IBM® DB2® for Linux®, UNIX®, and Windows®

Best practices

Managing data growth

Liping Zhang
Senior Software Engineer

Naresh Chainani
Software Developer

Alexandria R. Burkleaux
Software QA Engineer

Roman B. Melnyk
Information Developer

DB2 for Linux, UNIX, and Windows Development

Issued: June 2012

Executive summary

Because of business needs for data retention and regulatory compliance, enterprises need to manage increasingly large databases ranging from hundreds of gigabytes to many terabytes, or even petabytes, in size. As data continues to grow at an exponential rate, DBAs and IT professionals in these organizations face daunting challenges when designing and operating such large databases. The data must be well organized to effectively cope with data growth and to meet service requirements. Challenges include how best to achieve the following objectives:

- Designing databases that can accommodate continuous data growth
- Keeping database systems lean and high performing
- Managing data lifecycles more efficiently and less intrusively to keep operational data highly available
- Reducing the cost and impact of data maintenance operations, such as backup and restore operations, to keep mission-critical data ready when needed, and reorganization and runstats operations to maximize system performance
- Satisfying near real-time requests for transactional data or the complex analytical query requirements for large data sets, often including historical data, while reducing the total cost of ownership (TCO)

DB2 for Linux, UNIX, and Windows offers a rich set of features that help you to meet these challenges and to benefit from winning solutions. This paper describes best practices for managing data growth that you can consider during the stages of database planning, design, implementation, and operation. By leveraging these best practices, DBAs and IT professionals can use DB2 data server's extensible architecture and the layered data partitioning and organization schemes to take full advantage of proven approaches to managing data growth.

Introduction

This paper, whose contents are based on DB2 for Linux, UNIX, and Windows Version 9.7, is primarily intended for database administrators and solution architects in a data warehouse environment who are looking for physical database design and planning information that is applicable to data lifecycle management and data maintenance scenarios characterized by rapid data growth. The best practices described in this paper are also helpful in mixed-workload environments.

Strategies for managing data growth

There are a few general strategies for managing growth in a DB2 database system. Consider these strategies when planning your database system or application. You can also apply them when your system requirements change unexpectedly and you need to adjust accordingly.

Strategy 1: Control data growth

You might use this strategy when you have data that is constantly produced, but you have different access patterns for the data based on its age. Growth could be controlled by keeping the size of data in tables used for key applications constant, even as the size of the overall database grows, or it could be controlled by keeping the size of the entire database constant. Either way, you will need mechanisms for moving new, “active”, data into the tables and moving less relevant, “inactive”, data out of the tables or database. For example, a data warehouse might retain a single quarter's worth of data in its active data set and move older data to near-line storage for retention for regulatory compliance. By separating data according to access patterns, you can keep applications lean and optimize performance. You can also control costs by allocating hardware resources based on requirements for accessing the data.

Strategy 2: Adapt to growth

You might use this strategy when you have a system that needs to maintain good performance even as the size of its active data set continues to grow. For example, your data warehouse has produced so much business value, that now other departments want to take advantage of your application for their analyses. These departments want to include more columns from the transactional data in the warehouse. Another example is business growth. Increases in customers or sales, for example, could cause the same increase in data for your system. The DB2 software provides mechanisms that add capacity to the system to scale for good performance and to reduce the impact of data maintenance operations on increasingly large tables.

Combined strategy: Control and adapt to growth

Controlling and adapting to data growth can be combined into the same strategy. You might use this approach when you can separate data by access pattern, but the size of the

data in the active data set continues to grow due to increases in business or increasing data demands on the application from new use cases.

Database design considerations for data growth

As data volume increases over time, the data might become skewed and fragmented, resulting in decreased performance. These problems can be addressed through good initial planning during the design phase. The DB2 shared-nothing architecture provides unparalleled system extensibility, both in terms of additional processing power and increased storage capacity. Database partitioning, table partitioning, and multidimensional clustering (MDC) are innovative and industry-proven solutions for scalability, manageability, and performance.

The tips in this section are intended to help DBAs take full advantage of these features, either individually or in combination. The discussion covers best practices for the following tasks:

- Designing table spaces with all anticipated database considerations in mind, including query performance, data lifecycle management, archiving, data backup and recovery operations, and other database maintenance tasks
- Choosing the right database partitioning key so that the data can be distributed evenly across all of the database partitions for efficient parallel processing
- Achieving better data collocation for enhanced complex query performance
- Choosing a table partitioning key that facilitates data roll-in or roll-out for efficient data lifecycle management, and that helps complex queries to run faster by using the DB2 optimizer's data partition elimination capability to scan only relevant data partitions
- Determining the right MDC dimensions to achieve better query performance and require less data maintenance
- Choosing the right combination of data organization schemes to maximize benefits

Data lifecycle management

As a database grows, so does the relative amount of old data that is infrequently accessed yet consumes limited resources. Even with good scaling, it is increasingly important to keep the system lean to meet performance requirements and lower the TCO. It is also important to keep the old data available to meet business needs. This is accomplished by controlling the amount of active data and by defining a data retention strategy to handle the data aging process, for example by archiving historical data. Data lifecycle management refers to the management of data from the time at which the data is generated right to the end of its useful life.

The data lifecycle actually starts before the point of injection into a database system, and does not end when the data is rolled out from the active tables. Preparing, cleansing, and transforming the data before putting it into the database is a critical part of the process. Similarly, the management of data both while and after it is rolled out of the system should be considered part of the data lifecycle.

Today's database applications often require existing data to be available 24x7, so that the applications can run without interruption during both data roll-in and roll-out. The best practices for data lifecycle management are covered in the *"Data lifecycle management"* section. The best practices for accessing archived data are also outlined in that section.

Data maintenance

As the amount of data in a database continuously grows, database backups seem to take forever, data reorganization takes a long time, and runstats operations do not complete within the defined maintenance window. A highly active database can experience changing data distribution characteristics, including the balance of data across database partitions. These changes can result in fragmentation, reduced clustering ratios, and degraded performance over time. It is critical to regularly monitor the data distribution characteristics and to maintain current statistics to keep system performance from degrading.

Regular maintenance typically includes reorganizing data and indexes for better clustering, space reclamation, or defragmentation; refreshing statistics to help the optimizer improve data access plans; and redistributing the data in partitioned database environments to eliminate skew. Because such operations can be very time consuming and resource-intensive, especially with large amounts of data, careful consideration should be given during the physical database design phase to reduce or eliminate the impact of these maintenance tasks during database growth.

Another important operational aspect of managing data in a growing database is implementing a good data backup and recovery strategy to ensure business continuity after either planned or unexpected outages.

Responding to changes

No planning is perfect! As business needs change and data continues to grow, the original design assumptions for the database might no longer be valid. Although small changes to storage allocation or memory management can easily be accommodated, in some cases you might need to make significant changes. For example, the growing needs of your business might call for horizontal scaling, which you can address by adding a new database partition. Or the distribution key has developed a lot of skew and no longer seems to be the best choice.

The *"Responding to changes"* section provides guidelines that help you to reconsider your choices around data organization schemes by assessing current realities and anticipating future trends. The following topics are covered in this section:

- Adding new database partitions to accommodate increasing data volume requirements and satisfy user demands
- Modifying the distribution key in response to skew
- Changing table partitioning granularity to accommodate data growth

Database design considerations for data growth

When designing a database, it is important to understand the service requirements and expected growth (as anticipated at the time of planning) so that the system will have sufficient processing and storage capacity. Based on business growth trends, estimate the rate at which data is expected to grow and have a plan for monitoring and extending system capacity. It is also important to physically organize the data for optimal performance and manageability.

To effectively manage data growth, it is important to streamline the following processes:

- Adding new data to the operational database
- Removing old data from the operational database
- Moving old data to cheaper, slower storage for continued availability in the operational database

As the volume of data grows, so too does the cost of storing and maintaining that data. By performing maintenance operations only on the necessary subsets of the data, you can limit the maintenance window and reduce costs. A good database design combines these considerations with the performance, temperature, and data availability requirements of the system.

This section describes some considerations when choosing table spaces and outlines the DB2 data organization schemes. It explains best practices for choosing keys for the different schemes and how the right choice can make the task of managing data growth easy while helping to optimize performance.

Table space considerations

Table spaces enable you to physically group database objects for common configuration and the application of maintenance operations at the table space level rather than at the database level, which significantly reduces the impact on data availability and resource requirements by confining such operations to specific objects.

It is generally recommended that you create separate table spaces for the following objects:

- Regular tables
- Staging tables
- Materialized query tables (MQTs)
- Individual data partitions of range partitioned tables
- Indexes

For a range partitioned table, placing individual data partitions into different table spaces enables you to back up or restore an individual partition without impacting other partitions. For many applications, you can avoid backing up older historical partitions by just

backing up the partition with current data. The merge backup utility can merge the partition backup of current data with a previous full backup image to create a new full backup image. Thus, you can reduce the need to take a full backup of large databases.

Isolating a partition in its own table space helps to reduce fragmentation. For example, if unsorted data is loaded into a range partitioned table where all partitions are in the same table space, the extents that are allocated for the partitions will be interleaved, reducing the performance of table scans. If each data partition resides in its own table space, or the data is sorted on the table partitioning key, fragmentation should be reduced.

Local indexes on range partitioned tables are placed by default in the same table space as their corresponding data partition, and this default behavior should meet the needs of most applications where ease of administration and maintainability are priorities. On the other hand, if query performance is the priority, place each index partition into its own table space so that local indexes use a separate buffer pool from that of the data. This will increase the likelihood of index leaf pages residing in the buffer pool when they are needed. Keep in mind, however, that having each data partition and each index partition in their own table space will increase the number of table spaces, thereby incurring more administrative complexity and cost.

Create each global index on a range partitioned table in a separate table space, because such indexes can become very large. For optimal performance, ensure that the leaf pages of frequently used global indexes reside in the buffer pool. If memory constraints exist, place the most frequently used global indexes in table spaces that have containers on faster disks.

Keep the number of table spaces per database partition below 1000. Having too many table spaces or data partitions can increase administrative complexity and overhead and decrease performance. Consider grouping related tables or multiple data partitions in the same table space based on the archiving strategy, data temperature, or the granularity of backup and restore operations. Use MDC rather than range partitioning for finer granularity partitioning.

For example, if a table is partitioned by week and contains 7 years of data, there are 364 data partitions that reside in 364 table spaces, if each data partition is in its own table space. However, if the business backup policy is to perform daily backups of the current quarter, it might make sense to use MDC to partition by week and utilize quarterly range partitions. Perform daily incremental backups of the table space holding the most recent data. The merge backup utility can be used to merge backup images at another server to create new full backup images. This reduces the number of required table spaces for this table to 28.

Similarly, you can reduce the number of required table spaces based on the granularity of the data that is used for archiving. For example, suppose that data is partitioned by date, with each partition containing one month of data. If data is archived quarterly, having three partitions on a table space will reduce both the number of table spaces and the table space-based administrative overhead that is associated with the archiving procedure.

The DB2 data organization schemes

The DB2 data organization scheme hierarchy is the industry-proven solution to support physical database designs for scalability, manageability, and performance.

- *Database partitioning* distributes table data across multiple database partitions in a shared-nothing manner in which each database partition "owns" a subset of the data.
- *Table partitioning* partitions table data into different storage objects based on the table partitioning column and defined ranges. When database partitioning is combined with table partitioning, identical table partitions are created on each database partition.
- *Multidimensional clustering* physically organizes table data into blocks along one or more dimensions, or clustering keys.

Table 1 shows the clauses on the CREATE TABLE statement that corresponds to each data organization scheme.

Table 1. The DB2 data organization schemes

Data organization scheme	Clause in the CREATE TABLE statement
Database partitioning	DISTRIBUTE BY HASH
Table (or range) partitioning	PARTITION BY RANGE
Multidimensional clustering	ORGANIZE BY DIMENSION

These schemes can be used individually or in combination, based on operational and workload requirements. Figure 1 illustrates the data organization scheme hierarchy for the following table:

```
CREATE TABLE Sales (
    TxID INT,
    OrderDate DATE,
    CustRegion CHAR(5),
    ShipDate DATE,
    ProductID BIGINT,
    CustomerID BIGINT
    )
  DISTRIBUTE BY HASH (TxID)
  PARTITION BY RANGE (OrderDate)
    (PARTITION Jan2009
```

```
      STARTING '2009-01-01' ENDING '2009-01-31' IN Ts1,
    PARTITION Feb2009
      STARTING '2009-02-01' ENDING '2009-02-28' IN Ts2)
    …
    PARTITION Dec2011
      STARTING '2011-12-01' ENDING '2011-12-31' IN Ts36)
  ORGANIZE BY DIMENSIONS (CustRegion);
```

The Sales table is partitioned by month and stores a 36-month history of sales. The Sales table is hash-partitioned by the transaction ID column (TxID) across two database partitions named P1 and P2. The table is range partitioned by month based on the order date (OrderDate). Data can be efficiently rolled in and out every month. Notice how each data partition is in its own table space. Data within each table space is clustered based on sales region (CustRegion), which facilitates aggregation for complex analytical queries.

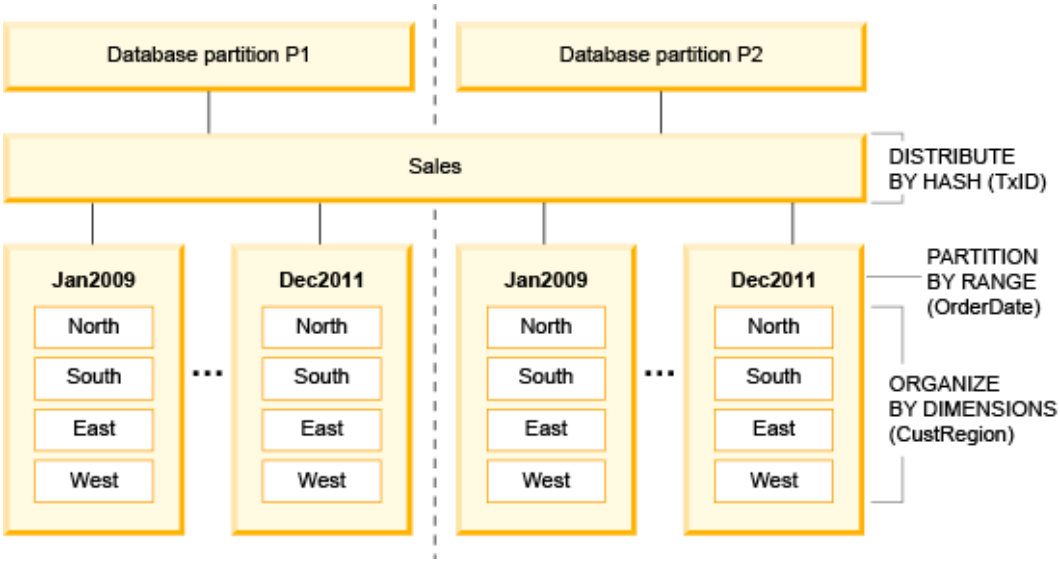


Figure 1. The example data organization scheme hierarchy

Database partitioning

Database partitioning helps you to adapt to data growth by providing a way to expand the capacity of the system and scale for performance. In a partitioned database environment, a database is divided into database partitions, and each database partition has its own set of computing resources, including CPU and storage. Each table row is distributed to a database partition according to the distribution key that was specified in the CREATE TABLE statement. The DB2 data server provides near-linear scalability when

new database partitions are added to expand the processing power and storage capacity of the system.

A *distribution key* is a column (or group of columns) that is used to determine the database partition in which a particular row of data is stored. Index data is also partitioned with the corresponding table data and stored locally at each database partition. When a query is processed, the request is distributed to each database partition so that subsets of the data can be processed in parallel.

Choosing the distribution key

The following guidelines will help you to choose a distribution key.

- Choose the distribution key from those columns having the highest cardinality. Unique keys are good candidates. Columns with uneven data distribution or columns with a small number of distinct values might result in skew, where query processing involves more work on a subset of database partitions and less work on others.
- Choose the distribution key from columns with simple data types, such as integer or fixed-length character; this will improve hashing performance.
- Choose the distribution key to be a subset of join columns to facilitate join collocation.
- Avoid choosing a distribution key with columns that are updated frequently.
- In an online transaction processing (OLTP) environment, ensure that all columns in the distribution key participate in transactions through equality predicates. This ensures that an OLTP transaction is processed within a single database partition and avoids the communication overhead inherent with multiple database partitions.
- Include columns that often participate in a GROUP BY clause in the distribution key.
- Unique index key columns must include all of the distribution key columns. The DB2 database manager creates unique indexes to support unique constraints and primary key constraints, and these system-defined unique indexes have the same requirement that the constraint key columns must be a superset of the distribution key.

Best practices for table collocation

As data grows, complex queries performing joins can result in an increasing amount of data interchange between database partitions. Collocation needs to be factored into your database design to maintain query performance as data volumes increase.

The following conditions facilitate collocation:

- The tables being joined are defined in the same database partition group.
- The tables have the same number of columns in the distribution key, and the data types of key columns are partition compatible.
- For each column in the distribution keys of the joined tables, an equijoin predicate is used in the WHERE clause of the query.

To collocate multiple dimension tables with one fact table, collocate the fact table with the largest commonly joined dimension table when data skew is not significant, and then replicate the rest of the dimension tables. If the dimension tables are large, replicate a vertical subset of commonly used columns. If you already have a complete set of replicated

dimension tables, the choice of partitioning the fact tables should be based on the common fact-fact table joins.

When the distribution key that is chosen on the basis of collocation conflicts with the one that is based on data balancing, choose the one that will help to distribute the data evenly.

Table partitioning

Table partitioning (also known as range partitioning) helps with managing growth by reducing the impact of maintenance operations, such as table reorganization and backup. For example, as data grows, you need to reorganize only active data partitions instead of the entire table. Applications can take advantage of logical partitioning so that queries only access a subset of the table rows to satisfy range queries. Table partitioning facilitates fast roll-out of data, which helps to manage the size of the table by removing inactive data efficiently. With table partitioning, rows that satisfy a defined range of column values are stored together in a data partition. Although a partitioned table is a single entity that queries can access in the same way as a nonpartitioned table, each data partition is actually a separate database object.

Table partitioning has the following benefits pertaining to data growth management:

- *Improved query performance*: Query performance can be enhanced by a technique known as *partition elimination*, in which the DB2 optimizer directs a query to access only those data partitions that contain rows in the result set. The DB2 optimizer is data partition-aware and can limit query access to relevant partitions on the basis of query predicates that are defined on the table partitioning key.
- *Optimized data roll-in process*: The ALTER TABLE…ATTACH PARTITION statement instantly attaches a data partition with pre-loaded data to a partitioned operational table.
- *Efficient data roll-out process*: The ALTER TABLE…DETACH PARTITION statement detaches a data partition containing obsolete data from a partitioned operational table that remains accessible to read or write queries.
- *Flexible database administration*: Table partitioning facilitates the management of very large tables by adopting a "divide and conquer" strategy. This concept is also known as partition independence, and refers to the fact that maintenance operations on a single data partition do not impact access to the rest of the table. You can place ranges of data across multiple table spaces and then leverage table space-level backup and restore operations to back up critical data more frequently. Offline data reorganization can also be done on individual data partitions.
- *The option to create local or global indexes on a range partitioned table*: Each global index can be placed in a different table space for space management or performance tuning reasons.
- *Better compression*: Each data partition has its own compression dictionary. If partitioning is based on time, compressed data in older partitions is not affected by the changing characteristics of newly inserted data. The newly inserted data is placed in new partitions with their own compression dictionaries.

- *Improved concurrency*: The impact of lock escalation is minimized because the DB2 data server locks at the data partition level rather than at the table level, which is the case for nonpartitioned tables.
- *Alignment with multi-temperature technology*: Table partitioning and the concept of data temperature share the same time-based view of data. These two technologies complement each other and reduce the TCO; data partitions containing older data can be moved from a faster, more expensive storage tier to an inexpensive storage tier as data ages.

Table partitioning tips:

- Limit the number of data partitions per table to a few hundred per database partition. A table with 120 data partitions and monthly ranges can hold 10 years of data. Thousands of data partitions increase the metadata overhead, complicate administration, and can negatively impact DML performance.
- Ensure that data is more-or-less evenly distributed across data partitions. The DB2 optimizer assumes that the table data is homogeneously distributed across all data partitions. Extreme skew between partitions can lead to sub-optimal query plans. For example, if December holiday sales are anticipated to be significantly more than the slowest month, consider creating two ranges for the December data; one for the first two weeks of December and another for the last two weeks.
- Avoid creating too many empty data partitions for future data. Having many empty partitions can contribute to skewed data distribution statistics. On the other hand, adding one partition at a time might not be the best strategy either, because ADD PARTITION requires a Z lock on the entire table. A reasonable compromise is to add a few data partitions during each maintenance window.

Choosing the table partitioning key

Choosing the right table partitioning key is critical to taking full advantage of this data organization scheme. The following guidelines will help you to choose a table partitioning key.

- Select columns that facilitate data roll-out. For example, consider a business that retains sales information in a Sales table for 36 months after the sales date. The DBA can choose OrderDate as the table partitioning key and use the ALTER TABLE…DETACH PARTITION statement to efficiently roll out the data corresponding to the oldest month.
- Select columns that facilitate data roll-in. For example, consider a business that ingests data into the Sales table based on the OrderDate attribute. The warehouse can ingest a month of sales data instantly by using the ALTER TABLE…ATTACH PARTITION statement if the table partitioning key is OrderDate. However, if the data roll-in and roll-out requirements are different, the table partitioning key should favor the roll-out requirements, and the LOAD command, INSERT statement, or an alternate data organization scheme (such as MDC) should be considered for data roll-in.
- Select columns that facilitate partition elimination. For example, if most reporting queries against the Sales table have a predicate on OrderDate that identifies a particular quarter, the access plan targets only the three data partitions containing data for

that quarter. Note that if the table partitioning key is a composite key, data partition elimination is possible only if there are predicates on the leading column of the composite key, because non-leading columns are not independent.

- For any global clustering indexes, prefix the index key columns with table partitioning key columns to achieve optimal clustering. Clustered local indexes, on the other hand, are always clustered on the basis of data in each partition and do not need the partitioning key to be prefixed.
- Use multiple columns for the table partitioning key when the data timeline is represented by more than one column; for example, year and month columns. The following code example shows how to partition such data into quarters:

```
CREATE TABLE SalesData (OrderID INT, Year INT, Month INT)
PARTITION BY RANGE (Year, Month)
 (PART Q1_2011 STARTING (2011,1)  ENDING (2011, 3),
  PART Q2_2011 STARTING (2011,4)  ENDING (2011, 6),
  PART Q3_2011 STARTING (2011,7)  ENDING (2011, 9),
  PART Q4_2011 STARTING (2011,10) ENDING (2011, 12));
```

Local index versus global index

Table partitioning supports both local indexes (also known as partitioned indexes) as well as global indexes (also known as nonpartitioned indexes). Local indexes enable the concept of partition-level independence, which becomes increasingly important as data volumes grow. Partition-level independence allows maintenance operations at the granularity of a partition, thereby minimizing the impact on the overall database.

Use local indexes to streamline data roll-in and roll-out. This reduces the impact of the SET INTEGRITY statement. Global indexes can adversely impact roll-in (due to long-running SET INTEGRITY operations) and roll-out (due to asynchronous index cleanup) but can help improve the performance of some queries that need to sort large amounts of data.

It is important to consider the table partitioning key and possible unique indexes on the table when planning. If unique indexes are to be partitioned, the unique index key must be a superset of the table partitioning key. Use global indexes if it is not viable to include the table partitioning key as part of the unique index key.

Multidimensional clustering

Multidimensional clustering (MDC) provides an elegant method for clustering table data across multiple dimensions using block indexes. MDC tables automatically maintain the clustering of data, thereby eliminating the need to reorganize for clustering. This ability to reduce maintenance costs makes MDC attractive in a rapidly growing database environment.

For complex queries that access large amounts of data, block index scans are more efficient, because the block indexes are smaller and yield I/O savings.

Choosing MDC table dimensions

Follow these guidelines to choose dimension columns.

- Choose columns that are frequently used in query predicates or the GROUP BY clause. The DB2 optimizer considers access plans that use block indexes. When queries have predicates on dimension values, the optimizer can use the block index to identify (and fetch from) the extents that contain these values. Because extents are physically contiguous pages on disk, this minimizes I/O and improves performance.
- Choose columns that have a moderate number of distinct values to avoid sparsely-populated cells.
- Use generated columns to limit table cardinality when there are no obvious candidate columns. In the following example, a built-in function is used to convert date values into year-month values (12 values per year), thereby significantly reducing the cardinality.

```
CREATE TABLE Sales (
  OrderId BIGINT,
  OrderDate DATE,
  YearMonth INT GENERATED ALWAYS AS
    INTEGER(DATE(OrderDate))/100
  )
  ORGANIZE BY (YearMonth);
```

- Choose columns that facilitate data roll-in. Ingesting data into an MDC table is faster than ingesting data into a regular table, because block indexes require less maintenance when the blocks for a key already exist and there is room for more data.
- Choose columns that facilitate data roll-out. Deleting data from an MDC table is faster than deleting rows from a non-MDC table, because entire blocks are marked as rolled out and any RID indexes can be cleaned up asynchronously after the transaction commits.
- Validate your choices with the DB2 Design Advisor (use the db2advis command). If table partitioning and MDC are being considered for the same table, define the table partitioning scheme before using the DB2 Design Advisor for MDC recommendations.

MDC space usage considerations

Estimate the total cells needed by an MDC table by multiplying the number of distinct values in each dimension with a query like: SELECT DISTINCT dim1, dim2, ... dimN from <TableName>. In a fast growing database, the table size can increase rapidly if new distinct values are inserted for the dimension key columns, because each unique combination of dimension column values is stored in separate cells.

After MDC is deployed, periodically check the space usage to detect excessive space consumption. One important factor to consider is the total number of active blocks versus the total number of rows in a table. The closer these two numbers are to one another, the worse the space wastage, which means that the density of the cell is low and many of the blocks hold only a few rows. When using MDC in a partitioned database environment, space wastage can occur on each database partition.

When significant space wastage is detected, revisit the MDC table definition and redesign the dimension keys, but this can be very time and resource consuming. That is the reason we emphasize the importance of choosing the MDC dimension keys before creating the table. During the redesign phase, consider using a generated column to roll up the granularity, or use a table space with smaller extent size.

Choose dimension key columns to improve query performance and to ingest data quickly. Ensure that MDC table cells are dense for best space utilization. If necessary, use generated columns to increase the density of the cells.

Bringing it all together

The following table summarizes the advantages of each data organization scheme and lists some of the scenarios that can benefit from each scheme as data grows.

Table 2. Data organization scheme summary

	More users, increasing workload	Data lifecycle management	Data maintenance	Query performance
Database partitioning	Increased capacity for additional power and storage		Database partition-level backup, restore, and reorganization	Parallel processing, collocation for joins
Table partitioning	Separation of recent and historical data	Historical data roll-out, new data roll-in	More frequent backup and restore of subsets of the table data, ability to reorganize a single data partition	Partition elimination
Multidimensional clustering	Clustering automatically maintained	Fast deletion of stale data, efficient insertion of new data	Auto clustering, automatic addition of storage blocks, no need for reorganization	Complex queries on the dimension columns, efficient block index scans

Use database partitioning, table partitioning, and multidimensional clustering together under the following scenarios:

- Data volumes are very large.
- There is a need for periodic data roll-in or roll-out.
- The data has a distinct lifecycle and access patterns that are based on time or specific ranges of certain values. For example, "hot" data that is frequently accessed or modified (and therefore fragmented), so that periodic reorganization is required, or histori-

cal data that is no longer required in service but that should be rolled out and archived.
- Some columns with a moderate number of distinct values are frequently used in query predicates.

Depending on your business requirements, choose one or more of these data organization schemes together for best results. The following scenarios provide further guidance.

Scenario where combining table partitioning with MDC can be beneficial

Suppose a health insurance provider has a large claims table that stores, among other things, the plan ID (indicating the plan to which a claim belongs) and claim date (the date on which the claim was received by the insurance company). Assume that the provider offers 100 different plans and deals with millions of claims every week. The business retention policy for claims data is 7 years. Regular reports aggregate data over some period of time (month/quarter/year), with queries returning results such as the number of claims received last month, or the total dollar value of all claims paid out.

In this scenario, table partitioning by month on the claim date results in 84 data ranges for the table and facilitates the efficient roll-out of the oldest data at the end of each month. Multidimensional clustering on the plan ID facilitates both the loading of data and query processing by using small efficient block indexes. As data grows, reporting queries continue to access only a subset of the data ranges rather than the entire table.

Scenario where one data organization scheme might be a better choice

If using table partitioning could result in thousands of data partitions, MDC can be a more attractive alternative to consider.

Materialized query tables

Use materialized query tables (MQT) in a data warehouse environment to improve query performance. The rerouting of queries to an MQT is performed automatically by the optimizer. Weigh the costs against the potential benefits of using MQTs in your environment. There are storage overheads when using MQTs and maintenance overheads to keep them up-to-date when the base table is modified.

For large range partitioned fact tables, create range partitioned MQTs using the same partitioning strategy. Not only does this enable partition elimination for complex queries routed to the MQT, it also streamlines MQT maintenance. When old data is detached from the base table, the MQT can be instantly refreshed by detaching the corresponding data partition from the MQT. When new data is ingested into a base table, dependent MQTs must be refreshed. Alternatives to refreshing MQTs after data roll-in on the base table are described in the section *"Partitioned MQT maintenance after rolling data into a base table"*.

Replicate small dimension tables to improve collocation for join query performance in a partitioned database environment. The technique to accomplish this involves creating MQTs over the dimension tables. When dimension tables are large, however, it might not be a good idea from a storage cost perspective to replicate them.

Deploy table partitioning for very large dimension tables to leverage the benefits of fast data roll-in, roll-out, and partition elimination. Replicating such large dimension tables is not recommended because of increased storage overhead.

Use materialized query tables to increase the performance of expensive or frequently used queries that aggregate large amounts of data. Replicate small dimension tables for join collocation in a partitioned database environment.

Compression

Compression is recommended for very large tables. It can provide significant space savings and also improve query performance, because fewer I/O operations are required to access the same amount of data. In a data warehouse, it is not uncommon to see space savings of up to 60% or more when compression is used. Buffer pool hit ratios are improved, because more data can fit in the buffer pool. User data in log records is compressed too, yielding further storage savings. All of these savings translate into less I/O and better throughput during query processing. Additionally, compression reduces time and space requirements for backup and restore operations, and for log archiving.

Compression can be enabled by specifying the COMPRESS YES option on the CREATE TABLE statement or by using the ALTER TABLE statement to enable the compression flag. A compression dictionary is created automatically when the table reaches a certain size or, alternatively, you can manually issue a REORG TABLE command to effect compression.

As data continues to grow, the compression ratio is likely to get worse, because the effectiveness of compression depends on the data and the quality of the dictionary that exists is based on an old view of the data. Monitor the effectiveness of compression periodically as data grows by comparing the current savings from compression to the estimated savings if you were to reset the compression dictionary. The ADMIN_GET_TAB_COMPRESS_INFO_V97 table function can be used to report the existing compression information, as well as estimate compression information for a table, and this metadata can help you to determine whether it is worthwhile to reset and rebuild the compression dictionary.

Compression tips:

- Compress large tables for better query performance and storage savings.
- Use the REORG TABLE…RESETDICTIONARY command after the table is fully populated. This can yield higher compression ratios because the compression dictionary is based on a complete data set.
- Enable index compression by using the ALTER INDEX statement, followed by index reorganization to rebuild the (compressed) index.

- As data grows, estimate the compression savings from rebuilding the existing compression dictionary and take action to maximize compression benefits.
- Use the ADMIN_GET_TAB_COMPRESS_INFO_V97 and ADMIN_GET_INDEX_COMPRESS_INFO administrative functions to display current and estimated potential compression ratios for each table and index.
- Optim Configuration Manager 2.1.1 introduces the capability to automate the process of obtaining and evaluating compression estimates.

Data lifecycle management

As data volumes grow, it is important to keep the system lean by managing the amount of active data. In some cases, it is also important to keep the old data available to meet business needs.

This section covers the following topics that will help you to manage large data volumes associated with rapidly growing databases in data warehouse or mixed-workload environments:

- Adding new data
- Removing historical data for archiving or purging
- Maintaining MQTs and referential integrity
- Accessing historical data

Adding new data: Which solution to use?

Consider the following factors when choosing a solution for adding new data in your environment:

- Is data "trickling in" or already available in files or tables?
- Is the target table required to be online while new data is being added?
- Should new data be immediately accessible?
- How much system resource, such as log space or locking, is required during the operation?

There are four methods to add new data into a database:

- ALTER TABLE…ATTACH PARTITION statement, followed by the SET INTEGRITY statement, against a range partitioned table
- ALTER TABLE…ADD PARTITION statement, followed by the INSERT statement or the LOAD command, against a range partitioned table
- INSERT statement
- LOAD command

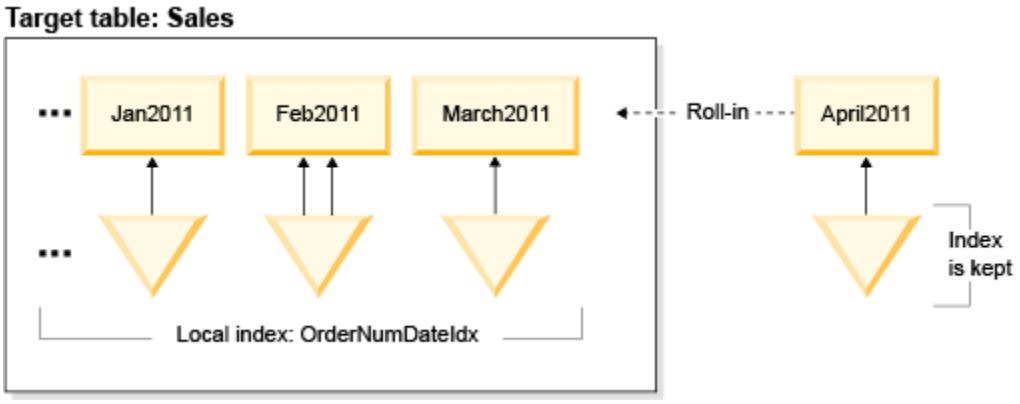
Method 1: ATTACH PARTITION + SET INTEGRITY

This method provides a fast way for adding new data to a range partitioned table by attaching another table (the source table) as a new data partition of the range partitioned

table (the target table). This method is appropriate if the new data is available "in bulk" and if the target table is partitioned along some temporal scale. All of the new data will be available to applications at the same time.

During attach, the data object of the table being attached becomes a new partition of the target table. Indexes from the source table become index partitions for the new partition if they match any of the local indexes that are defined on the target table. Because there is no data movement involved, this method is extremely fast. The cost of index maintenance can largely be avoided by using local indexes (Figure 2). The SET INTEGRITY statement is needed to perform integrity processing and range validation on the new data, and to maintain global indexes for the new rows.

Figure 2. Roll-in with local indexes



The following example shows the two steps that are required to complete the roll-in operation:

```
Step 1:
  ALTER TABLE Sales ATTACH PARTITION
    Jan2012 STARTING ('2012-01-01') ENDING ('2012-01-31')
    FROM TABLE CurrentSales REQUIRE MATCHING INDEXES;
  COMMIT;
Step 2:
  SET INTEGRITY FOR Sales IMMEDIATE CHECKED;
  COMMIT;
```

The CurrentSales table holds transaction data for the current month. At the end of the month, this table is attached to the Sales table, and a new CurrentSales table is initialized for the following month. After Step 1, the existing data partitions are available for read and write access; however, the data in the newly attached data partition remains invisible until the SET INTEGRITY statement commits.

Roll-in best practices

- Use local indexes instead of global indexes on a range partitioned table to streamline data roll-in and roll-out.
- Prepare the source table of an attach operation by creating indexes on the source table that match all of the local indexes on the target table. Use the optional REQUIRE MATCHING INDEXES clause on the ALTER TABLE…ATTACH PARTITION statement to enforce this best practice.
- Define and enforce all applicable data constraints, such as check constraints or range checking, in the extraction, transformation, and loading (ETL) process to ensure that the data is compliant prior to issuing the ALTER TABLE…ATTACH PARTITION statement. This will reduce the amount of work performed by the SET INTEGRITY statement, thereby making new data available sooner.
- If data is not cleansed, specify an exception table on the SET INTEGRITY statement to reduce the chance of a long-running SET INTEGRITY operation failing; this also enables you to monitor and correct errors in the data.
- Issue a COMMIT statement after each step of the roll-in process to release locks.
- Set LOCK TIMEOUT to WAIT to prevent a long-running SET INTEGRITY statement from failing because of a possible lock conflict.

Method 2: ADD PARTITION

The ALTER TABLE…ADD PARTITION statement adds an empty data partition to an existing range partitioned table. Following this, you can use the INSERT statement or the LOAD command to populate the new partition. Because the ADD PARTITION operation also places an exclusive lock on the target table, a COMMIT statement right after the ALTER TABLE…ADD PARTITION statement is recommended. A SET INTEGRITY statement is not required following an ADD PARTITION operation.

```
ALTER TABLE Sales ADD PARTITION Q1_2012

  STARTING ('2012-01-01') ENDING ('2012-03-31')

  IN Q1_2012Tbsp INDEX IN Q1_2012IdxTbsp;

COMMIT;
```

ADD PARTITION is a good alternative to ATTACH PARTITION when you need to ingest large volumes of data and the table has many global indexes. It is also an appropriate method when the target table needs to grow to accommodate new data, but the new data is "trickling in" rather than available in bulk.

Because you can control the frequency of COMMIT statements after insert or load operations, you can minimize the consumption of active log space with frequent commit operations.

Add a *few* partitions at each maintenance window to minimize the impact of frequent ADD PARTITION operations to expand table capacity. Avoid adding too many partitions in advance, because a large number of empty data partitions distort some ta-

ble-level statistics, which might adversely affect the query optimizer's ability to choose an optimal access plan.

Method 3: INSERT

Consider using this method of adding data with MDC tables. The INSERT statement offers the best concurrency among all of the methods to populate a table. A disadvantage of this approach is the logging overhead and potential row locking if data volumes are large. In a data warehousing environment, this method is often used with MDC tables. Locking at a block level reduces lock memory consumption. Moreover, block indexes are updated only when a new block is created, not after each row is inserted, as is the case with regular indexes on non-MDC tables.

Method 4: LOAD

The DB2 load utility can quickly ingest large amounts of data into any table. It is particularly useful for populating tables in a data warehouse environment and works well with all organization schemes.

The ALLOW READ ACCESS option keeps the table data available for concurrent applications during most of the load operation. Keep in mind that the load utility acquires a super-exclusive lock (Z-lock) for a short duration of time near the beginning and the end of the process, regardless of what access option has been specified.

In some cases, applications that generate long running reports might not be able to drain queries prior to execution of the load utility. The best practice in such cases is to use the load utility to rapidly load data into staging tables, and then to populate the primary tables using an INSERT statement with a subselect. It is also good practice to understand the workload and schedule load operations to occur when report applications are not running.

Considerations when using LOAD

- Because load operations can only be done at the table level, even if the data is targeting one particular data partition, access options such as ALLOW NO ACCESS or READ ACCESS specify the level of access to the entire table.
- Loading data directly into a subset of data partitions while the remaining partitions remain fully online is not supported. You have to perform a DETACH PARTITION, LOAD, ATTACH PARTITION + SET INTEGRITY sequence to achieve that result. When there are no global indexes on the table, this option is a reasonable choice for many scenarios.
- The load utility can insert data records into the correct data partitions. There is no requirement to partition the data in some external ETL tool before loading.
- Pre-sorting the data on the table partitioning key or the MDC key can improve load performance. Do not sort the data on the distribution key, because that can negatively impact LOAD inter-parallelism.

ETL tools facilitate data roll-in

In nearly all production systems, data is cleansed by an ETL process before the data is added to a database. Generally speaking, therefore, you can view ETL as part of the data roll-in process. The key point is that ETL can perform all of the cleansing, transforming, sorting, aggregating, and joining of data outside of the operational database system. Data preparation during ETL facilitates data roll-in and reduces its impact on the production database. Pre-sorting the data on the table partitioning key can facilitate the loading of large volumes of data into a range partitioned table. Creating appropriate indexes on the source table can help data roll-in. IBM InfoSphere DataStage is one example of the ETL tools that are widely used in the industry.

Removing old data: Which solution to use?

The choice of which data roll-out method to use is typically based on the following factors:

- Is the target table accessible while data is being rolled out?
- How much time is required to move data out of operational tables?
- What are the recoverability options?

There are basically three methods to roll out or remove data from an operational table:

- ALTER TABLE…DETACH PARTITION statement against a range partitioned table
- DELETE statement
- TRUNCATE statement

Method 1: DETACH PARTITION

This method provides fast roll-out of large amounts of data from a range partitioned table by detaching a data partition and the corresponding local index partition from the table (the source table) into a stand-alone result table (the target table). Because there is no data movement involved, the operation requires minimal logging and can easily be undone.

You can archive the target table or purge it, depending on your business requirements. You can roll the target table into another range partitioned table that maintains historical data. This method for data roll-out operates online and provides an extremely efficient means of moving historical data out of an operational table or removing obsolete data from a warehouse. During a DETACH PARTITION operation, long-running reporting queries can continue to access the entire table.

Partition detach is a two-phase operation. The first phase, which is performed by the ALTER TABLE…DETACH PARTITION statement, logically detaches the data partition from the table. The second phase, which is managed by the system in the background without the need for user intervention, completes the physical detach asynchronously after ensuring that all other existing activities that access the partitioned table have completed. This background process is referred to as the asynchronous partition detach

(APD) process. In most cases, you will not notice or care when the APD process is finished. If you need to take subsequent action, you can monitor APD progress by using the following approach:

1. Run the LIST UTILITIES SHOW DETAIL command.
2. Query the STATUS column of the SYSCAT.DATAPARTITIONS catalog view for the partition in question. You can use a program to determine when the APD process has finished; for an example of how to implement such a program, see *"Online roll-out with table partitioning in InfoSphere Warehouse"*.

In the absence of detached dependent tables, the asynchronous partition detach task starts after the transaction issuing the ALTER TABLE…DETACH PARTITION statement commits.

If there are any dependent tables that need to be incrementally maintained with respect to the detached data partition (these dependent tables are referred to as detached dependent tables), the asynchronous partition detach task starts only after the SET INTEGRITY statement is run against all detached dependent tables.

The following example shows the basic syntax of the roll-out operation:

```
ALTER TABLE Sales DETACH PARTITION Jan2011 INTO Jan2011Sales;

COMMIT;
```

Issue a COMMIT statement immediately after the ALTER TABLE…DETACH PARTITION statement completes. If there are dependent tables, complete SET INTEGRITY processing on all of the dependent tables as soon as possible, so that the second phase of the DETACH PARTITION operation can begin.

If you need to detach multiple data partitions during a maintenance window, detach the partitions one at a time and issue a COMMIT statement immediately after each ALTER TABLE…DETACH PARTITION statement. Let the first APD process complete before issuing another ALTER TABLE…DETACH PARTITION statement; this avoids a potential locking conflict during concurrent updates of the metadata.

Method 2: DELETE

Consider using this method of deleting data with MDC tables. The DELETE statement offers the best concurrency among all of the methods to remove data from a table. A disadvantage of this approach is the logging overhead and potential row locking if data volumes are large. In a data warehousing environment, this method is often used with MDC tables. Locking at a block level reduces the lock memory consumption.

Roll-out deletion can be a good option for MDC tables, particularly if all of the data is being deleted or the DELETE statement includes predicates on the dimension key columns, which results in reduced logging (page-level rather than row-level logging).

DELETE performance with MDC tables can be further improved by deferring the cleanup of any RID indexes. This can be done in either of the following ways:

- Set the DB2_MDC_ROLLOUT registry variable to DEFER
- Use the SET CURRENT MDC ROLLOUT MODE DEFERRED statement to override the default value of the DB2_MDC_ROLLOUT registry variable for the duration of the application connection

By default, RID indexes are cleaned up immediately as part of the transaction that performs the deletion. Not only does this index maintenance operation consume logging resources, it is also time consuming, in proportion to the number of RID indexes and the amount of data being deleted. When the deferred mode is used, index maintenance is faster, because the database manager drives index cleanup asynchronously and in parallel after the DELETE transaction commits.

When deleting a large amount of data from an MDC table with many RID indexes, set the CURRENT MDC ROLLOUT MODE special register to DEFERRED; this improves performance and reduces logging overhead. For smaller MDC tables, the overhead of asynchronous cleanup can outweigh any benefits. Deferred roll-out is not supported for range partitioned RID indexes.

Method 3: TRUNCATE

Use the TRUNCATE statement to quickly delete all of the data from an existing table. Because this statement does not perform row-level logging, a table truncation operation is very fast but cannot be undone.

To simulate the effect of truncating a data partition in a range partitioned table, the partition can be rolled out using an ALTER TABLE…DETACH PARTITION statement, and the resulting table truncated, followed by an ALTER TABLE…ATTACH PARTITION statement. If there are no global indexes defined on the partitioned table, this procedure works well. For example:

1. Detach the data partition that should be truncated.

```
ALTER TABLE Sales DETACH PARTITION Jan2012 INTO Jan2012Sales
```

2. Wait for the DETACH operation to complete asynchronously. For information about how to implement this wait, see *"Method 1: DETACH PARTITION"*.
3. Truncate the partition, which is now a stand-alone table.

```
TRUNCATE TABLE Jan2012Sales IMMEDIATE
```

4. Optionally, roll new data into that range, using the LOAD command or the INSERT statement.
5. Reattach the partition.

```
ALTER TABLE Sales ATTACH PARTITION Jan2012

  STARTING ('2012-01-01') ENDING ('2012-01-31')
```

```
  FROM TABLE Jan2012Sales REQUIRE MATCHING INDEXES
```

An alternate approach that provides you with the opportunity to reconsider table space placement is to detach data partitions and then add new (empty) partitions instead of re-attaching the old partitions.

Maintaining referential integrity as part of data lifecycle management

In a star schema configuration, the fact table typically leverages various data organization schemes for efficient access to large volumes of data. In some environments, the dimension tables can be large and might benefit from table partitioning. Fact and dimension tables are related through referential integrity. A dimension table is the parent with a defined primary key, and the fact table is a child that has a foreign key relationship with one or more dimension tables.

When attaching a data partition to a child table, the subsequent SET INTEGRITY operation enforces referential integrity constraints, among other things. If there is no exception table, SET INTEGRITY processing will fail if constraints are violated. Even with an exception table, SET INTEGRITY performance can suffer when violations are detected, because removing rows from the base table and inserting them into an exception table takes time. The best practice is to have the ETL process perform referential integrity checking to ensure that there are no violations.

When detaching a data partition from a child table, it might be necessary to roll out data from the parent table as well. However, the ALTER TABLE…DETACH PARTITION statement cannot be run against parent tables, because there is no automatic way of avoiding orphans in the child table after a DETACH operation. There is a work-around that must be used with caution, however, because it can compromise the integrity of your database if used incorrectly. The following example shows you the correct way to proceed if you want to use this work-around. Complete these steps in a single transaction to avoid lock time-outs.

1. Detach a data partition from each child table C.
```
ALTER TABLE C DETACH PARTITION Part1 INTO ChildDetached
```

2. Convert the RI constraint into an informational constraint for every child table C that has a foreign-key relationship with the parent table P from which you want to detach a data partition.
```
ALTER TABLE C ALTER FOREIGN KEY Fk NOT ENFORCED
```

3. Detach a data partition from the parent table P.
```
ALTER TABLE P DETACH PARTITION Part1 INTO ParentDetached
```

4. Place each child table C in SET INTEGRITY pending state to avoid expensive table scans when the RI constraint reverts to being enforced.
```
SET INTEGRITY FOR C OFF
```

5. Convert the informational constraint back to an enforced constraint on each child table.

```
ALTER TABLE C ALTER FOREIGN KEY Fk ENFORCED
```

6. Bring each child table C out of SET INTEGRITY pending state. If you suspect that data integrity might have been compromised while the constraint was not being enforced, use the IMMEDIATE CHECKED option instead of the IMMEDIATE UNCHECKED option. The former ensures data integrity but can take a long time to complete, depending on data volume.

```
SET INTEGRITY FOR C ALL IMMEDIATE UNCHECKED
```

7. Commit the transaction.

```
COMMIT
```

Maintaining MQTs as part of data lifecycle management

Materialized query tables are very popular in a data warehouse environment. When data in the underlying base table changes as a result of ongoing data roll-in and roll-out activities, MQTs become obsolete and need to be refreshed.

Use a SET INTEGRITY statement to update REFRESH IMMEDIATE MQTs after attaching data partitions to underlying range partitioned base tables. Include the base tables and MQTs in the same SET INTEGRITY statement. This prevents the new data from having to be scanned multiple times, and reduces the total amount of time that is required to check constraints on the base tables, maintain global indexes, and refresh the MQTs.

Similarly, use a SET INTEGRITY statement to update REFRESH IMMEDIATE MQTs after detaching data partitions from underlying range partitioned base tables.

Partitioned MQT maintenance after rolling data into a base table

Partitioned MQTs are often used when the base tables are partitioned. Maintaining a partitioned MQT can be tricky after data roll-in on the base table using an ALTER TABLE…ATTACH PARTITION statement, because ATTACH PARTITION is not directly supported on a partitioned MQT.

There is a workaround. The following example shows you how to proceed if you want to use this workaround. Assume that SalesMqt is a partitioned MQT on the Sales table, and that when the CurrentSales table is attached to Sales, CurrentSalesMqt (which needs to be a regular table, not an MQT) must be attached to SalesMqt (which is an MQT).

1. Convert the target MQT (SalesMqt) into an ordinary table.

```
ALTER TABLE SalesMqt DROP MATERIALIZED QUERY;
COMMIT;
```

2. ATTACH a new partition to the base table.

```
ALTER TABLE Sales ATTACH PARTITION Jan2012
  STARTING ('2012-01-01') ENDING ('2012-01-31')
  FROM TABLE CurrentSales REQUIRE MATCHING INDEXES;
```

```
COMMIT;

SET INTEGRITY FOR Sales IMMEDIATE CHECKED;
COMMIT;
```

3. ATTACH the source table CurrentSalesMqt (which is based on the MQT schema but is not an MQT) to the target MQT (which is an ordinary table now).

```
ALTER TABLE SalesMqt ATTACH PARTITION Jan2012
  STARTING ('2012-01-01') ENDING ('2012-01-31')
  FROM TABLE CurrentSalesMqt REQUIRE MATCHING INDEXES;
COMMIT;
```

4. Run a SET INTEGRITY statement against the target MQT.

```
SET INTEGRITY FOR SalesMqt IMMEDIATE CHECKED
COMMIT;
```

5. Convert the SalesMqt table back into an MQT. The subsequent SET INTEGRITY statement bypasses a full refresh of the MQT.

```
ALTER TABLE SalesMqt ADD MATERIALIZED QUERY <original MQT defi-
nition here>;
COMMIT;

SET INTEGRITY FOR SalesMqt ALL IMMEDIATE UNCHECKED;
COMMIT;
```

Detaching data partitions from a partitioned MQT

The ALTER TABLE…DETACH PARTITION statement can be applied directly to range partitioned MQTs. The behavior is the same as detaching a data partition from a regular range partitioned table. Use the SET INTEGRITY IMMEDIATE UNCHECKED statement to skip data validation following a data roll-out operation against a partitioned MQT. For example:

```
-- DETACH from base table

ALTER TABLE Sales DETACH PARTITION Jan2012 INTO Sales_Jan2012;

-- DETACH from MQT

ALTER TABLE SalesMqt DETACH PARTITION Jan2012 INTO
SalesMqt_Jan2012;

-- Skip data validation during SET INTEGRITY

SET INTEGRITY FOR SalesMqt ALL IMMEDIATE UNCHECKED;
```

Accessing data after roll-out

To manage data growth, businesses maintain active data in operational tables that typically reside on high-end storage devices. After the data has exceeded its active life span, it is rolled out. In some cases, the rolled-out data still needs to be retained in compliance with corporate policy, government regulations, or business needs. For example, the Sarbanes-Oxley Act sets the policy for corporations to retain certain accounting records. Bank and credit agencies often need to perform analytics or audits on huge amounts of historical data over very long time periods. The historical data might be retained in the same DB2 database or in a different database for future use, and can reside on inexpensive hardware. It is a critical part of data lifecycle management planning to consider the data retention, archiving, and retrieval strategy. Two common solutions for accessing archived historical data are UNION ALL views and the IBM Optim Data Growth Solution.

UNION ALL views

Suppose that the active data resides in a range partitioned table and that the historical data resides in another range partitioned table. A UNION ALL view is created over these two tables so that applications can seamlessly access all of the data. Historical data tends to be read-only and extensive, making it an ideal candidate for compression. The availability of historical data is important, but applications might be able to tolerate longer access times. The enterprise can therefore reduce the TCO by placing historical data on slower, more inexpensive hardware.

Keep the following points in mind when considering a solution using UNION ALL views.

- UNION ALL views have performance limitations when they are defined over a large number of tables, where each table represents a range.
- With UNION ALL views, each leg of the view can come from a different data source in a federated system. This can be advantageous in some environments. In addition, UNION ALL views enable you to define a different set of indexes over active data and historical data.
- UNION ALL views can include a range partitioned table as a leg, as demonstrated in the following example. The range partitioned history table can be in a separate database, with applications transparently accessing it through the view using federated technology.

```
CREATE VIEW SalesAll AS

SELECT * FROM Sales

UNION ALL

SELECT * FROM SalesHistory
```

Optim Data Growth Solution

IBM Optim Data Growth Solution is a leading solution for addressing growth, compliance, and the management of data. It preserves application integrity by archiving busi-

ness objects rather than single tables. For example, it retains foreign key relationships and preserves metadata within the archive.

These features enable you to have flexible access to data and the ability to selectively restore archived data into the original database table, a new table, or even into a different database. For more information, see *"Manage Data Growth"*.

Multi-temperature data warehouses

In a large warehouse, only a portion of the data is frequently accessed. Users expect optimal performance when accessing this data and this data is referred to as "hot" data. The remainder of the data is "cold" data that is rarely accessed or updated, yet needs to be available for regulatory compliance or other business requirements. Using faster, more expensive storage devices for hot data and slower, cheaper storage devices for cold data optimizes the performance of those queries that matter the most, while helping to reduce the overall cost.

A good strategy is to store data in table spaces based on its temperature. As data ages, it becomes less critical, and its temperature changes as well, typically from hot to warm or cold. Table partitioning and multi-temperature storage share the same time-based view of the data, and table partitioning can therefore be used to isolate hot data from cold data. Classify data into two or three tiers of storage. Place the critical data partitions in table spaces whose containers are defined in the hot tier of storage, and place the warm or cold data partitions in table spaces whose containers are defined in the warm or cold tier of storage.

For a large data warehouse, have a warm storage tier for the large amounts of historical data. By applying multi-temperature concepts to your growing data warehouse, you can reduce the total operating cost of your warehouse. Classifying data on the basis of its temperature also enables you to back up hot data more frequently and the relatively static cold data less frequently.

See *"DB2 best practices: Multi-temperature data management"* for information about the following tasks:

- Identifying and characterizing data into temperature tiers
- Designing the database in an IBM Smart Analytics System environment to accommodate multiple data temperatures
- Moving data from one temperature tier to another
- Using DB2 workload management to allocate more resources to requests for hot data than to requests for cold data
- Planning a backup and recovery strategy when a data warehouse includes multiple data temperature tiers

Data maintenance

Managing a large data warehouse is challenging, particularly in a dynamic growing environment. As data is added, updated, and removed, the data characteristics can significantly change. Periodic data maintenance operations are necessary to keep the system lean and high-performing. These maintenance operations consume critical system resources and can impact data availability. This section presents best practices for efficient data maintenance.

Reorganizing data and indexes

Table and index reorganization are necessary maintenance operations that keep a system performing well in the face of continued data growth and modification. Use the REORG command to help cluster the data based on index order, reduce fragmentation, claim empty space and, in some cases, complete ALTER TABLE operations. Use the REORGCHK command to determine whether tables or indexes need to be reorganized or cleaned up.

Consider reorganizing a table or index if any of the following statements applies to your scenario:

- A high volume of insert, update, and delete activity has occurred since the table was last reorganized.
- The performance of queries that use an index with a high cluster ratio has changed significantly.
- Executing the RUNSTATS command to refresh statistical information does not improve performance.
- Output from the REORGCHK command suggests that performance can be improved by reorganizing a table or its indexes.
- Using an MDC table reduces the need of reorganization because the data is guaranteed to be clustered. However, reorganization to reclaim empty space is often needed after MDC roll-out deletion.

Use partition-level REORG to reorganize data and indexes in a range partitioned table. With time-based partitioning, historical data tends to be static, whereas recent data is actively modified. By using the ON DATA PARTITION clause of the REORG TABLE and REORG INDEXES ALL commands to reorganize only the data partitions containing recent data, you will save considerable time and resources compared to reorganizing the entire table.

When a data partition is being reorganized, the remaining data partitions of the table are available for read and write operations if only local indexes exist on the table. If the table has any global indexes, the entire table is offline. Nevertheless, in many cases, a partition-level REORG operation is still preferable to table-level reorganization, because the time required to reorganize one partition and rebuild the global indexes will be significantly less than the time needed to reorganize the entire table.

Run multiple partition-level REORG commands concurrently to significantly reduce down time if multiple partitions need to be reorganized and sufficient CPU and memory resources are available. For example, consider the Sales table, which has grown with the addition of new data partitions. The DBA runs REORGCHK and determines that the last three partitions in 2011 should be reorganized. Checking the system resources and workload in progress, she sees that there are abundant resources to perform the reorganization in parallel, and issues the following commands in three different sessions concurrently:

Session 1:

```
REORG TABLE Sales ALLOW NO ACCESS ON DATA PARTITION Oct2011;
```

Session 2:

```
REORG TABLE Sales ALLOW NO ACCESS ON DATA PARTITION Nov2011;
```

Session 3:

```
REORG TABLE Sales ALLOW NO ACCESS ON DATA PARTITION Dec2011;
```

Drop all global indexes when multiple partitions need to be reorganized to avoid rebuilding the indexes as part of every partition REORG operation. Doing so also enables you to reorganize partitions concurrently if system resources permit. After data partition reorganization is complete, create the global indexes that were dropped.

Runstats

The optimizer determines a query execution plan that is based on data statistics. Up-to-date statistics are necessary for the optimizer to generate optimal plans. As the volume of data grows, the amount of time and resources required to collect statistics increase, thereby impacting workload performance, particularly when tables become very large (hundreds of gigabytes with millions of rows). A DBA must set policies that strike a good balance between limiting system resource utilization and maintaining current statistics.

Use the sampling clause of the RUNSTATS command when collecting table and index statistics for large tables to reduce the I/O and CPU overhead of RUNSTATS processing. Start with a 10% page-level sample by specifying TABLESAMPLE SYSTEM(10). Check the accuracy of the statistics and whether system performance has degraded due to changes in the access plan. If it has degraded, try a 10% row-level sample instead, by specifying TABLESAMPLE BERNOULLI(10). If the accuracy of the statistics is insufficient, increase the sampling amount. Similarly, for indexes, use the SAMPLED DETAILED clause to collect index statistics with nearly the same accuracy but less CPU and memory consumption.

Besides sampling, another technique to reduce the time and resources that are consumed by RUNSTATS on large tables is to collect statistics only on a subset of columns that are frequently used in query predicates. This is accomplished by using the ON COLUMNS clause of the RUNSTATS command.

Statistics must be collected after data roll-in and roll-out operations.

- Issue the RUNSTATS command after data roll-in, for instance, after ATTACH PARTITION + SET INTEGRITY or ADD PARTITION + LOAD.
- After detaching data partitions, issue a RUNSTATS command only after completion of the asynchronous index cleanup process.
- Check the STATUS field in the SYSCAT.DATAPARTITIONS catalog view and ensure that no partitions are in the L (logically detached), I (index cleanup), or D (detached with dependent MQT) states. If there are multiple roll-in or roll-out operations, collect statistics only once after all data lifecycle management operations are complete.

Issue a COMMIT statement immediately after RUNSTATS processing completes to release locks that are held by the RUNSTATS utility.

Parallel LOAD for range partitioned tables speeds up data roll-in performance

If you need to load data into multiple data partitions, performance can be significantly improved by loading the data in parallel. Because the DB2 data server does not support load operations at the data partition level, you can achieve essentially the same result by issuing multiple LOAD commands, one for each range. This approach works if there are only a few global indexes. The following steps outline the high-level approach:

1. Ensure that the data that is to be loaded into each range is in a separate file. This might require modifying the ETL logic to sort the data based on the table partitioning key and to split the data into separate files, one for each data partition.
2. Detach each data partition that is to be loaded. Each detached partition is a stand-alone table.
3. Load the data from different files into the appropriate tables that were created by the previous detach operation.
4. Re-attach the tables to the original range partitioned table.
5. Run a SET INTEGRITY statement against the original table.
6. Execute a RUNSTATS operation to collect statistics on the table.

In the absence of global indexes, this approach can be used for loading data into a single data partition. Otherwise, the availability of all data partitions is impacted by LOAD, even though the data goes into a single data partition.

Database recovery: Backup and restore

As data continues to grow, the time and storage space that is consumed by backup operations can be prohibitive. It might no longer be feasible to use offline backup, because that impacts data availability for longer durations. Perform table space-level backups instead of database-level backups for easier administration of backup and recovery. Classify your data as either active data or historical data and back up the active data more frequently than the historical data.

Table partitioning makes it easy to separate active data from historical data. It enables you to place data on different table spaces, manage smaller backup images, and have the flexibility to define a different backup policy for critical data.

If some tables are "scratch tables" that can be rebuilt as needed, you can avoid backing up table spaces that contain these scratch tables.

ROLLFORWARD TO END OF LOGS operations can also be more granular. That is, you can recover from a disk failure by restoring only the affected table spaces. However, point-in-time rollforward operations must include all table spaces that are related to the table. By carefully designing your backup policy to leverage table space-level backup operations with table partitioning, you can reduce disk space usage, backup time, and the impact on data availability.

Backup tips:

- Back up critical data first and more frequently.
- Perform incremental backup for very large table spaces.
- Use backup compression (by specifying the COMPRESS option on the BACKUP DATABASE command) to reduce the size of your backup images.

Responding to changes

As data continues to grow through consolidation or on-boarding a new application, original design assumptions might no longer be valid. Although small changes to storage allocation or memory management can easily be accommodated, in some cases, you might need to make significant changes. For example, if the table partitioning key has developed a lot of skew, the key might not continue to be the best choice. Perhaps the MDC key is causing the creation of many sparsely filled blocks, resulting in wasted space. In such cases, it can be a good idea to consider whether the defined data organization schemes should be altered or replaced. For example, if you are dealing with thousands of data partitions, and the options for reducing the number of partitions are not attractive, you might consider using MDC instead.

Adding new database partitions to accommodate data growth

To ensure good performance for business applications in an actively growing database system, and to meet service requirements, DBAs need to continuously monitor and understand the database system's capacity indicators and decide when new database partitions should be added. The following guidelines summarize this process.

- Determine the capacity of resources in your environment at the outset so that you can periodically compare used and available capacity.
- Collect data on a regular basis and collate the output both for the current month and for a rolling 12 months. This enables you to compare current performance against your performance baseline and can help identify trends in resource usage.

- Use software (such as IBM InfoSphere Optim Performance Manager or IBM DB2 Performance Expert) and tools (including nmon, vmstat, topas, and sar) to collect statistics on resource usage.
- Understand how resources are used and where resource usage is trending with respect to your performance baseline and performance forecasts in the context of overall capacity. Align service-level objectives with capacity planning indicators so that metrics can easily be compared.

Begin planning for storage expansion when used storage reaches 60% of capacity and is projected to reach 80% within 12 months. This window gives you sufficient time to plan and implement a successful expansion project before storage utilization reaches 100%.

After you purchase and install the physical system and install all of the required software, including DB2 data server, the general process of adding a database partition includes the following steps:

1. Issue the db2start dbpartitionnum <db-partition-number> command for each database partition to be added. This process does not redistribute any data to the new database partitions, and no outage is required.
2. Modify database partition groups to add the new database partitions.
3. Issue the REDISTRIBUTE DATABASE PARTITION GROUP command. Issue the command in offline mode to ensure that the process completes as quickly as possible. In line with best practice recommendations for all system upgrades, complete a full database backup before and after running the REDISTRIBUTE DATABASE PARTITION GROUP command. By completing a full backup, you have a restore point that refers to the original database partitions in the event of hardware failure.
4. Issue the RUNSTATS command to refresh statistics.

Rebalancing data

As data grows over time, the existing distribution keys might no longer be optimal. Significant data skew can lead to degraded query performance. You can change the distribution key for a table by using the SYSPROC.ADMIN_MOVE_TABLE() stored procedure.

Using COUNT() and DBPARTITIONNUM() is a simple method to check whether the table data is reasonably well distributed among all database partitions in the database partition group. For example, the following query returns the data distribution for the SALES table across all database partitions.

```
SELECT DBPARTITIONNUM(TxID), COUNT(*) FROM Sales GROUP BY DBPAR-
TITIONNUM(TxID);
```

This approach is simple but might take a long time to run if the number of rows in some database partitions is very large. An alternate approach is to use the ESTIMATE_EXISTING_DATA_SKEW routine (available from http://www.ibm.com/developerworks/data/library/techarticle/dm-1005partitioningkeys/#download), which provides more user-friendly output, including a list of database partitions, the skew percentage in comparison to the average, and more.

It is recommended that you perform this check during maintenance windows or during off-peak workload hours. The sampling option (10% to 25%, depending on the size of the table) is also recommended. The following example shows how to measure data skew in the Sales table, using a sampling rate of 25%.

```
SET serveroutput ON;


CALL estimate_existing_data_skew('AbcDept', 'Sales', 25);


  Return Status = 0


DATA SKEW ESTIMATION REPORT FOR TABLE: ABC_DEPT.SALES

Accuracy is based on 25% sample of data

---------------------------------------------------------------

ABC_DEPT.SALES

Estimated total number of records in the table: : 4,932,160

Estimated average number of records per partition : 1,233,040


Row count at partition 1 :   986,432 (Skew: -20.00%)

Row count at partition 2 : 1,850,794 (Skew: 50.10%)

…

Number of partitions: 4 (1,2,3,4)
```

If the data shows that the distribution key does not help to evenly distribute the data, consider a new distribution key to replace the old one.

To check for query collocation, collect the queries that characterize the workload and create a workload file (for example, my_new_workload_file) that can be used by the db2advis utility to make recommendations about new distribution keys.

```
db2advis –d <database name> -i <my_new_workload_file> -m P
```

Alternatively, use the following approach to create a report based on the most recently executed queries in the workload if the queries are still available in the DB2 package cache:

```
db2advis –d <database name> -g -m P
```

After you determine the new distribution key, create a table that is like the existing table. Load a small percentage of rows from the existing table into the new table to validate the data distribution based on the new distribution key.

Use the ADMIN_MOVE_TABLE procedure to automatically change the distribution key while keeping the table fully accessible for both read and write operations. In the following example, the distribution key for the Sales table is changed from Tx_Id to (Tx_Id, Prod_Id). The LOAD option is used to improve performance of the ADMIN_MOVE_TABLE procedure.

```
CALL SYSPROC.ADMIN_MOVE_TABLE ('ABC_CO', 'Sales', '', '', '', '',
'TxId, ProdId','', '', 'COPY_USE_LOAD, FORCE', 'MOVE');
```

Tips for coping with data growth and rebalancing:

- Regularly check the distribution of table data across the database partition groups.
- If necessary, rebalance the data by choosing a new distribution key for the table. Use the DB2 Design Advisor (the db2advis utility) for distribution key recommendations. Use the ADMIN_MOVE_TABLE procedure to change the distribution key while keeping the table fully accessible for read and write operations.
- Use the ADMIN_MOVE_TABLE procedure to modify the table partitioning key or the dimension key for a table. This procedure makes a new copy of the table, so you need to ensure that enough storage space exists for the new copy during the move.

Changing table partitioning granularity to accommodate data growth

With time, you might discover that the partitioning granularity of your original design is no longer adequate. You might find that your partitioned table has grown rapidly to hundreds or even thousands of partitions. In such cases, you have a couple of options. You can merge data partitions or you can consider using MDC as an alternative and rely on the ADMIN_MOVE_TABLE procedure to transform the data.

If you decide that your data partitions have become too large and are difficult to manage, you can split the large partitions.

Converting a nonpartitioned table to a range partitioned table

As data volumes grow, it may become unmanageable to keep a large table within a single table space. Based on the design choices described in this paper, you might decide to use table partitioning to break up the table into a number of smaller pieces. This would improve the maintainability of the table and enable you to perform efficient data lifecycle management using ATTACH and DETACH operations. You can use one of the methods below to accomplish this task.

If you are an experienced user who wants to complete the task faster and have more control over the process, follow these steps:

1. Export or unload the data from the current (nonpartitioned) table.
2. Create a new range partitioned table whose definition is the same as the existing non-partitioned table (do not create indexes yet). Place table partitions into different table spaces by taking data temperature and backup or restore granularity requirements into account.
3. Load the data into the newly created partitioned table.
4. Create necessary indexes and constraints; strive to create local indexes for efficient data roll-in and roll-out.
5. Drop the nonpartitioned table.
6. Rename the new range partitioned table.

An alternative that keeps the table online for the duration of the conversion is to use the ADMIN_MOVE_TABLE stored procedure. The stored procedure provides options to complete all operations that are related to the move in a single step or in different steps. The latter enables you to schedule when the source table is taken offline briefly to swap the partitioned table with the nonpartitioned table.

SPLIT PARTITION

During the design phase, you might have chosen the table partitioning strategy to facilitate the even distribution of data across ranges. Suppose that an increase in customer demand for your products has caused holiday sales during the month of December to be orders of magnitude greater than during the other months, resulting in data skew and decreased query performance for this partition. In such a scenario, it might be a good idea to split the large December partition into two partitions by using the following approach:

1. Detach the December partition into DecemberTable and commit.
2. Attach a DecemberFirstHalf partition whose range definition covers the first half of December from DecemberTable.
3. Create an exception table called DecemberSecondHalf.
4. Run SET INTEGRITY with exceptions being written to the DecemberSecondHalf table; at this point, the DecemberFirstHalf partition has data for the first half of December, and the DecemberSecondHalf table has data for the remainder of December.
5. Attach a DecemberSecondHalf partition whose range definition covers the second half of December from the DecemberSecondHalf table.
6. Run SET INTEGRITY.

Indexes need to be maintained as part of the split operation. With local indexes, only indexes on the partition being split need to be maintained. Global indexes make this splitting method more expensive, and depending upon the amount of data involved in the split, you might be better off dropping and recreating the global indexes.

In general, to improve the performance of a split partition operation, choose the range with the least amount of data to be the range that will populate the exception table; this minimizes the amount of data that needs to be copied.

MERGE PARTITION

During the design phase, you might have chosen the table partitioning strategy by taking into account the expected ingest rate. For example, you might have decided to create two data partitions for each month, based on an ingest rate of 10 million records per month. After monitoring the actual ingest rate, you might realize that you could merge the two partitions and reduce the administrative overhead around managing so many table spaces, one per data partition. In such a scenario, it might be a good idea to merge the two partitions by using the following approach:

1. Detach the DecemberFirstHalf partition into the DecemberPart1 table and commit.
2. Detach the DecemberSecondHalf partition into the DecemberPart2 table and commit.
3. Copy data from the smaller of the two tables into the other. You can use either the INSERT statement or the LOAD from cursor command to complete this data movement operation.
4. Attach a December partition whose range definition covers the entire month of December.
5. Run SET INTEGRITY.

With local indexes, the amount of index maintenance is bound by the number of rows in the smaller of the two data partitions that are being merged. Global indexes incur more overhead with this method, because during the detach operation, asynchronous index cleanup will clean up index data for detached partitions, and during the subsequent attach operation, SET INTEGRITY processing will insert index keys for the attached (merged) data partition.

Best practices

- Choose a distribution key to partition data evenly across database partitions.
- Take advantage of collocation for complex join queries.
- When choosing a table partitioning key, select columns that facilitate data roll-out, data roll-in, and partition elimination.
- Create all indexes as local indexes to streamline data roll-in and roll-out (unless they are unique indexes that do not include the table partitioning key).
- Choose dimension key columns for an MDC table to improve query

performance and to ingest data quickly.

- Use generated columns for dimension keys to increase cell density for better space utilization.
- To improve load performance, sort data by the table partitioning key (for range partitioned tables) or the dimension key (for MDC tables) before issuing a LOAD command.
- Use materialized query tables to increase the performance of expensive or frequently used queries that aggregate large amounts of data.
- Use row compression for very large tables, in conjunction with other data organization schemes. It can provide 20-40% space reduction and also improve query performance, because fewer I/O operations are required to access the same amount of data.
- To minimize the impact of periodic ADD PARTITION operations to expand table capacity, add a few partitions at each maintenance window. Avoid adding too many empty partitions.
- Issue a COMMIT statement immediately after all DDL, including ATTACH PARTITION, DETACH PARTITION, RUNSTATS, and SET INTEGRITY.
- Reorganize data or indexes at the table partition level.
- Use the RUNSTATS page sampling option on large tables.
- Monitor data growth rates periodically and plan for storage and capacity expansion in advance.
- Place data partitions in different table spaces to facilitate backup and restore operations.

Conclusion

The best practices presented in this paper are intended to help you to manage scenarios that are characterized by rapid data growth. The paper began with an overview of the DB2 data organization schemes, and described the best database design practices for scalability, manageability, and performance, including the use of these schemes in combination.

This was followed by a discussion of best practices around data lifecycle management, the process by which your system is kept lean to meet performance requirements, and archived data remains available to meet various business needs.

A section on data maintenance included best practices information on reorganizing data and indexes for better clustering, space reclamation, or defragmentation; refreshing statistics to help the optimizer improve data access plans; redistributing the data in partitioned database environments to eliminate skew; and backup and recovery strategies to ensure business continuity after planned or unexpected outages.

A final section provided guidelines that help you to reconsider your choices around data organization schemes by assessing current realities and anticipating future trends.

By leveraging these best practices, you can use DB2 data server's extensible architecture and the layered data partitioning and organization schemes to take full advantage of proven approaches to managing data growth.

Further reading

- Information Management best practices:
  http://www.ibm.com/developerworks/data/bestpractices/

- DB2 for Linux, UNIX, and Windows best practices:
  http://www.ibm.com/developerworks/data/bestpractices/db2luw/

    - *Physical database design for online transaction processing (OLTP) environments* (details on translating logical design to physical design)
    - *Multi-temperature data management* (details on how to move data to new storage repositories across its lifecycle)
    - *Storage optimization with deep compression* (details on controlling growth with row compression)
    - *Ingesting data into an IBM Smart Analytics System* (IBM Smart Analytics System represents the best practice for the implementation and configuration of hardware, firmware, and software for a data warehouse.)
- *Table partitioning* in the IBM DB2 Database for Linux, UNIX, and Windows Information Center
- IBM developerWorks:
    - *Choosing partitioning keys in DB2 Database Partitioning Feature environments*
    - *Unleash the power of table partitioning in your DB2 warehouse*

Contributors

Kevin Beck
Software Developer, Information Management

James Cho
STSM, Architect, Tier 1 Database and Smart Analytics Solutions

Enzo Cialini
STSM, Chief Architect, Quality Assurance, DB2 Distributed and Data Warehousing

Garrett Fitzsimons
Data Warehouse Best Practices Consultant

Jay Lennox
Senior Developer, DB2 Product Development

Paul McInerney
User Experience Professional, Information Management

Christopher Tsounis
Executive I/T Specialist, Software Sales

Angela Yang
DB2 Advanced Technical Support

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

Without limiting the above disclaimers, IBM provides no representations or warranties regarding the accuracy, reliability or serviceability of any information or recommendations provided in this publication, or with respect to any results that may be obtained by the use of the information or observance of any recommendations provided herein. The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any recommendations or techniques herein is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Anyone attempting to adapt these techniques to their own environment does so at their own risk.

This document and the information contained herein may be used solely in connection with the IBM products discussed in this document.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

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.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

Contacting IBM

To provide feedback about this paper, write to db2docs@ca.ibm.com

To contact IBM in your country or region, check the IBM Directory of Worldwide Contacts at http://www.ibm.com/planetwide

To learn more about IBM Information Management products, go to http://www.ibm.com/software/data/