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

Best Practices
Storage Optimization with Deep Compression

Thomas Fanghaenel
DB2 for Linux, UNIX, and Windows Kernel Development

Bill Minor
Information Management Tooling

Issued: April 2012

Table of contents

Executive summary

This document communicates best practices for using the DB2 Storage Optimization Feature with the DB2® for Linux®, UNIX®, and Windows® product. This paper illustrates how this feature can be a crucial driver of a broader space-conscious storage strategy for your large OLTP workload or data warehouse.

You can use the DB2 Storage Optimization Feature to apply compression on various types of persistently stored data and temporary data. The benefits that you can realize from using the best practices that are described in this paper include the following ones:

- Your data consumes less storage space.
- Your storage is consumed at a slower rate.
- Your database performance might improve, depending on your environment.

Introduction

The DB2 for Linux, UNIX, and Windows Version 10.1 product (DB2 Version 10.1) provides various means to help control, manage, and reduce the storage consumption of objects in your database by means of compression.

You can apply compression to primary user data (row and XML data, indexes), system-generated data (temporary tables), and administrative data (backup images and archived transaction logs). Advanced compression features for primary user and system-generated data are available through the DB2 Storage Optimization Feature. Compression facilities for administrative data are available in all editions of the DB2 software.

This document presents the various compression methods in DB2 Version 10.1. For each method, the document lists the points that are worth considering when you determine whether to adopt that method. You are provided with detailed information about how you identify candidate tables and indexes for compression. You are also provided with detailed information about the best practices to help achieve maximum storage space savings when you first adopt compression techniques.

Alternative row format with value compression

You can choose between two row formats for your user-created tables. The row format determines how rows are packed when they are stored on disk. There are standard and alternative row formats, and they can differ significantly in terms of their storage space requirements. The alternative row format allows for more compact storage of NULL and system default values as well as zero-length values in columns with variable-length data types. (The variable-length data types are VARCHAR, VARGRAPHIC, LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB, and XML.) Use of this format is referred to as *NULL and default value compression* or just *value compression*. .

You can specify the row format for each table individually. To create a table that uses the alternative row format, issue the following statement:

```
CREATE TABLE … VALUE COMPRESSION
```

To change the row format that is used in a table, issue one of the following statements:

```
ALTER TABLE … ACTIVATE VALUE COMPRESSION

ALTER TABLE … DEACTIVATE VALUE COMPRESSION
```

If you activate or deactivate value compression for a table, the existing data is not modified. Rows remain in their current row format unless you apply any of the measures listed under "Adoption strategies" later in this document.

In the standard row format, space for fixed-length column values is allocated even if the actual stored value is NULL. Similarly, zero-length values that are stored in columns with variable-length data types still consume a small amount of space. However, in a table that uses the alternative row format, NULL values in all columns and zero-length values in columns with variable-length data types consume no space.

For tables that use the alternative row format, you can save additional space by enabling default value compression for fixed-length numeric and character columns. This results in system default values (0 for numeric columns and blank for fixed-length character columns) not being materialized in the on-disk representation of the row. You must enable compression of system default values for each column individually by specifying the COMPRESS SYSTEM DEFAULT column option for the column in a CREATE TABLE or ALTER TABLE … ALTER COLUMN statement.

When compared to the standard row format, the alternative row format also reduces the storage overhead for all other values in columns with variable-length data types. However, the storage consumption for all non-NULL values that are stored in fixed-length columns increases. Formulas to determine the byte counts for values of all supported data types and row formats are in the reference documentation for the CREATE TABLE statement.

Although the standard row format is a good choice in most cases, there are some tables for which the alternative row format yields a more compact storage layout:

- Sparsely populated tables, that is, tables that contain many rows with many NULL or system default values, should use the alternative row format. However, storage space requirements increase for a row whenever you update a NULL or system default value to a value other than NULL or the system default. This change generally causes overflow records to occur even in tables with no variable-length columns, that is, tables in which all rows have the same space requirements when the standard row format is used.

- Tables having variable-length data types for the vast majority of their columns should use the alternative row format.

For some tables that do not have those characteristics, using the alternative row format can increase storage space requirements. A test in your environment might be worthwhile.

Unlike for some of the advanced compression features, no additional processing overhead is associated with either row format. Hence, you can use value compression even in situations where your workload is largely CPU bound.

You can use the alternative row format regardless of whether you have a license for the DB2 Storage Optimization Feature. This flexibility allows you to choose the most compact storage layout for each table, even if you are not planning on using row compression.

If you are planning on using row compression, choosing the more compact row format to start with yields a smaller on-disk footprint of the table in most cases, though the effect is less dramatic or even negligible. This is due to the fact that row compression can compress the materialized NULL and system default values in tables with the standard row format very well.

Row compression

Row compression was introduced in the DB2 for Linux, UNIX, and Windows Version 9.1 product (DB2 Version 9.1). Significant improvements to the row compression functionality have been made in every release since, culminating in the next-generation adaptive compression functionality in DB2 Version 10.1. Row compression requires a license for the DB2 Storage Optimization Feature.

Storage space savings from compression typically translate into fewer physical I/O operations for reading the data in a compressed table, because the same number of rows is stored on fewer pages. Because compression allows more data rows to be packed into the same number of pages, and buffer pool hit ratios increase. In many cases, I/O savings and improved buffer pool utilization results in higher throughput and faster query execution times.

You can enable row compression individually for each table. Enabling row compression results in space savings for the vast majority of practical tables. Compression ratios are typically 50% - 80% or more. The storage footprint for a table that uses row compression never exceeds that of the uncompressed version of the same table.

Starting in DB2 Version 10.1, there are two types of row compression:

- *Classic row compression* refers to the compression technology that has been used as follows:
  - For user table data since DB2 Version 9.1 was introduced
  - For XML and temporary data since the DB2 for Linux, UNIX, and Windows Version 9.7 product (DB2 Version 9.7) was introduced
- *Adaptive row compression* is a new compression mode that was introduced in DB2 Version 10.1 that you can apply to user table data. Adaptive row compression is superior to classic row compression in that it generally achieves better compression and requires less database maintenance to keep the compression ratio near an optimal level.

Classic row compression

Classic row compression is based on a dictionary-based compression algorithm. There is one compression dictionary for each table object. The dictionary contains a mapping of patterns that frequently occur in rows throughout the whole table. This compression dictionary is referred to as the *table-level compression dictionary*.

For row compression to occur, you must enable the table for compression, and a dictionary must exist for the data or XML object. To enable a table for classic row compression in DB2 Version 10.1 at table creation time, issue the following statement:

```
CREATE TABLE … COMPRESS YES STATIC
```

To enable the compression mode for an existing table, issue the following statement:

```
ALTER TABLE … COMPRESS YES STATIC
```

In DB2 Version 10.1, the STATIC option in the COMPRESS YES clause is mandatory in both cases. In earlier DB2 versions, you use the COMPRESS YES clause without any further qualification, as follows:

```
CREATE TABLE … COMPRESS YES

ALTER TABLE … COMPRESS YES
```

For classic row compression to start taking effect in a table that you enabled for compression, a dictionary must exist for this table. If you use DB2 Version 9.1, you must be conscious of whether compression dictionaries exist and, if necessary, initiate their creation by performing a classic table reorganization or by running the INSPECT utility.

The building process for the table-level dictionary has become more automatic with the introduction of automatic dictionary creation (ADC) in the DB2 for Linux, UNIX, and Windows Version 9.5 product (DB2 Version 9.5). With ADC, whenever a table that you enabled for compression exceeds a size of 2 MB, the next insertion triggers the building of the table-level compression dictionary. However, no compression is applied to existing rows: only subsequent insertions or updates produce compressed rows.

Since DB2 Version 9.5, ADC and classic table reorganization have been the two primary means of building table-level compression dictionaries. The two approaches can differ in terms of the compression ratios that they yield. ADC creates the compression dictionary based on the data that exists at the time that the dictionary is created. If the table grows or changes significantly, only a small subset of the data in that table existed when the dictionary was built. Hence, pattern detection could be performed only on that small subset of data. However, during a classic table reorganization, the entire table is scanned, and an evenly distributed sample of records from the entire table is used to build the dictionary.

As a consequence, dictionaries that are built with ADC might, over time, yield lower storage space savings than those built during a classic table reorganization. Also, over time, the table-level dictionary for a table whose data is frequently updated might no longer contain the most efficient patterns for the changing data, leading to a degradation of the compression ratio In such cases, regular classic table reorganizations may be required to maintain consistently high storage space savings.

Adaptive row compression

Adaptive compression not only yields significantly better compression ratios in many cases but can also adapt to changing data characteristics.

In DB2 Version 10.1, to create a table that has adaptive row compression enabled, issue one of the following statements:

```
CREATE TABLE … COMPRESS YES ADAPTIVE

CREATE TABLE … COMPRESS YES
```

To enable adaptive row compression on an existing table, issue one of the following statements:

```
ALTER TABLE … COMPRESS YES ADAPTIVE

ALTER TABLE … COMPRESS YES
```

In DB2 Version 10.1, the default type of row compression is adaptive compression. Therefore, the ADAPTIVE option is the default value for the COMPRESS YES clause.

When you upgrade a database from the DB2 for Linux, UNIX, and Windows Version 9.8 product (DB2 Version 9.8) or an earlier release, existing tables that had classic row

compression enabled keep their compression settings. If you want to enable the tables for adaptive row compression, you must use one of the ALTER TABLE statements shown earlier.

Adaptive compression builds on classic row compression, and table-level compression dictionaries are still used. The table-level dictionary is complemented by *page-level compression dictionaries*, which contain entries for frequently occurring patterns within a single page. The table-level dictionary helps eliminate repeating patterns in a global scope, while page-level dictionaries take care of locally repeating patterns.

Page-level dictionaries are maintained automatically. When a page becomes filled with data, the database manager builds a page-level compression dictionary for the data in that page. Over time, the database manager automatically determines when to rebuild the dictionary for pages where data patterns have changed significantly.

The result of using adaptive compression is not only higher overall compression savings. Adaptive compression also ensures that compression ratios do not degenerate as much over time as with classic row compression. In many practical cases, the compression ratio remains nearly optimal over time. Thus, by using adaptive compression, you can reduce the cost that is associated with monitoring compression ratios of tables and performing maintenance (classic, offline table reorganization) on tables to improve storage utilization.

Performing a classic table reorganization still gives the best possible compression. However, the improvements that a classic table reorganization can provide for tables with adaptive row compression are less significant and often even negligible.

Revealing compression settings

To determine whether a table has row compression enabled and which row format it uses, query the COMPRESSION column in the SYSCAT.TABLES view. The possible values are as follows:

V. The table uses the alternative row format and no row compression.

R. The table uses row compression and the standard row format.

B. The table uses the alternative row format and row compression.

N. The table does not use row compression but uses the standard row format.

In DB2 Version 10.1, to determine which of the row compression types is used on your tables, query the ROWCOMPMODE column in the SYSCAT.TABLES view. The possible values are as follows:

S. The table uses classic row compression.

A. The table uses adaptive row compression.

Blank. The table is not enabled for row compression.

In DB2 Version 9.8 and earlier, the ROWCOMPMODE column does not exist in the SYSCAT.TABLES view. All tables that have row compression enabled implicitly use classic row compression.

The following query shows the compression settings for all user tables in DB2 Version 10.1:

```
SELECT SUBSTR(TABSCHEMA, 1, 10) AS TABSCHEMA,
       SUBSTR(TABNAME, 1, 10) AS TABNAME,
       COMPRESSION, ROWCOMPMODE
FROM SYSCAT.TABLES
WHERE TABSCHEMA NOT LIKE 'SYS%'
```

Sample results are as follows:

```
TABSCHEMA  TABNAME    COMPRESSION ROWCOMPMODE
---------- ---------- ----------- -----------
DB2INST1   ACCTCR     R           S
DB2INST1   BKPF       R           A
DB2INST1   BSIS       B           A
DB2INST1   CDCLS      N
DB2INST1   CDHDR      V
DB2INST1   COSP       B           S

  6 record(s) selected.
```

In this example, the ACCTCR, BKPF, BSIS, and COSP tables are all enabled for row compression. The ACCTCR and COSP tables use classic row compression, and the BKPF and BSIS tables use adaptive row compression. The BSIS and COSP tables also use the alternative row format, whereas the ACCTCR and BKPF tables use the standard row format. The CDHDR table uses the alternative row format without row compression, and the CDCLS table uses the standard row format without row compression.

Identifying candidate tables for row compression

If you are not using row compression, you might want to examine your database to determine which tables are candidates for compression. Data compression helps you initially save storage on existing uncompressed tables and optimize storage growth in the future. You can find your storage “pain points” in existing tables that contain a significant amount of data or in tables for which you anticipate substantial growth over time.

Naturally, the largest tables are obvious candidates for compression, but do not overlook smaller tables. If you have hundreds or thousands of smaller tables, you can benefit from the aggregate effect of compression over many smaller tables. “Large” and “small” are relative terms: your database design determines whether tables of a million or several million rows are “large” or “small.”

The following example uses the SQL administration function ADMIN_GET_TAB_INFO to return an ordered list of all your table names and table data object sizes for a particular

schema. To display the current compression settings and the row compression mode that are in use, join the result set with the SYSCAT.TABLES view, and select the COMPRESSION and ROWCOMPMODE columns.

```
SELECT SUBSTR(T.TABSCHEMA, 1, 10) AS TABSCHEMA,
       SUBSTR(T.TABNAME, 1, 10) AS TABNAME,
       SUM(TI.DATA_OBJECT_P_SIZE)/1024/1024 AS STORAGESIZE_GB,
       T.COMPRESSION AS COMPRESSION,
       T.ROWCOMPMODE AS ROWCOMPMODE
FROM TABLE (SYSPROC.ADMIN_GET_TAB_INFO('DB2INST1', '')) TI
       JOIN SYSCAT.TABLES T ON T.TABSCHEMA = TI.TABSCHEMA AND
                               T.TABNAME = TI.TABNAME
GROUP BY T.TABSCHEMA, T.TABNAME, T.COMPRESSION, T.ROWCOMPMODE
ORDER BY STORAGESIZE_GB DESC
```

This query identifies the top space consumers in your current database, and gives you a list of candidate tables that you should start working with. In the example, all the tables are currently uncompressed, and all except the BSIS and CDHDR tables use the standard row format.

```
TABSCHEMA  TABNAME    STORAGESIZE_GB COMPRESSION ROWCOMPMODE
---------- ---------- -------------- ----------- -----------
DB2INST1   ACCTCR                 14 N
DB2INST1   BKPF                   12 N
DB2INST1   BSIS                   12 V
DB2INST1   CDCLS                  10 N
DB2INST1   CDHDR                   9 V
DB2INST1   COSP                    7 N

  6 record(s) selected.
```

In many practical scenarios, the bulk of the storage space of a database is occupied by relatively few tables.

Small tables under a few hundred KB might not be good candidates for classic row compression because the space savings might not offset the storage requirements of the data compression dictionary (approximately 100 KB). The dictionary is stored within the physical table data object. Generally, consider using classic row compression for tables that you expect to grow to 2 MB and larger. With adaptive compression, no such considerations are necessary. Page-level dictionaries are built even if the table-level dictionary does not exist. Space savings are achieved from the very beginning, even for very small tables.

After you determine candidate tables based on their storage consumption, consider the typical SQL activity against the data in those tables:

- Tables that are read only are excellent candidates for row compression. Tables with a read/write ratio of 70% or more reads and 30% or less writes are good candidates for row compression.
- Tables that experience only a limited number of updates are likely to be good candidates for row compression.

- Tables that undergo heavy updates might not be as good candidates for row compression.
- Large tables that are accessed mostly through table scans rather than index scans are good candidates for compression. This generally includes large fact tables in data warehouses, where a significant number of queries perform heavy aggregations. The I/O savings and the increased buffer pool utilization that result from applying compression are likely to improve query performance for such tables.

Whether it is better to use classic or adaptive row compression depends less on the data access patterns than on the actual compression savings that you can achieve by using either type of compression. You make the decision about which row compression mode to choose later in the process.

Row compression performs best in I/O- or memory-bound environments, where the workload is not bottlenecked on the CPU. Extra CPU cycles are required to perform row compression and expansion of data rows whenever they are accessed or modified. This overhead can be offset by efficiencies that are gained in doing fewer I/O operations. Row compression works very well with decision-support workloads that are composed of complex analytic queries that perform massive aggregations, where row access is mostly sequential and less random.

Some of the side effects of enabling compression can only be fully observed after the changes are made. This may include the effects of I/O savings and additional CPU overhead. Consider using a test environment to benchmark or assess performance when compression is in effect in your environment, before making the changes in your production environment.

Side effects of enabling row compression

Following the implementation of row compression, the amount of user data that is written to log records as a result of insert, update, and delete activity is generally smaller. However, some update log records can be larger when compressed than when not compressed. The larger log records are due to the fact that when a compressed row is updated, the compressed row image can change, in both size and content, even if only a small portion of the row is updated.

Besides compression, there are other factors that affect log space consumption for update operations. These factors include whether the updated table is enabled for replication (through DATA CAPTURE CHANGES), whether queries use currently committed semantics, and where the updated columns are positioned in the table. The DB2 Version 10.1 Information Center has more information on column ordering and its effects on update logging.

Starting in DB2 Version 10.1, you can reduce the amount of space that is taken by log archives by compressing them. This feature is independent of whether row compression is enabled, and it does not affect the amount of data that must be logged. Log archive compression is described later in this document.

Estimating storage space savings for row compression

After you determine the list of candidate tables for compression, based on storage consumption and data access characteristics, it is time to determine the storage space savings that you can expect to achieve. You can estimate the space savings for each of the candidate tables before you enable the tables for row compression. You can estimate storage space savings even if you do not have a license for the DB2 Storage Optimization Feature.

As with the compression functionality in general, the mechanisms for compression estimation have evolved over time, with the goal of providing quicker and simpler ways to perform this task. The mechanism of choice depends on which version of the DB2 software you are using. All the tools and functions that are available in older releases are also available in DB2 Version 10.1. However, you might find that the improved counterparts in DB2 Version 10.1 are easier to use or faster to respond with the data that you need.

Estimating row compression savings in DB2 Version 10.1

In DB2 Version 10.1, the preferred way to estimate compression ratios is to use the ADMIN_GET_TAB_COMPRESS_INFO administrative function. You can use this function to estimate compression savings for a particular table, for all tables in a particular schema, or for all tables in a database.

The function calculates current compression savings and savings projections for classic and adaptive compression. The following example returns the values for all tables in schema DB2INST1:

```
SELECT SUBSTR(TABNAME,1,10) AS TABNAME,
       PCTPAGESSAVED_CURRENT,
       PCTPAGESSAVED_STATIC,
       PCTPAGESSAVED_ADAPTIVE
FROM TABLE(SYSPROC.ADMIN_GET_TAB_COMPRESS_INFO('DB2INST1', ''))
```

The table and schema names are optional parameters and can have empty or NULL values. The function computes the estimates for all tables in a particular schema if you do not specify the table name or all tables in the database if you do not specify both the table and schema names. The resulting columns and estimates are as follows:

- **PCTPAGESSAVED_CURRENT:** Current compression savings.
- **PCTPAGESSAVED_STATIC:** Optimal compression savings that you can achieve with classic row compression.
- **PCTPAGESSAVED_ADAPTIVE:** Optimal compression savings that you can achieve with adaptive row compression.

If your schema or database contains hundreds or thousands of tables, the processing time might be substantial. If this is the case, restrict your queries to compute only estimates for those tables for which you are considering compression.

You can use the result of the previous query to determine whether to enable row compression and which row compression mode to choose. For example, consider the following result set of the previous query:

```
TABNAME    PCTPAGESSAVED_CURRENT PCTPAGESSAVED_STATIC PCTPAGESSAVED_ADAPTIVE
---------- --------------------- -------------------- ----------------------
ACCTCR                         0                   68                     72
BKPF                           0                   83                     90
BSIS                           0                   82                     90
CDCLS                          0                   11                     17
CDHDR                          0                   70                     73
COSP                           0                   87                     91

  6 record(s) selected.
```

Of the six tables for which the compression estimates were computed, five show very good compression potential. Only the CDCLS table compresses rather poorly. The compression savings might not justify enabling compression on this particular table.

Calculating the differences between the values of the PCTPAGESSAVED_STATIC and PCTPAGESSAVED_ADAPTIVE columns helps you to determine the best choice for the row compression mode. However, be aware that the scale of the value of the PCTPAGESSAVED_CURRENT column is not linear. For example, consider the ACCTCR and COSP tables. For the ACCTCR table, you can expect 68% space savings with classic row compression and 72% space savings with adaptive row compression. For the COSP table, the estimated savings are 87% with classic row compression and 91% with adaptive row compression. Although the absolute difference between both compression types is 4% for either table, the relative savings that adaptive row compression can achieve over classic row compression are different. Assume that each of the tables is 100 GB in size when compression is not enabled. For the ACCTCR table, the estimated sizes are 32 GB with classic row compression and 28 GB with adaptive row compression, which constitutes a difference of approximately 12.5%. However, the estimated sizes for the COSP table are 13 GB with classic row compression and 9 GB with adaptive row compression, which is a difference of approximately 30%.

In the previous example, the COSP table is a good candidate for adaptive row compression, but the ACCTCR table might not be a good candidate. Also, the estimated values indicate that adaptive compression would reduce the storage size for table CDHDR by only another 10% over classic row compression. Consequently, you might not want to enable adaptive compression for the CDHDR and ACCTCR tables unless you expect the data characteristics to change significantly or you expect lots of new data to be inserted. For the rest of the tables, adaptive compression might be the better choice because of the significant additional storage space savings of up to 45%.

The values in the PCTPAGESSAVED_CURRENT column indicate how well your current data in a particular table compresses, assuming that you enabled compression for that table. These measurements are similar to what the RUNSTATS command would calculate for the PCTPAGESSAVED column in the SYSCAT.TABLES view, except that the measurements in the PCTPAGESSAVED_CURRENT column are always up to date.

Therefore, you do not have to issue the RUNSTATS command to get an accurate picture of what you are currently saving. The values in the PCTPAGESSAVED_CURRENT column can help you decide what tables to switch from classic to adaptive row compression when you upgrade your database to DB2 Version 10.1. These values can also help monitor the compression ratio over time.

Consider the following example, which shows a situation that you might face upon upgrading to DB2 Version 10.1 if you use row compression in earlier DB2 releases. In this case, all the tables are enabled for classic row compression, and the previous query might return a result as follows:

```
TABNAME    PCTPAGESSAVED_CURRENT PCTPAGESSAVED_STATIC PCTPAGESSAVED_ADAPTIVE
---------- --------------------- -------------------- ----------------------
ACCTCR                        67                   68                     72
BKPF                          78                   83                     90
BSIS                          80                   82                     90
CDCLS                          0                   11                     17
CDHDR                         69                   70                     73
COSP                          87                   87                     91

  6 record(s) selected.
```

All tables except for the CDCLS table are already enabled for row compression. Some of the tables (such as BKPF, BSIS, and CDHDR) show that the compression ratio can be significantly improved, even by merely performing a classic table reorganization. But you might want to strongly consider enabling adaptive row compression on the BKPF, BSIS, and COSP tables, because they would benefit significantly from the better compression capabilities. You might also want enable the ACCTCR and COSP tables for adaptive row compression, although on a somewhat lower priority, because the storage space savings to be gained on those tables are lower than on the others.

The processing time of the ADMIN_GET_TAB_COMPRESS_INFO table function in DB2 Version 10.1 is significantly lower than the processing times of similar administrative functions in earlier DB2 versions. The database manager now performs scan sampling, which drastically reduces the amount of data that must be read before an accurate prediction can be made. You will experience the biggest improvements with very large tables.

Estimating row compression savings in DB2 Version 9.5 and Version 9.7

In DB2 Version 9.5 and 9.7, administrative functions that are similar to the ADMIN_GET_TAB_COMPRESS_INFO function in DB2 Version 10.1 are available, though they have slightly different names and signatures. In DB2 Version 9.7, the function is called ADMIN_GET_TAB_COMPRESS_INFO_V97, and in DB2 Version 9.5, its name is ADMIN_GET_TAB_COMPRESS_INFO. Unlike the function in DB2 Version 10.1 that takes two input parameters, the DB2 Version 9.5 and Version 9.7 functions take three. The third parameter is the execution mode, and for compression estimation, you pass the string ESTIMATE as a value for this parameter. The result sets of the DB2

Version 9.5 and Version 9.7 functions also differ significantly from the result set of the DB2 Version 10.1 function.

For the same set of tables as in the previous example, you perform compression estimation in DB2 Version 9.5 as follows:

```
SELECT SUBSTR(TABNAME,1,10) AS TABNAME,
       PAGES_SAVED_PERCENT
FROM TABLE(SYSPROC.ADMIN_GET_TAB_COMPRESS_INFO('DB2INST1',
                                               '',
                                               'ESTIMATE'))

TABNAME    PAGES_SAVED_PERCENT
---------- -------------------
ACCTCR                      68
BKPF                        83
BSIS                        82
CDCLS                       11
CDHDR                       70
COSP                        87

  6 record(s) selected.
```

The result set columns are different than in DB2 Version 10.1 because DB2 Version 9.5 and Version 9.7 support only classic row compression. Furthermore, the processing time for the administrative functions might be significantly longer than in DB2 Version 10.1 because a full table scan is performed for each table to compute the compression estimates.

As with the DB2 Version 10.1 administrative function, the Version 9.5 and 9.7 administrative functions can help you decide whether to enable row compression and on which of your candidate tables to enable compression. You can also use those administrative functions to monitor the changes in compression ratios over time, to help you decide when a classic table reorganization might be necessary to improve compression. In order to determine whether a classic table reorganization may be beneficial, do not compare the results of using the REPORT and the ESTIMATE modes. Using the REPORT mode does not return the current compression savings. Instead, it gives information about the compression savings at the time when the dictionary was built, and which might be long outdated. Instead of comparing the results of using the modes, first make sure that your statistics are current. Next, compare the PERCENTAGE_PAGES_SAVED value that is returned by the administrative function in ESTIMATE mode with the value of the PCTPAGESSAVED column in the SYSCAT.TABLES view for the same table. If the latter is significantly lower than the estimated compression savings, this can indicate that you should reorganize the table and have a new table-level dictionary built by that process.

Estimating row compression savings in DB2 Version 9.1

DB2 Version 9.1, the first DB2 release to support row compression, does not provide any of the administrative functions that were introduced in later releases. However, you can

perform compression estimation by running the INSPECT utility with the ROWCOMPESTIMATE parameter, as shown in the following example:

```
INSPECT ROWCOMPESTIMATE
  TABLE NAME ACCTCR
  SCHEMA DB2INST1
  RESULTS acctcr.rowcompestimate.out
```

Running this utility produces an INSPECT utility output file in the diagnostic data directory path. You must format this file by using the **db2inspf** command line tool, as shown in the following example:

```
db2inspf acctcr.rowcompestimate.out acctcr.rowcompestimate.txt
```

The formatted output file contains information about the estimated compression savings in text form, as shown in the following example:

```
DATABASE: TDB2
VERSION : SQL10010
2011-11-07-18.39.23.555355


Action: ROWCOMPESTIMATE TABLE
Schema name: DB2INST1
Table name: ACCTCR
Tablespace ID: 4  Object ID: 8
Result file name: acctcr.rowcompestimate.out

    Table phase start [...]

      Data phase start. Object: 8  Tablespace: 4
      Row compression estimate results:
      Percentage of pages saved from compression: 68
      Percentage of bytes saved from compression: 68
      Compression dictionary size: 32640 bytes.
      Expansion dictionary size: 32768 bytes.
      Data phase end.
    Table phase end.
Processing has completed. 2011-11-07-18.40.08.349345
```

The INSPECT utility also inserts a table-level dictionary into the target table if that table is enabled for row compression and a table-level dictionary does not exist. This utility therefore provides a way to build a compression dictionary in DB2 Version 9.1 without performing a classic table reorganization. In later DB2 releases, you should not use the INSPECT utility to build a table-level dictionary but should rely on ADC instead.

Index compression

You can apply compression to indexes if you are using DB2 Version 9.7 or a later release and have a license for the DB2 Storage Optimization Feature. Compressing indexes

results in fewer leaf pages, which in many cases helps reduce the depth of the index tree. This allows for fewer index page accesses to find a specific key and like in case of row compression, allows for a higher utilization of the buffer pool and fewer physical I/O operations. In many practical cases, index compression can significantly improve query performance.

You can enable index compression for each index individually. When you create an index, the index inherits its compression setting from the underlying table. That is, all indexes that you create on tables that are enabled for classic or adaptive row compression are compressed as well. You might want to verify that your database build scripts take this dependency into account. You can either create the tables with a COMPRESS YES clause or alter the tables to enable row compression before creating any indexes on them.

Index compression uses a combination of three techniques to reduce the amount of data that is stored on disk. Space savings are achieved by dynamically adjusting the number of index keys rather than by reserving space for the highest possible number of keys that can be stored in a page. The index entries, each of which consists of an index key and a RID list, are compressed by eliminating redundant prefixes from the keys in a single page. For duplicate keys, the associated RIDs in each list are compressed by applying a delta encoding.

Like adaptive row compression and unlike classic row compression, index compression is fully automatic. Optimal compression savings are maintained over time, and there is no need to monitor and potentially reorganize indexes to improve compression ratios.

Identifying candidate indexes for index compression

The best candidates for compression are indexes with long keys (for example, multicolumn indexes or indexes on character data) and indexes that contain many duplicate keys (for example, indexes that are based on a single or very few low-cardinality columns). In addition, indexes that involve variable-length columns (for example, VARCHAR columns) might benefit from the better space utilization that the dynamic index page space management offers.

Multicolumn indexes generally compress better if the leading key columns have low to medium cardinality and the higher-cardinality or unique columns are toward the end of the key column list. Depending on the characteristics of the queries that you expect your indexes to support, column reordering might be possible without sacrificing query performance. It might be worthwhile checking the order and cardinalities for the columns of your larger indexes, especially in data warehousing scenarios. In these cases, the majority of queries aggregate values over a larger number of qualifying rows and query characteristics are generally more predictable than in OLTP workloads.

Examples of indexes that might benefit less from compression are indexes on a single unique numeric column or unique multicolumn indexes that have the highest-cardinality column as the leading index column.

Estimating storage space savings for index compression

When you are deciding which indexes to compress, estimation of storage space savings plays a bigger role than it does for tables. The computational overhead that is associated with index compression is less than for row compression, and storage space savings generally correspond more directly to overall performance improvements. If you can achieve significant space savings for any of your indexes in an OLTP workload, it is a good idea to compress them.

Similar to the row compression estimation function, there is an administrative function to project storage space savings for index compression. This function is called ADMIN_GET_INDEX_COMPRESS_INFO. You can use it to estimate compression savings for a single index, for all indexes on a particular table, for all indexes in a particular schema, or for all indexes in a database. You control the execution mode of the function by the first three of its five input parameters, which are the object type, schema, and name. If you want to estimate space savings for a single index or for all indexes in a particular schema, specify an object type of 'I'. If you want to estimate space savings for all indexes on a particular table or all tables in a schema or database, specify an object type of 'T', NULL, or the empty string. Depending on the value of the object type parameter, you can use the object name and schema parameters to identify the table or index that you want. Alternatively, you can leave the values of the object name and schema parameters as NULL or empty strings if you want to estimate compression savings for a range of indexes.

For example, you would use the following query to obtain estimates for compression savings for all indexes on the table BKPF on all partitions:

```
SELECT SUBSTR(INDNAME, 1, 20) AS INDNAME,
       COMPRESS_ATTR,
       PCT_PAGES_SAVED
FROM TABLE(SYSPROC.ADMIN_GET_INDEX_COMPRESS_INFO('T',
                                                 'DB2INST1',
                                                 'BKPF',
                                                 NULL,
                                                 NULL))
```

A sample result set of this query is as follows:

```
INDNAME             COMPRESS_ATTR PCT_PAGES_SAVED
------------------- ------------- ---------------
BKPF~0              N                          46
BKPF~1              N                          57
BKPF~2              N                          71
BKPF~3              N                          71
BKPF~4              N                          45
BKPF~5              N                          71
BKPF~6              N                          70
BKPF~BUT            N                          71

  8 record(s) selected.
```

It shows that most of the indexes on this table compress very well, with up to 70% space savings to be gained.

If you invoke the ADMIN_GET_INDEX_COMPRESS_INFO function on indexes that are already compressed, the compression savings reflect the actual savings rather than estimates. Alternatively, you can obtain information about the current savings that are generated by index compression from the PCTPAGESSAVED column of the SYSCAT.TABLES catalog view. The RUNSTATS command maintains the value of the column.

Compression of temporary tables

Starting in DB2 Version 9.7, if you have a license for the DB2 Storage Optimization Feature, you can apply compression to temporary tables. Unlike with row and index compression, you do not have to enable temporary tables for compression. Compression is done automatically and applies to both user-defined and system temporary tables.

Temporary tables are used in different situations. There are user-defined global temporary tables, which come in two variants: created global temporary tables (CGTTs) and declared global temporary tables (DGTTs). Temporary tables are also used by some utilities and maintenance operations, such as table reorganization and data redistribution. During query processing, it might be necessary for the database manager to use temporary tables for certain operations that must accumulate intermediate results, such as sorts, hash joins, or table queues.

The mechanism that is used to compress temporary tables is the same as the one that is used for classic row compression with ADC, though the runtime behavior is slightly different than for permanent tables. Most temporary tables, especially small ones, do not cause any physical I/O. Thus, the threshold at which the compression dictionary is built is 100 MB instead of 2 MB. The higher threshold ensures that small temporary tables that typically remain fully buffered are not compressed but larger temporary tables that might spill to disk have compressed data. In addition to avoiding physical I/O, compression for temporary tables ensures that large temporary tables use the buffer pool more efficiently, which helps further in avoiding physical I/O.

Adoption strategies

After you determine the set of tables and indexes that you are going to compress, the next step is to enable these tables for compression. Changing the compression settings for existing tables with data in them helps slow their growth rate, and data that is being inserted benefits from the new compression settings. If your goal is to reduce the footprint of your existing data, perhaps because you want to reduce the physical space that is allocated by your database, you must apply compression to this data.

You can use the following strategies to help apply compression to large amounts of existing data, free up disk space that is currently consumed by that data, and release this space to the file system. You will find this information most useful when you first implement row or index compression or when you are upgrading to DB2 Version 10.1 from earlier DB2 releases.

Applying compression to existing data

The most straightforward way to apply compression to existing data is by performing a classic table reorganization. If you changed the settings for row or value compression, use the REORG TABLE command with the RESETDICTIONARY parameter. If some of your tables have XML columns, you should specify the LONGLOBDATA clause as well to compress the non-inlined XML documents. If you changed the compression settings only for indexes, the REORG INDEXES ALL command is sufficient to compress the index data. A full reorganization of the table data is not necessary. You might consider performing an index reorganization rather than a full reorganization of all table data if, for example, you are upgrading from DB2 Version 9.5 to DB2 Version 10.1 and already have row compression enabled.

If you cannot take your tables offline, consider using the ADMIN_MOVE_TABLE stored procedure, which has been available since DB2 Version 9.7. You can use this procedure to move data that is in an active table into a new data table object with the same name and the same or a different storage location. The ADMIN_MOVE_TABLE procedure provides a way to effectively reorganize a table in the same way that a classic table reorganization would, but without causing any downtime. The data remains online for the duration of the move. The table-level dictionary is built or rebuilt with the same quality as during reorganization. To build or rebuild the dictionary, the ADMIN_MOVE_TABLE procedure performs the following steps:

1. Collect a Bernoulli sample of all the rows in the table
2. Build a new table-level compression dictionary from that sample
3. Insert the dictionary into the target table before starting the copy phase, so that rows are compressed when they are inserted into the target table during the copy phase

The following example shows how you initiate an online table move operation in the most straightforward way:

```
CALL
SYSPROC.ADMIN_MOVE_TABLE('DB2INST1','ACCTCR',
                         '','','','','','','','','MOVE')
```

You can use the ADMIN_MOVE_TABLE_UTIL procedure to exercise fine-grained control over parameters and the behavior of the online table move operation, such as determining the size of the sample that is used for dictionary building. However, even if you invoke the function in the standard 'MOVE' execution mode, the compression dictionary that is built is optimal.

Unlike for classic table reorganization, the default behavior for the ADMIN_MOVE_TABLE procedure is to rebuild the dictionary. Because only a small random sample of the table data is used, rebuilding the dictionary typically does not cause an excessive performance overhead compared to not rebuilding the dictionary.

You might find the ADMIN_MOVE_TABLE procedure particularly useful when you upgrade your database from an earlier DB2 release. You can use the procedure to move tables between table spaces and change other parameters, such as column data types, at the same time.

Managing disk space consumption

Applying compression to existing tables reduces the number of pages that are allocated for those tables. However, applying compression does not immediately affect the disk space consumption of your DMS or automatic storage table spaces. Those table spaces merely have a larger number of unused extents.

When you first implement row and index compression, it is a good practice to reevaluate the utilization of your table space containers and the projected growth rate of your data. With compression, your table space utilization can be significantly reduced. It might take a long time until your database grows enough to consume all the space that you initially freed up. In that case, you might want to reduce the container sizes and make more disk space available to other consumers.

Effectively reducing the on-disk footprint of your database requires that you reduce the size of your table space containers. The key factor that determines how much you can reduce the size of your table space containers is the *high water mark*. The high water mark reflects the location of the highest extent in a table space that is allocated for a table or used as a space map entry.

The high water mark can increase when you create tables, extend existing tables, or perform a classic table reorganization that does not use a temporary table space. You can reduce the high water mark by dropping or truncating the table that owns the extent at the high water mark. However, dropping, truncating, or reorganizing other objects does not affect the high water mark, and merely results in creating more unused extents below the high water mark. Those unused extents below the high water mark can be reused only when you create new objects or when objects grow. Only unused extents above the high water mark can be reclaimed and returned to the operating system.

Consequently, the key to reducing the on-disk footprint is to lower the high water mark and truncate the table space containers. The remainder of this section describes the mechanisms for performing this task. Those mechanisms depend on what kind of table spaces you are using, which DB2 release you used to create these table spaces, and which DB2 release you are currently using.

Reclaimable storage in DB2 Version 9.7 and later

Non-temporary DMS and automatic storage table spaces that you create in DB2 Version 9.7 or later support reclaimable storage. For those table spaces, the database manager can change the physical location of extents in the containers. This mechanism is called *extent movement* and can be used to consolidate all the unused space in table space containers to the physical ends of the containers. After that consolidation takes place, the high water mark can be lowered, and the table space containers can subsequently be shrunk to the smallest possible size.

You can use the ALTER TABLESPACE statement to lower the high water mark and shrink table space containers. The process of lowering the high water mark involves extent movement and can thus take some time. After the high water mark is lowered, containers can be reduced in size by returning unused extents above the high water mark to the file system.

To determine how much the size of a DMS or automatic storage table space can be reduced, use the MON_GET_TABLESPACE table function, as shown in the following example:

```
SELECT SUBSTR(TBSP_NAME, 1, 15) AS TBSP_NAME,
       TBSP_FREE_PAGES,
       TBSP_PAGE_SIZE * TBSP_FREE_PAGES / 1024 /1024
         AS TBSP_RECLAIMABLE_MB
FROM TABLE(MON_GET_TABLESPACE('TBSP_USR_D_1', NULL))
```

The following sample result set indicates that the table space can be reduced by about 23 million pages, worth 365 MB in the on-disk footprint:

```
TBSP_NAME       TBSP_FREE_PAGES      TBSP_RECLAIMABLE_MB
--------------- -------------------- --------------------
TBSP_USR_D_1                23369760               365150

  1 record(s) selected.
```

You can easily shrink automatic storage table spaces to their most compact format by using the ALTER TABLESPACE statement with the REDUCE MAX clause, as shown in the following example:

```
ALTER TABLESPACE TBSP_USR_D_1 REDUCE MAX
```

For DMS table spaces, lowering the high water mark and reducing containers is a two-step process, as shown in the following two sample statements:

```
ALTER TABLESPACE TS_DAT_D LOWER HIGH WATER MARK

ALTER TABLESPACE TS_DAT_D REDUCE (ALL CONTAINERS 100 G)
```

When you upgrade your database from an older DB2 release to DB2 Version 9.7 or later, consider taking the following steps to get the benefits of reclaimable storage:

1. Create new table spaces for all your automatic storage or DMS table spaces.
2. Enable row and index compression on the tables.
3. Use the ADMIN_MOVE_TABLE procedure to move your tables from the old to the new table spaces.

You should enable row and index compression before moving the tables, so that they get compressed in the process of moving.

Reducing container sizes in DB2 Version 9.5 and earlier

For DMS and automatic storage table spaces that you create with DB2 Version 9.5 and earlier, extent movement is not supported. Therefore, the highest used extent will always hold up the high water mark. You cannot reclaim the space that is allocated for unused extents below the highest used extent, unless you manually help rearrange your data so this highest used extent is freed up.

You can use the **db2dart** utility with the **/DHWM** option to determine what kind of extent is holding up the high water mark and whether you can lower the high water mark. If you can lower the high water mark, you can list the necessary steps by using the **db2dart** utility with the **/LHWM** option.

If a table data object or an XML object is holding up the high water mark, performing a classic table reorganization might help lower it. If an index object is holding up the high water mark, reorganizing the indexes for the related table might help lower it. If the high water mark is held up by an empty SMP extent, use the ALTER TABLESPACE … REDUCE statement in DB2 Version 9.5 and later or the **db2dart** utility with the **/RHWM** option to discard this SMP extent.

Before you use the **db2dart** utility to lower the high water mark, you must deactivate your database, and the affected table space might be placed in backup-pending state, as described by Technote 1006526.

If you cannot upgrade to a more recent release than DB2 Version 9.5, consider using a temporary table space when you perform the classic table reorganizations to apply compression to your tables. This helps avoid the situation where the reorganization requires additional extents to be used, beyond the current high water mark. If possible, put the largest tables into a separate table space before attempting to compress them.

If you have many tables in the same DMS or automatic storage table space, perform the classic table reorganizations starting with the smallest tables first. This ensures that the high water mark grows as little as possible and that any unused extents that are generated below the high water mark are used when the next largest table is reorganized.

Verifying row compression and avoiding common pitfalls

You can reveal the compression settings for tables by querying the system catalog. In the SYSCAT.TABLES view, the two columns COMPRESSION and ROWCOMPMODE contain information about the current compression settings. A few other interesting columns in the SYSCAT.TABLES view provide deeper insight into how row compression performs. The RUNSTATS command populates all these columns.

- The most important metric is in the PCTPAGESSAVED column. This column indicates how much space you are currently saving on a table by means of row compression.
- From the PCTROWSCOMPRESSED column, you can determine how many rows in the table are compressed.
- The AVGCOMPRESSEDROWSIZE column provides the average size of all the compressed rows. In most practical scenarios, the percentage of compressed rows should be near 100%. In this case, the average size of compressed rows is identical to the average row size that is maintained in the AVGROWSIZE column.

If the percentage of compressed rows in a table is significantly lower than 100%, this situation might be due to one of a few common issues:

- Your table might be in a regular table space, and compression might have cause the average row size to be very short. For a table in a regular table space, the effective minimal row size is limited by the fact that each page can hold at most 255 rows. If your average row size is near or below that ratio, consider converting the regular table space to a large table space. To do so, use the ALTER TABLESPACE statement with the CONVERT TO LARGE option, or use the ADMIN_MOVE_TABLE procedure to move your table to a large table space.
- If you enable row compression on an existing table with data in it, the table-level dictionary is built automatically by means of ADC when the table grows. Existing data remains in its uncompressed form until you perform a classic table reorganization or you use the ADMIN_MOVE_TABLE function.
- When you create a table with classic row compression enabled and start populating the table, the table-level dictionary is built automatically when the table size reaches the ADC threshold of 2 MB. However, the rows in the very beginning of the table remain uncompressed. If your table is only a few MB in size, this initial chunk of uncompressed rows can account for a significant fraction of the total rows that are stored in the table.

If none of the previous situations apply, check whether the table contains random data. For example, your table might have a large CHAR FOR BIT DATA or inlined LOB column that contains binary data that is compressed in some form by the application layer. In such cases, row compression might not be able to compress your data any further, so you should not enable it on that table.

Pay special attention when you apply row compression to MDC tables. Cell size and data density are important considerations for the physical design of MDC tables and

play an important role in choosing appropriate dimensions. In some cases, clustering dimensions are chosen so that cells use a very small number of pages.

Applying row compression reduces the size of each record and therefore the average size of the cells. If row compression causes the average cell size to be near or below the size of a single block, you might not see storage space savings reach their full potential. This is because space allocation in MDC tables is done block wise. Partially used blocks consume a full extent on disk but can contain a significant portion of free space.

When you do your physical database design, if you know that you will be using row compression, remember to factor in the compression ratio when you do the calculations for the cell size projection. If you decide to apply row compression on existing MDC tables, revalidate the cell size calculations. You might find it necessary to coarsify some of the dimensions or reduce the extent size of your table spaces.

Verifying index compression

Like for row compression, you can query the system catalog to check index compression settings. The COMPRESSION column in the SYSCAT.INDEXES view provides information about the compression settings for each index.

Verify that index compression is enabled for all the indexes that you considered for compression. Pay special attention to the system-generated indexes, for example, indexes that are implicitly created for primary key and unique columns. If you create your table without enabling row compression and subsequently use the ALTER TABLE statement to enable row compression, those indexes are uncompressed. You must enable index compression for those indexes explicitly by using the ALTER INDEX statement.

For those indexes that are enabled for compression, the PCTPAGESSAVED column provides information about the storage space savings. The RUNSTATS command populates this column.

Compression of backup images and log archives

Since DB2 Universal Database Version 8, you have been able to compress your backup images. Starting in DB2 Version 10.1, you can also apply compression to your log archives. These functions are available regardless of whether you have a license for the DB2 Storage Optimization Feature.

The default algorithm that is used to compress backup images and archived logs is similar to the one that the `compress(1)` UNIX utility uses.

Backup compression

You can enable backup compression independently for each backup image. When performing the backup operation, specify the COMPRESS clause, for example, as follows:

```
BACKUP DATABASE TDB2 to /vol_aux/backups/ COMPRESS
```

If you use row and index compression and you follow the guidelines to reduce the container sizes for your table spaces, the overall size of your database might be significantly reduced. Consequently, your backup images are automatically smaller than they would be if row and index compression were disabled. At the same time, the extra space savings that you can achieve by applying compression on backup images might be greatly reduced. This particularly applies when you have many tables that use adaptive row compression. However, compression on backup images can also compress metadata, LOBs, the catalog tables, and other database objects that cannot be compressed by any other means.

When deciding whether to compress your backup images, the most important indicators are the CPU and disk utilization rates while the backup is in progress. Backup compression can cause a significant overhead in CPU consumption. You should use backup compression only if the backup process is bottlenecked on I/O. This can happen, for example, if you store your backup images on a volume that is not striped across different physical disks or if you back up to network-attached or non-local storage media. In those cases, you increase CPU utilization throughout the backup process but the savings in I/O overhead might effectively shorten the backup window.

If you store your backups on Tivoli Storage Manager (TSM) software, you should use the compression and de-duplication functionality that is built into TSM. The de-duplication logic can lead to significantly better overall storage space savings if you keep a series of multiple recent backup images.

If only some of your tables use row and index compression, it is a good practice to separate the tables and indexes into different table spaces, based on their compression settings. Also, consider taking backups at the table space level rather than at the database level. In this case, you can balance the compression settings and compress only the backups for those table spaces that contain tables and indexes that are not compressed.

Log archive compression

If you configured your database for log archiving, you can enable compression for those log archives through database configuration parameters. You can enable or disable log archive compression independently for the primary or secondary archiving method and the corresponding archive locations. Log archive compression requires the database manager to handle the archiving process, that is, you must set your archiving method to DISK, TSM, or VENDOR. The following example shows how to set up a primary, disk-based log archive with compression enabled:

```
UPDATE DB CFG FOR TDB2 USING LOGARCHMETH1 DISK:/vol_aux/archive/tdb2

UPDATE DB CFG FOR TDB2 USING LOGARCHCOMPR1 ON
```

For the secondary archiving method, you can enable compression in a similar fashion.

Log archive compression, after you enable it, is fully automatic. The database manager automatically compresses log extents when they are moved from the active log path to the archive location. Upon retrieval of log files, which can happen during ROLLBACK and ROLLFORWARD command operations, the database manager automatically expands compressed log files when they are moved from the archive into the active or overflow log path. If, during recovery, the database manager encounters a compressed log extent in the active or overflow log path, it automatically expands that extent. This can happen, for example, if you manually retrieve log extents from the archive location into the active or overflow log path.

When deciding whether to enable compression on your log archives, considerations similar to those for backup compression apply. Check the I/O bandwidth that is available for your archive location, and see whether this is the bottleneck during the process of archiving.

Consider enabling compression if your log archives are on non-local volumes or volumes that are not striped across a number of disks. In addition, consider enabling compression if you are archiving your logs to an on-disk staging location for subsequent transfer onto tape through the db2tapemgr utility. In this case, compression helps reduce the time that it takes to transfer the archived logs to tape.

If you are archiving to TSM, consider using its built-in compression facilities. The behavior for log archives is very similar to that of the compression facilities in DB2 Version 10.1 because the same algorithm is used. Unlike backup compression, there is generally no additional benefit from de-duplication.

When you use row and index compression, some of the data that is written to the transaction log is already in compressed format. Therefore, the amount of data being logged is less than would be necessary if you performed the same operations on uncompressed tables. However, even if you are widely using row and index compression, there is generally enough metadata and other uncompressed items in the transaction log that you can achieve significant storage space savings on almost any log archive.

Best practices

- Carefully choose candidate tables and indexes for compression.
- Estimate compression ratios before choosing which row compression mode to apply.
- Understand and choose a specific approach to compressing your existing tables and indexes.
- Easily return storage space that is saved by compression back to the file system by using reclaimable storage table spaces.
- Monitor and gauge efficiency of row compression by using statistics.
- Mitigate the need for table reorganization by using adaptive row compression.
- Control storage space that is consumed by administrative data by compressing your backup images and log archives.

Conclusion

Applying the compression techniques that are provided by the DB2 Storage Optimization Feature can significantly reduce current and future storage space requirements and improve performance, especially where workload is I/O bound. By using this software with other compression features for administrative data, it is not difficult to implement a space-conscious storage strategy for your large OLTP workload or data warehouse.

Further reading

- DB2 Best Practices – http://www.ibm.com/developerworks/data/bestpractices/db2luw/
- DB2 Best Practices: Physical Database Design for Online Transaction Processing (OLTP) Environments – http://www.ibm.com/developerworks/data/bestpractices/databasedesign/
- DB2 Best Practices: Database Storage – http://www.ibm.com/developerworks/data/bestpractices/databasestorage/
- IBM Database Magazine, 2007 Volume 12 Issue 3; Distributed DBA: DB2 Deep Compression (Roger E. Sanders)
- IBM Database Magazine, 2008 Volume 13 Issue 1; Distributed DBA: DB2 Deep Compression Part 2 (Roger E. Sanders)
- IBM DB2 Deep Compression Brings Operational Savings to SAP Customers – https://www-304.ibm.com/easyaccess/fileserve?contentid=172719
- DB2 10: Multi-Temperature Data Management Recommendations – http://www.ibm.com/developerworks/data/library/long/dm-1205multitemp/index.html
- IBM Tivoli Storage Manager Information Center – http://publib.boulder.ibm.com/infocenter/tsminfo/v6r3/index.jsp

Contributors

Serge Boivin
Senior Writer, DB2 Information Development

Victor Chang
Team Lead, DB2 Data Warehouse Performance QA

Tom Hart
Software Engineer, DB2 Development

Bill Phu
Software Engineer, Systems Optimization Competency Center

Quentin Presley
Software Engineer, DB2 Development

Allan Risk
Senior Writer, DB2 Information Development

Sripriya Srinivasan
Software Engineer, DB2 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

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

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

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