member parameters are specified. Remember, most native database operations default to using the first member in a physical file or partitioned table unless the access is through a logical file or SQL view.

Here is an example of setting up access to a particular partition:

```
OVRDBF FILE(table's-short-name) TOFILE(*FILE) MBR(partition-name)
CALL MY_PGM
```

The program now opens the partition specified in the override database file command (refer to Figure 6).

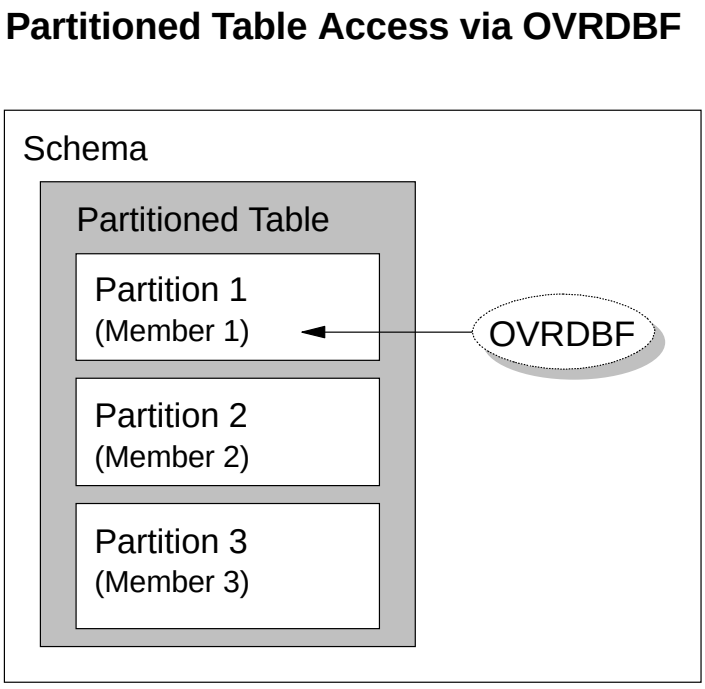


Figure 6: Partitioned table access with OVRDBF

Migration strategies

The best time to consider partitioning a table is during the design and initial construction of the data model. But in many situations, the requirement to partition a table comes after the nonpartitioned table is created and populated. In this case, the table has to be altered or re-created as a partitioned table, and the data must be migrated to the new structure. Though it is simple and straightforward to use the ALTER TABLE support provided with DB2 for i, a more effective, efficient, and faster approach can be employed.

During the ALTER TABLE operation, a new partitioned table is created and the data is copied from the nonpartitioned table to the partitioned table. Any indexes are re-created as partitioned indexes. In the copy process, the column data that corresponds to the partition key is used to determine which partition receives the respective row. This process can be time-consuming. Furthermore, the copy process does not run in parallel. This lack of parallelism can be particularly troublesome if the original data set is large and the migration to the partitioned table must complete in as little time as possible. Instead of using the ALTER TABLE exclusively, it can be advantageous to design and build a programmatic process that makes use of parallel processes.

The programmer can operate on each partition directly with native record-level write or SQL insert operations. If using SQL, the preferred approach is to create an SQL alias for each partition and then reference the alias on the *insert* statement. By inserting directly into each partition, the programmer takes the responsibility for ensuring that the row is placed into the appropriate partition. Remember that the system-generated check constraints only allow the insertion of rows that match the partitioning key criteria.

To employ parallelism, consider running concurrent SELECT and INSERT statements for each partition that is expected to be populated. Using the previous example of range partitioning with 36 months, theoretically 36 operations might run concurrently (in parallel) to migrate data from the nonpartitioned table to the partitioned table. This high degree of parallelism consumes more resources to reduce the data migration time.

For the partition and range represented by Alias_1:

```
INSERT INTO Alias_1
  SELECT *
  FROM MY_NON_PARTITIONED_TABLE
  WHERE YEAR = 2004
  AND MONTH = 1;
```

For the partition and range represented by Alias_2:

```
INSERT INTO Alias_2
  SELECT *
  FROM MY_NON_PARTITIONED_TABLE
  WHERE YEAR = 2004
  AND MONTH = 2;
```

For the partition and range represented by Alias_3:

```
INSERT INTO Alias_3
  SELECT *
  FROM MY_NON_PARTITIONED_TABLE
  WHERE YEAR = 2004
  AND MONTH = 3;
```

For hash partitioning, the specific partition (number) where the row is to be inserted, is determined based on the same algorithm that the database engine uses. The algorithm can be represented as follows:

Partition number = MOD(Hash(partitioning columns), number of partitions) + 1

Where, the hash function returns a value between 0 and 1023. Any null values can be placed in the first partition or the last partition.

One idea when using hash partitioning is to identify a column that has cardinality equal to or near the number of partitions. For example, if three partitions are defined and three distinct years of data are represented, predetermine the partition number by using the algorithm with each year value; then select rows from the nonpartitioned table based on one year, inserting these rows into the appropriate partition through the respective alias. Again, running concurrent, parallel processes increases the I/O velocity and decreases the migration time (refer to Figure 7).

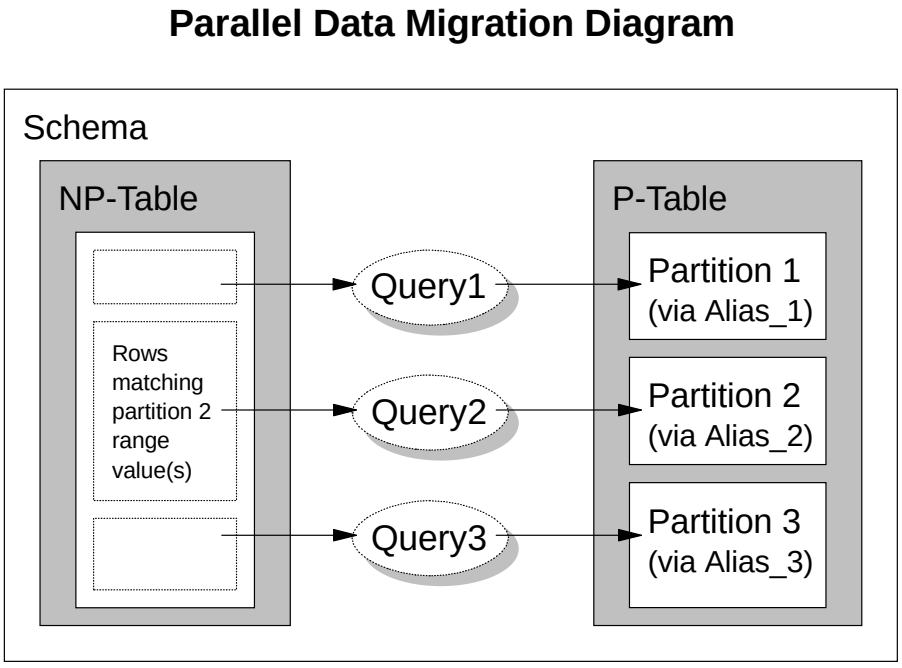


Figure 7: Parallel data migration diagram

General list and sequence of migration events

Following is a suggested list and sequence of activities to migrate nonpartitioned tables to partitioned tables:

- Ensure that there is enough storage to contain the nonpartitioned and partitioned table.
- Do a full, dedicated save of the nonpartitioned table and indexes.
- Verify the saved data.
- Create the new partitioned table with the appropriate ranges and partition key.
- To help verify the migration results, count the rows to be placed in each partition by running queries on the ranges within the nonpartitioned table – noting the cardinality of the partition column is also advantageous.
- Create an SQL ALIAS for each partition.
- Use the OVRDBF procedure to increase the blocking level to 256 KB for both the source and target tables.

```
CALL QSYS2.OVERRIDE_TABLE ('schema', 'source table', '*BUF256KB');
CALL QSYS2.OVERRIDE_TABLE ('schema', 'target table', '*BUF256KB');
```

- Migrate the data in parallel by running concurrent SQL INSERT statements with subselects, where:
  - The subselects choose rows from the nonpartitioned table based on the range that is defined for the target partition
  - The INSERT statement inserts into the specific target partition through the respective alias.
- Verify that all the data has been migrated to the partitioned table. (Count the rows in each partition; the sum of all the partitions needs to match the source table row count.)
- Create indexes on the partitioned table, in parallel using the DB2 SMP licensed feature.
- Drop the nonpartitioned table and indexes.
- Rename the new database objects as needed.

Similar processes can be used when migrating from one partitioned table to another, or when migrating back to a nonpartitioned table.

When considering migration to a partitioned table with a programmatic approach, use the following guidelines as a starting point for the number of concurrent processes:

- Use one job per partition for each processor that is available.
- Move as much memory as possible to the memory pool where the concurrent jobs run.
- Ensure that the MAXACTIVE value for the memory pool is set to the number of concurrent jobs.

Depending on the system configuration and the available resources, more than one process per processor can be employed. The goal is to use 100% of the processor resources. This might not be possible, depending on the I/O subsystem capacity and overall system configuration.

When evaluating the creation and use of partitioned tables, it is imperative to consider the data migration time and effort, and to plan accordingly. This is especially true in large database environments or environments where the tables must be online and available most or all of the time. An appropriate and realistic window of time and resources must be provided for the migration effort. This window of time can possibly be measured in days depending on the amount of data that must be moved.

Planning for success

Before migrating to partitioned tables, check with any software vendor who uses the tables being considered. This is especially important if using a high availability solution that logically replicates transaction data to another system.

Before migrating to partitioned tables, gather some baseline metrics on how the SQL requests are optimized and executed using tools such as the Index Advisor, SQL Plan Cache Snapshot, or SQL Performance Monitor. Also, gather some baseline information on how the system resources are being used. Collection Services (which is a performance collection facility in the IBM i operating system) is a good instrument for this. After migrating to partitioned tables, the baseline information can be used to quantify any differences in behavior or performance of the application.

Seriously consider running proofs of concept, proofs of technology and benchmarks to test the migration strategy and the application's database interfaces before implementing partitioned tables. A great place to run a benchmark is the IBM i Performance and Scalability Center in Rochester, Minnesota. Understanding how the application behaves with partitioned tables before moving the code into a production environment is a critical success factor.

You can find more information about the IBM Performance and Scalability Center at: **ibm.com**/systems/services/labservices/psscontact.html

It is also highly recommended to obtain additional knowledge and guidance from the IBM DB2 for i Center of Excellence team at **ibm.com**/systems/services/labservices/platforms/labservices_power.html. Assistance is provided initially with the **DB2 for i Very Large Database (VLDB) Consulting Workshop**. During the workshop, business and technical requirements are gathered, the current environment is studied, and potential solutions are discussed and potentially prototyped. Optionally, application and database modernization concepts and strategies are covered.

You can contact the author to get more information about the VLDB consulting workshop.

Summary

With the latest versions of DB2 for i, IBM continues to deliver additional features and functionality to assist with implementing robust data-centric applications. The ability to partition a table provides yet another option for data storage and data access in a growing environment. This paper, along with the aforementioned publications, provides some guidance and insight on using this feature. Additionally, specific and targeted information can be provided to the DB2 for i Database Engineer on an as-needed basis. You can contact the author for further details.

You can find additional information and insight on IBM i information and data management at: db2fori.blogspot.com

Appendix A: SQL query engine (SQE) restrictions

In IBM i 7.1, SQE is not capable of optimizing and running queries that contain or use:

- Read triggers
- Distributed tables
- Non-SQL interfaces: WRKQRY, RUNQRY,, OPNQRYF, QQQQry API

In IBM i 7.2, SQE is not capable of optimizing and running queries that contain or use:

- Read triggers
- Distributed tables
- Non-SQL interface: QQQQry API

If the query against the partitioned table contains any of these items, then CQE is used to optimize and run the query.

Appendix B: Acknowledgements

Special thanks to Kent Milligan, Tom McKinley, and Rob Bestgen for their input and reviews.

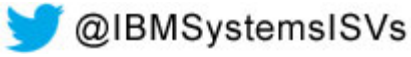
Appendix C: Resources

The following websites provide useful references to supplement the information contained in this paper:

- DB2 for i home page
  ibm.com/systems/power/software/i/db2/index.html
- Indexing and statistics strategy white paper
  ibm.biz/indexing
- IBM i Knowledge Center
  ibm.com/support/knowledgecenter/ssw_ibm_i/welcome
- DB2 for i Manuals
  ibm.com/systems/power/software/ i/db2/docs/books.html
- DB2 for i SQL Performance Workshop
  ibm.biz/SQLPerf
- DB2 for i Center of Excellence
  ibm.biz/db2icoe

About the author

Mike Cain is a senior technical staff member and the team leader of the DB2 for i Center of Excellence in Rochester, Minnesota. Before his current position, he worked as an IBM systems engineer and technical consultant for the IBM i platform. You can send your questions regarding partitioned table support or any DB2 for i topic to: mcain@us.ibm.com.



Trademarks and special notices

References in this document to IBM products or services do not imply that IBM intends to make them available in every country.

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Other company, product, or service names may be trademarks or service marks of others.

Information is provided "AS IS" without warranty of any kind.

All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.

Information concerning non-IBM products was obtained from a supplier of these products, published announcement material, or other publicly available sources and does not constitute an endorsement of such products by IBM. Sources for non-IBM list prices and performance numbers are taken from publicly available information, including vendor announcements and vendor worldwide homepages. IBM has not tested these products and cannot confirm the accuracy of performance, capability, or any other claims related to non-IBM products. Questions on the capability of non-IBM products should be addressed to the supplier of those products.

All statements regarding IBM future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. Contact your local IBM office or IBM authorized reseller for the full text of the specific Statement of Direction.

Some information addresses anticipated future capabilities. Such information is not intended as a definitive statement of a commitment to specific levels of performance, function or delivery schedules with respect to any future products. Such commitments are only made in IBM product announcements. The information is presented here to communicate IBM's current investment and development activities as a good faith effort to help with our customers' future planning.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Photographs shown are of engineering prototypes. Changes may be incorporated in production models.

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