Db2 11.1 for Linux, UNIX, and Windows

Data Movement Utilities Guide and Reference

Db2 11.1 for Linux, UNIX, and Windows

Data Movement Utilities Guide and Reference

Notice regarding this document

This document in PDF form is provided as a courtesy to customers who have requested documentation in this format. It is provided As-Is without warranty or maintenance commitment.

Contents

Figures

Tables

Data movement utilities and reference

Comparison between the ingest, import, and load utilities

The following tables summarize some of the key similarities and differences between the ingest, import, and load utilities.

Table 1. Supported table types

Table type	Ingest	Load	Import
Detached table	not supported	not supported	not supported
Global temporary table	not supported	not supported	not supported
Multidimensional clustering (MDC) or insert time clustering (ITC) table	supported	supported	supported
Materialized query table (MQT) that is maintained by user	supported	supported	supported
Nickname	supported	not supported	supported
Range-clustered table (RCT)	supported	not supported	supported
Range-partitioned table	supported	supported	supported
Summary table	supported	supported	supported
Temporal table	supported	supported	supported
Typed table	not supported	not supported	supported
Untyped (regular) table	supported	supported	supported
Updatable view (except typed view)	supported	not supported	supported

Table 2. Supported data types

Table type	Ingest	Load	Import
Numeric: SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, DOUBLE, DECFLOAT	supported	supported	supported
Character: CHAR, VARCHAR, NCHAR, NVARCHAR, plus corresponding FOR BIT DATA types	supported	supported	supported
Graphic: GRAPHIC, VARGRAPHIC	supported	supported	supported
Long types: LONG VARCHAR, LONG VARGRAPHIC	supported	supported	supported
Date/time: DATE, TIME, TIMESTAMP, including TIMESTAMP(p)	supported	supported	supported

Table 2. Supported data types (continued)

Table type	Ingest	Load	Import
BOOLEAN	supported	supported	supported
DB2SECURITYLABEL	supported	supported	supported
LOBs from files: BLOB, CLOB, DBCLOB, NCLOB	not supported	supported	supported
Inline LOBs	not supported	supported	supported
XML from files	not supported	supported	supported
Inline XML	not supported	supported	supported
Distinct type	supported (if based on a supported built-in data type)	supported	supported
Structured type	not supported	not supported	supported
Reference type	supported	supported	supported

Table 3. Supported input sources

Input type	Ingest Restartable?	Load Restartable?	Import Restartable?
Cursor	not supported n/a	supported yes	not supported n/a
Device	not supported n/a	supported yes	not supported n/a
File	supported yes	supported yes	supported yes
Pipe	supported yes	supported yes	not supported n/a

Table 4. Supported input formats

Table type	Ingest	Load	Import
ASC (including binary)	supported	supported	supported
Db2® for z/OS® UNLOAD format	not supported	not supported	not supported
DEL	supported	supported	supported
IXF	not supported	supported	supported

There are a number of other important differences that distinguish the ingest utility from the load and import utility:

- The ingest utility allows the input records to contain extra fields between the fields that correspond to columns.
- The ingest utility supports update, delete, and merge.
- The ingest utility supports constructing column values from expressions containing field values.
- The ingest utility allows other applications to update the target table while ingest is running.

File formats and data types

Export/Import/Load utility file formats

Four operating system file formats supported by the Db2 export, import, and load utilities are described:

DEL Delimited ASCII, for data exchange among a wide variety of database managers and file managers. This common approach to storing data uses special character delimiters to separate column values.

ASC Non-delimited ASCII, for importing or loading data from other applications that create flat text files with aligned column data.

PC/IXF
PC version of the Integration Exchange Format (IXF), the preferred method for data exchange within the database manager. PC/IXF is a structured description of a database table that contains an external representation of the internal table.

CURSOR
A cursor declared against an SQL query. This file type is only supported by the load utility.

When using DEL or ASC data file formats, define the table, including its column names and data types, before importing the file. The data types in the operating system file fields are converted into the corresponding type of data in the database table. The import utility accepts data with minor incompatibility problems, including character data imported with possible padding or truncation, and numeric data imported into different types of numeric fields.

When using the PC/IXF data file format, the table does not need to exist before you begin the import operation. However, the user-defined distinct type (UDT) does need to be defined, otherwise you receive an undefined name error (SQL0204N). Similarly, when you are exporting to the PC/IXF data file format, UDTs are stored in the output file.

When using the CURSOR file type, the table, including its column names and data types, must be defined before beginning the load operation. The column types of the SQL query must be compatible with the corresponding column types in the target table. It is not necessary for the specified cursor to be open before starting the load operation. The load utility will process the entire result of the query associated with the specified cursor whether or not the cursor has been used to fetch rows.

Moving data across platforms - file format considerations

Compatibility is important when exporting, importing, or loading data across platforms. The following sections describe PC/IXF and delimited ASCII (DEL) file format considerations when moving data between different operating systems.

PC/IXF file format

PC/IXF is the recommended file format for transferring data across platforms. PC/IXF files allow the load utility or the import utility to process (normally machine dependent) numeric data in a machine-independent fashion. For example, numeric data is stored and handled differently by Intel and other hardware architectures.

To provide compatibility of PC/IXF files among all products in the Db2 family, the export utility creates files with numeric data in Intel format, and the import utility expects it in this format.

Depending on the hardware platform, Db2 products convert numeric values between Intel and non-Intel formats (using byte reversal) during both export and import operations.

Implementations of Db2 database based on UNIX operating systems not create multiple-part PC/IXF files during export. However, they allow you to import a multiple-part PC/IXF file that was created by Db2. When importing this type of file, all parts should be in the same directory, otherwise an error is returned.

Single-part PC/IXF files created on UNIX operating systems with the Db2 export utility can be imported by Db2 database for Windows.

Delimited ASCII (DEL) file format

DEL files have differences based on the operating system on which they were created. The differences are:

- Row separator characters
  - Text files from UNIX operating systems use a line feed (LF) character.
  - Text files from other operating systems use a carriage return/line feed (CRLF) sequence.
- End-of-file character
  - Text files from UNIX operating systems do not have an end-of-file character.
  - Text files from other operating systems have an end-of-file character (X'1A').

Since DEL export files are text files, they can be transferred from one operating system to another. File transfer programs can handle operating system-dependant differences if you transfer the files in text mode; the conversion of row separator and end-of-file characters is not performed in binary mode.

Note: If character data fields contain row separator characters, these will also be converted during file transfer. This conversion causes unexpected changes to the data and, for this reason, it is recommended that you do not use DEL export files to move data across platforms. Use the PC/IXF file format instead.

Delimited ASCII (DEL) file format

A Delimited ASCII (DEL) file is a sequential ASCII file with row and column delimiters. Each DEL file is a stream of ASCII characters consisting of cell values ordered by row, and then by column. Rows in the data stream are separated by row delimiters; within each row, individual cell values are separated by column delimiters.

The following table describes the format of DEL files that can be imported, or that can be generated as the result of an export action.

```
DEL file ::= Row 1 data || Row delimiter ||
             Row 2 data || Row delimiter ||
             .
             .
             .
             Row n data || Optional row delimiter

Row i data ::= Cell value(i,1) || Column delimiter ||
               Cell value(i,2) || Column delimiter ||
```

```
                .
                .
                .
                Cell value(i,m)

Row delimiter ::= ASCII line feed sequence[a]

Column delimiter ::= Default value ASCII comma (,)[b]

Cell value(i,j) ::= Leading spaces
                  || ASCII representation of a numeric value
                     (integer, decimal, or float)
                  || Delimited character string
                  || Non-delimited character string
                  || Trailing spaces

Non-delimited character string ::= A set of any characters except a
                                   row delimiter or a column delimiter

Delimited character string ::= A character string delimiter ||
                               An extended character string ||
                               A character string delimiter ||
                               Trailing garbage

Trailing garbage ::= A set of any characters except a row delimiter
                     or a column delimiter

Character string delimiter ::= Default value ASCII double quotation
                               marks (")[c]

extended character string ::= || A set of any characters except a
                                 row delimiter or a character string
                                 delimiter if the NODOUBLEDEL
                                 modifier is specified
                              || A set of any characters except a
                                 row delimiter or a character string
                                 delimiter if the character string
                                 is not part of two consecutive
                                 character string delimiters
                              || A set of any characters except a
                                 character string delimiter if the
                                 character string delimiter is not
                                 part of two consecutive character
                                 string delimiters, and the DELPRIORITYCHAR
                                 modifier is specified

End-of-file character ::= Hex '1A' (Windows operating system only)

ASCII representation of a numeric value[d] ::= Optional sign '+' or '-'
    || 1 to 31 decimal digits with an optional decimal point before,
       after, or between two digits
    || Optional exponent

Exponent ::= Character 'E' or 'e'
    || Optional sign '+' or '-'
    || 1 to 3 decimal digits with no decimal point

Decimal digit ::= Any one of the characters '0', '1', ... '9'

Decimal point ::= Default value ASCII period (.)[e]
```

- [a] The record delimiter is assumed to be a new line character, ASCII x0A. Data generated on the Windows operating system can use the carriage return/line feed 2-byte standard of 0x0D0A. Data in EBCDIC code pages should use the EBCDIC LF character (0x25) as the record delimiter (EBCDIC data can be loaded using the `code page` file type modifier with the **LOAD** command).

- [b] The column delimiter can be specified with the `coldel` file type modifier.
- [c] The character string delimiter can be specified with the `chardel` file type modifier.

  Note: The default priority of delimiters is:
  1. Record delimiter
  2. Character delimiter
  3. Column delimiter
- [d] If the ASCII representation of a numeric value contains an exponent, it is a FLOAT constant. If it has a decimal point but no exponent, it is a DECIMAL constant. If it has no decimal point and no exponent, it is an INTEGER constant.
- [e] The decimal point character can be specified with the `decpt` file type modifier.

The export utility will replace every character string delimiter byte (default is double quote or x22) that is embedded within column data with two character string delimiter bytes (ie. doubling it). This is done so that the import parsing routines can distinguish between a character string delimiter byte that defines the beginning or end of a column, versus a character string delimiter byte embedded within the column data. Take caution when using an exported DEL file for some application other than the export utility, and note that the same doubling of character string delimiters occurs within 'FOR BIT' binary column data.

DEL data type descriptions:

The following table lists the data types and the acceptable forms for each one for the import and load utilities.

Table 5. Acceptable data type forms for the DEL file format

Data type	Form in files created by the export utility	Form acceptable to the import and load utilities
BIGINT	An INTEGER constant in the range -9223372036854775808 to 9223372036854775807.	An ASCII representation of a numeric value in the range -9223372036854775808 to 9223372036854775807. Decimal and float numbers are truncated to integer values.
BLOB, CLOB	Character data enclosed by character delimiters (for example, double quotation marks).	A delimited or non-delimited character string. The character string is used as the database column value.
BLOB_FILE, CLOB_FILE	The character data for each BLOB/CLOB column is stored in individual files, and the file name is enclosed by character delimiters.	The delimited or non-delimited name of the file that holds the data.
BOOLEAN	A Boolean value of 1 or 0. Other values (TRUE or FALSE, YES or NO, etc.) cannot be used.	A delimited or non-delimited character string containing the character "1" (indicating TRUE) or "0" (indicating FALSE).

Table 5. Acceptable data type forms for the DEL file format (continued)

Data type	Form in files created by the export utility	Form acceptable to the import and load utilities
CHAR	Character data enclosed by character delimiters (for example, double quotation marks).	A delimited or non-delimited character string. If required to match the width of the target column, the character string is leading truncated or padded with trailing spaces (X'20').
DATE	*yyyymmdd* (year month day) with no character delimiters. For example: 19931029 for 29 October 1993. Alternatively, the DATESISO option can be used to specify that all date values are to be exported in ISO format.	A delimited or non-delimited character string containing a date value in an ISO format consistent with the territory code of the target database, or a non-delimited character string of the form *yyyymmdd*.
DBCLOB (DBCS only)	Graphic data is exported as a delimited character string.	A delimited or non-delimited character string, an even number of bytes in length. The character string is used as the database column value.
DBCLOB_FILE (DBCS only)	The character data for each DBCLOB column is stored in individual files, and the file name is enclosed by character delimiters.	The delimited or non-delimited name of the file that holds the data.
DB2SECURITYLABEL	Column data is exported as "raw" data enclosed in quotation marks ("). Use the SECLABEL_TO_CHAR scalar function in the SELECT statement to convert the value to the security label string format.	The value in the data file is assumed by default to be the actual bytes that make up the internal representation of that security label, delimited by quotation marks (" ").
DECIMAL	A DECIMAL constant with the precision and scale of the field being exported. The `decplusblank` file type modifier can be used to specify that positive decimal values are to be prefixed with a blank space instead of a plus sign (+).	An ASCII representation of a numeric value that does not overflow the range of the database column into which the field is being imported. If the input value has more digits after the decimal point than can be accommodated by the database column, the excess digits are truncated.
FLOAT(long)	A FLOAT constant in the range -10E307 to 10E307.	An ASCII representation of a numeric value in the range -10E307 to 10E307.

Table 5. Acceptable data type forms for the DEL file format (continued)

Data type	Form in files created by the export utility	Form acceptable to the import and load utilities
GRAPHIC (DBCS only)	Graphic data is exported as a delimited character string.	A delimited or non-delimited character string, an even number of bytes in length. The character string is truncated or padded with double-byte spaces (for example, X'8140'), if necessary, to match the width of the database column.
INTEGER	An INTEGER constant in the range -2147483648 to 2147483647.	ASCII representation of a numeric value in the range -2147483648 to 2147483647. Decimal and float numbers are truncated to integer values.
LONG VARCHAR	Character data enclosed by character delimiters (for example, double quotation marks).	A delimited or non-delimited character string. The character string is used as the database column value.
LONG VARGRAPHIC (DBCS only)	Graphic data is exported as a delimited character string.	A delimited or non-delimited character string, an even number of bytes in length. The character string is used as the database column value.
SMALLINT	An INTEGER constant in the range -32768 to 32767.	AnASCII representation of a numeric value in the range -32768 to 32767. Decimal and float numbers are truncated to integer values.
TIME	*hh.mm.ss* (hour minutes seconds). A time value in ISO format enclosed by character delimiters, as shown in the following example: “09.39.43”	A delimited or non-delimited character string containing a time value in a format consistent with the territory code of the target database.
TIMESTAMP	*yyyy-mm-dd-hh.mm.ss.nnnnnn* (year month day hour minutes seconds microseconds). A character string representing a date and time enclosed by character delimiters.	A delimited or non-delimited character string containing a time stamp value acceptable for storage in a database.
VARCHAR	Character data enclosed by character delimiters (for example, double quotation marks).	A delimited or non-delimited character string. If required to match the width of the target column, the character string is leading or trailing truncated.

Table 5. Acceptable data type forms for the DEL file format (continued)

Data type	Form in files created by the export utility	Form acceptable to the import and load utilities
VARGRAPHIC (DBCS only)	Graphic data is exported as a delimited character string.	A delimited or non-delimited character string, an even number of bytes in length. The character string is truncated, if necessary, to match the maximum width of the database column.

Example DEL file:
Following is an example of a DEL file. Each line ends with a line feed sequence (on the Windows operating system, each line ends with a carriage return/line feed sequence).

```
"Smith, Bob",4973,15.46
"Jones, Bill",12345,16.34
"Williams, Sam",452,193.78
```

The following example illustrates the use of non-delimited character strings. The column delimiter has been changed to a semicolon, because the character data contains a comma.

```
Smith, Bob;4973;15.46
Jones, Bill;12345;16.34
Williams, Sam;452;193.78
```

Note:

1. A space (X'20') is never a valid delimiter.
2. Spaces that precede the first character, or that follow the last character of a cell value, are discarded during import. Spaces that are embedded in a cell value are not discarded.
3. A period (.) is not a valid character string delimiter, because it conflicts with periods in time stamp values.
4. For pure DBCS (graphic), mixed DBCS, and EUC, delimiters are restricted to the range of x00 to x3F, inclusive.
5. For DEL data specified in an EBCDIC code page, the delimiters might not coincide with the shift-in and shift-out DBCS characters.
6. On the Windows operating system, the first occurrence of an end-of-file character (X'1A') that is not within character delimiters indicates the end-of-file. Any subsequent data is not imported.
7. A null value is indicated by the absence of a cell value where one would normally occur, or by a string of spaces.
8. Since some products restrict character fields to 254 or 255 bytes, the export utility generates a warning message whenever a character column of maximum length greater than 254 bytes is selected for export. The import utility accommodates fields that are as long as the longest LONG VARCHAR and LONG VARGRAPHIC columns.

Delimiter considerations for moving data:

When moving delimited ASCII (DEL) files, it is important to ensure that the data being moved is not unintentionally altered because of problems with delimiter

character recognition. To help prevent these errors, Db2 enforces several restrictions and provides a number of file type modifiers.

Delimiter restrictions

There are a number of restrictions in place that help prevent the chosen delimiter character from being treated as a part of the data being moved. First, delimiters are mutually exclusive. Second, a delimiter cannot be binary zero, a line-feed character, a carriage-return, or a blank space. As well, the default decimal point (.) cannot be a string delimiter. Finally, in a DBCS environment, the pipe (`|`) character delimiter is not supported.

The following characters are specified differently by an ASCII-family code page and an EBCDIC-family code page:

- The Shift-In (0x0F) and the Shift-Out (0x0E) character cannot be delimiters for an EBCDIC MBCS data file.
- Delimiters for MBCS, EUC, or DBCS code pages cannot be greater than 0x40, except the default decimal point for EBCDIC MBCS data, which is 0x4b.
- Default delimiters for data files in ASCII code pages or EBCDIC MBCS code pages are:
  - string delimiter: "(0x22, double quotation mark)
  - column delimiter: **,**(0x2c, comma)
- Default delimiters for data files in EBCDIC SBCS code pages are:
  - string delimiter: "(0x7F, double quotation mark)
  - column delimiter: **,**(0x6B, comma)
- The default decimal point for ASCII data files is 0x2e (period).
- The default decimal point for EBCDIC data files is 0x4B (period).
- If the code page of the server is different from the code page of the client, it is recommended that the hex representation of non-default delimiters be specified. For example,

  ```
  db2 load from ... modified by chardel0x0C coldelX1e ...
  ```

Issues with delimiters during data movement

Double character delimiters

By default, for character-based fields of a DEL file, any instance of the character delimiter found within the field is represented by double character delimiters. For example, assuming that the character delimiter is the double quote, if you export the text `I am 6" tall.`, the output text in the DEL file reads `"I am 6"" tall."` Conversely, if the input text in a DEL file reads `"What a ""nice"" day!"`, the text is imported as `What a "nice" day!`

nodoubledel

Double character delimiter behavior can be disabled for the import, export, and load utilities by specifying the `nodoubledel` file type modifier. However, be aware that double character delimiter behavior exists in order to avoid parsing errors. When you use `nodoubledel` with export, the character delimiter is not doubled if it is present in character fields. When you use `nodoubledel` with import and load, the double character delimiter is not interpreted as a literal instance of the character delimiter.

nochardel
When you use the nochardel file type modifier with export, the character fields are not surrounded by character delimiters. When nochardel is used import and load, the character delimiters are not treated as special characters and are interpreted as actual data.

chardel
Other file type modifiers can be used to manually prevent confusion between default delimiters and the data. Thechardel file type modifier specifies x, a single character, as the character string delimiter to be used instead of double quotation marks (as is the default).

coldel
Similarly, if you wanted to avoid using the default comma as a column delimiter, you could use coldel, which specifies x, a single character, as the column data delimiter.

delprioritychar
Another concern in regards to moving DEL files is maintaining the correct precedence order for delimiters. The default priority for delimiters is: row, character, column. However, some applications depend on the priority: character, row, column. For example, using the default priority, the DEL data file:

```
"Vincent <row delimiter> is a manager",<row delimiter>
```

would be interpreted as having two rows: Vincent, and is a manager, since <row delimiter> takes precedence over the character delimiter ("). Using delprioritychar gives the character delimiter (") precedence over the row delimiter (<row delimiter>), meaning that the same DEL file would be interpreted (correctly) as having one row: Vincent is a manager.

Non-delimited ASCII (ASC) file format

The non-delimited ASCII format, known as ASC to the import and load utilities, comes in two varieties: fixed length and flexible length. For fixed length ASC, all records are of a fixed length. For flexible length ASC, records are delimited by a row delimiter (always a new line). The term *non-delimited* in non-delimited ASCII means that column values are not separated by delimiters.

When importing or loading ASC data, specifying the reclen file type modifier will indicate that the datafile is fixed length ASC. Not specifying it means that the datafile is flexible length ASC.

The non-delimited ASCII format, can be used for data exchange with any ASCII product that has a columnar format for data, including word processors. Each ASC file is a stream of ASCII characters consisting of data values ordered by row and column. Rows in the data stream are separated by row delimiters. Each column within a row is defined by a beginning-ending location pair (specified by **IMPORT** parameters). Each pair represents locations within a row specified as byte positions. The first position within a row is byte position 1. The first element of each location pair is the byte on which the column begins, and the second element of each location pair is the byte on which the column ends. The columns might overlap. Every row in an ASC file has the same column definition.

An ASC file is defined by:

```
ASC file ::= Row 1 data || Row delimiter ||
             Row 2 data || Row delimiter ||
             .
```

```
.
.
Row n data

Row i data ::= ASCII characters || Row delimiter

Row Delimiter ::= ASCII line feed sequence[a]
```

- [a] The record delimiter is assumed to be a new line character, ASCII x0A. Data generated on the Windows operating system can use the carriage return/line feed 2-byte standard of 0x0D0A. Data in EBCDIC code pages should use the EBCDIC LF character (0x25) as the record delimiter (EBCDIC data can be loaded using the `codepage` file type modifier with the **LOAD** command). The record delimiter is never interpreted to be part of a field of data.

ASC data type descriptions:

The following table lists the data types and the acceptable forms for each one for the import and load utilities.

Table 6. Acceptable Data Type Forms for the ASC File Format

Data type	Acceptable forms
BIGINT	A constant in any numeric type (SMALLINT, INTEGER, BIGINT, DECIMAL, or FLOAT) is accepted. Individual values are rejected if they are not in the range -9223372036854775808 to 9223372036854775807. Decimal numbers are truncated to integer values. A comma, period, or colon is considered to be a decimal point. Thousands separators are not allowed. The beginning and ending locations should specify a field whose width does not exceed 50 bytes. Integers, decimal numbers, and the mantissas of floating point numbers can have no more than 31 digits. Exponents of floating point numbers can have no more than 3 digits.
BLOB/CLOB	A string of characters. The character string is truncated on the right, if necessary, to match the maximum length of the target column. If the ASC *truncate blanks* option is in effect, trailing blanks are stripped from the original or the truncated string.
BLOB_FILE, CLOB_FILE, DBCLOB_FILE (DBCS only)	A delimited or non-delimited name of the file that holds the data.
BOOLEAN	A Boolean value of 1 or 0. Other values (TRUE or FALSE, YES or NO, etc.) cannot be used.
CHAR	A string of characters. If required to match the width of the target column, the character string is leading truncated or padded with trailing spaces.
DATE	A character string representing a date value in a format consistent with the territory code of the target database. The beginning and ending locations should specify a field width that is within the range for the external representation of a date.
DBCLOB (DBCS only)	A string of an even number of bytes. A string of an odd number of bytes is invalid and is not accepted. A valid string is truncated on the right, if necessary, to match the maximum length of the target column.

Table 6. Acceptable Data Type Forms for the ASC File Format (continued)

Data type	Acceptable forms
DECIMAL	A constant in any numeric type (SMALLINT, INTEGER, BIGINT, DECIMAL, or FLOAT) is accepted. Individual values are rejected if they are not in the range of the database column into which they are being imported. If the input value has more digits after the decimal point than the scale of the database column, the excess digits are truncated. A comma, period, or colon is considered to be a decimal point. Thousands separators are not allowed. The beginning and ending locations should specify a field whose width does not exceed 50 bytes. Integers, decimal numbers, and the mantissas of floating point numbers can have no more than 31 digits. Exponents of floating point numbers can have no more than 3 digits.
FLOAT(long)	A constant in any numeric type (SMALLINT, INTEGER, BIGINT, DECIMAL, or FLOAT) is accepted. All values are valid. A comma, period, or colon is considered to be a decimal point. An uppercase or lowercase E is accepted as the beginning of the exponent of a FLOAT constant. The beginning and ending locations should specify a field whose width does not exceed 50 bytes. Integers, decimal numbers, and the mantissas of floating point numbers can have no more than 31 digits. Exponents of floating point numbers can have no more than 3 digits.
GRAPHIC (DBCS only)	A string of an even number of bytes. A string of an odd number of bytes is invalid and is not accepted. A valid string is truncated or padded with double-byte spaces (0x8140) on the right, if necessary, to match the maximum length of the target column.
INTEGER	A constant in any numeric type (SMALLINT, INTEGER, BIGINT, DECIMAL, or FLOAT) is accepted. Individual values are rejected if they are not in the range -2147483648 to 2147483647. Decimal numbers are truncated to integer values. A comma, period, or colon is considered to be a decimal point. Thousands separators are not allowed. The beginning and ending locations should specify a field whose width does not exceed 50 bytes. Integers, decimal numbers, and the mantissas of floating point numbers can have no more than 31 digits. Exponents of floating point numbers can have no more than 3 digits.
LONG VARCHAR	A string of characters. If required to match the maximum length of the target column, the character string is leading truncated. If the ASC *truncate blanks* option is in effect, trailing blanks are stripped from the original or the truncated string.
LONG VARGRAPHIC (DBCS only)	A string of an even number of bytes. A string of an odd number of bytes is invalid and is not accepted. A valid string is truncated on the right, if necessary, to match the maximum length of the target column.

Table 6. Acceptable Data Type Forms for the ASC File Format (continued)

Data type	Acceptable forms
SMALLINT	A constant in any numeric type (SMALLINT, INTEGER, BIGINT, DECIMAL, or FLOAT) is accepted. Individual values are rejected if they are not in the range -32768 to 32767. Decimal numbers are truncated to integer values. A comma, period, or colon is considered to be a decimal point. Thousands separators are not allowed. The beginning and ending locations should specify a field whose width does not exceed 50 bytes. Integers, decimal numbers, and the mantissas of floating point numbers can have no more than 31 digits. Exponents of floating point numbers can have no more than 3 digits.
TIME	A character string representing a time value in a format consistent with the territory code of the target database. The beginning and ending locations should specify a field width that is within the range for the external representation of a time.
TIMESTAMP	A character string representing a time stamp value acceptable for storage in a database. The beginning and ending locations should specify a field width that is within the range for the external representation of a time stamp.
VARCHAR	A string of characters. If required to match the maximum length of the target column, the character string is leading truncated. If the ASC *truncate blanks* option is in effect, trailing blanks are stripped from the original or the truncated string.
VARGRAPHIC (DBCS only)	A string of an even number of bytes. A string of an odd number of bytes is invalid and is not accepted. A valid string is truncated on the right, if necessary, to match the maximum length of the target column.

Example ASC file:
Following is an example of an ASC file. Each line ends with a line feed sequence (on the Windows operating system, each line ends with a carriage return/line feed sequence).

```
Smith, Bob       4973     15.46
Jones, Suzanne   12345    16.34
Williams, Sam    452123  193.78
```

Note:

1. ASC files are assumed not to contain column names.
2. Character strings are *not* enclosed by delimiters. The data type of a column in the ASC file is determined by the data type of the target column in the database table.
3. A NULL is imported into a nullable database column if:
   - A field of blanks is targeted for a numeric, DATE, TIME, or TIMESTAMP database column
   - A field with no beginning and ending location pairs is specified
   - A location pair with beginning and ending locations equal to zero is specified

- A row of data is too short to contain a valid value for the target column
- The `NULL INDICATORS` load option is used, and an `N` (or other value specified by the user) is found in the null indicator column.

4. If the target column is not nullable, an attempt to import a field of blanks into a numeric, DATE, TIME, or TIMESTAMP column causes the row to be rejected.
5. If the input data is not compatible with the target column, and that column is nullable, a null is imported or the row is rejected, depending on where the error is detected. If the column is not nullable, the row is rejected. Messages are written to the message file, specifying incompatibilities that are found.

PC version of IXF file format

The PC version of IXF (PC/IXF) file format is a database manager adaptation of the Integration Exchange Format (IXF) data interchange architecture. The IXF architecture was specifically designed to enable the exchange of relational database structures and data. The PC/IXF architecture allows the database manager to export a database without having to anticipate the requirements and idiosyncrasies of a receiving product. Similarly, a product importing a PC/IXF file need only understand the PC/IXF architecture; the characteristics of the product which exported the file are not relevant. The PC/IXF file architecture maintains the independence of both the exporting and the importing database systems.

The IXF architecture is a generic relational database exchange format that supports a rich set of relational data types, including some types that might not be supported by specific relational database products. The PC/IXF file format preserves this flexibility; for example, the PC/IXF architecture supports both single-byte character string (SBCS) and double-byte character string (DBCS) data types. Not all implementations support all PC/IXF data types; however, even restricted implementations provide for the detection and disposition of unsupported data types during import.

In general, a PC/IXF file consists of an unbroken sequence of variable-length records. The file contains the following record types in the order shown:

- One header record of record type H
- One table record of record type T
- Multiple column descriptor records of record type C (one record for each column in the table)
- Multiple data records of record type D (each row in the table is represented by one or more D records).

A PC/IXF file might also contain application records of record type A, anywhere after the H record. These records are permitted in PC/IXF files to enable an application to include additional data, not defined by the PC/IXF format, in a PC/IXF file. A records are ignored by any program reading a PC/IXF file that does not have particular knowledge about the data format and content implied by the application identifier in the A record.

Every record in a PC/IXF file begins with a record length indicator. This is a 6-byte right-aligned character representation of an integer value specifying the length, in bytes, of the portion of the PC/IXF record that follows the record length indicator; that is, the total record size minus 6 bytes. Programs reading PC/IXF files should use these record lengths to locate the end of the current record and the beginning of the next record. H, T, and C records must be sufficiently large to include all of their defined fields, and, of course, their record length fields must agree with their actual lengths. However, if extra data (for example, a *new* field), is added to the

end of one of these records, pre-existing programs reading PC/IXF files should ignore the extra data, and generate no more than a warning message. Programs writing PC/IXF files, however, should write H, T and C records that are the precise length needed to contain all of the defined fields.

If a PC/IXF file contains LOB Location Specifier (LLS) columns, each LLS column must have its own D record. D records are automatically created by the export utility, but you will need to create them manually if you are using a third party tool to generate the PC/IXF files. Further, an LLS is required for each LOB column in a table, including those with a null value. If a LOB column is null, you will need to create an LLS representing a null LOB.

The D record entry for each XML column will contain two bytes little endian indicating the XML data specifier (XDS) length, followed by the XDS itself.

For example, the following XDS:

```
XDS FIL="a.xml" OFF="1000" LEN="100" SCH="RENATA.SCHEMA" />
```

will be represented by the following bytes in a D record:

```
0x3D 0x00 XDS FIL="a.xml" OFF="1000" LEN="100" SCH="RENATA.SCHEMA" />
```

PC/IXF file records are composed of fields which contain character data. The import and export utilities interpret this character data using the CPGID of the target database, with two exceptions:

- The IXFADATA field of A records.

  The code page environment of character data contained in an IXFADATA field is established by the application which creates and processes a particular A record; that is, the environment varies by implementation.
- The IXFDCOLS field of D records.

  The code page environment of character data contained in an IXFDCOLS field is a function of information contained in the C record which defines a particular column and its data.

Numeric fields in H, T, and C records, and in the prefix portion of D and A records should be right-aligned single-byte character representations of integer values, filled with leading zeros or blanks. A value of zero should be indicated with at least one (right-aligned) zero character, not blanks. Whenever one of these numeric fields is not used, for example IXFCLENG, where the length is implied by the data type, it should be filled with blanks. These numeric fields are:

```
IXFHRECL, IXFTRECL, IXFCRECL, IXFDRECL, IXFARECL,
IXFHHCNT, IXFHSBCP, IXFHDBCP, IXFTCCNT, IXFTNAML,
IXFCLENG, IXFCDRID, IXFCPOSN, IXFCNAML, IXFCTYPE,
IXFCSBCP, IXFCDBCP, IXFCNDIM, IXFCDSIZ, IXFDRID
```

Note: The database manager PC/IXF file format is not identical to the System/370.

PC/IXF record types:
There are five basic PC/IXF record types:

- header
- table
- column descriptor
- data
- application

There are seven application subtypes that Db2 uses:

- index
- hierarchy
- subtable
- continuation
- terminate
- identity
- Db2 SQLCA

Each PC/IXF record type is defined as a sequence of fields; these fields are required, and must appear in the order shown.

HEADER RECORD

FIELD NAME	LENGTH	TYPE	COMMENTS
IXFHRECL	06-BYTE	CHARACTER	record length
IXFHRECT	01-BYTE	CHARACTER	record type = 'H'
IXFHID	03-BYTE	CHARACTER	IXF identifier
IXFHVERS	04-BYTE	CHARACTER	IXF version
IXFHPROD	12-BYTE	CHARACTER	product
IXFHDATE	08-BYTE	CHARACTER	date written
IXFHTIME	06-BYTE	CHARACTER	time written
IXFHHCNT	05-BYTE	CHARACTER	heading record count
IXFHSBCP	05-BYTE	CHARACTER	single byte code page
IXFHDBCP	05-BYTE	CHARACTER	double byte code page
IXFHFIL1	02-BYTE	CHARACTER	reserved

The following fields are contained in the header record:

IXFHRECL
The record length indicator. A 6-byte character representation of an integer value specifying the length, in bytes, of the portion of the PC/IXF record that follows the record length indicator; that is, the total record size minus 6 bytes. The H record must be sufficiently long to include all of its defined fields.

IXFHRECT
The IXF record type, which is set to H for this record.

IXFHID
The file format identifier, which is set to IXF for this file.

IXFHVERS
The PC/IXF format level used when the file was created, which is set to '0002'.

IXFHPROD
A field that can be used by the program creating the file to identify itself. If this field is filled in, the first six bytes are used to identify the product creating the file, and the last six bytes are used to indicate the version or release of the creating product. The database manager uses this field to signal the existence of database manager-specific data.

IXFHDATE
The date on which the file was written, in the form *yyyymmdd*.

IXFHTIME
The time at which the file was written, in the form *hhmmss*. This field is optional and can be left blank.

IXFHHCNT
The number of H, T, and C records in this file that precede the first data record. A records are not included in this count.

IXFHSBCP
Single-byte code page field, containing a single-byte character representation of a SBCS CPGID or '00000'.

The export utility sets this field equal to the SBCS CPGID of the exported database table. For example, if the table SBCS CPGID is 850, this field contains '00850'.

IXFHDBCP
Double-byte code page field, containing a single-byte character representation of a DBCS CPGID or '00000'.

The export utility sets this field equal to the DBCS CPGID of the exported database table. For example, if the table DBCS CPGID is 301, this field contains '00301'.

IXFHFIL1
Spare field set to two blanks to match a reserved field in host IXF files.

```
TABLE RECORD

   FIELD NAME     LENGTH     TYPE        COMMENTS
   ----------     -------    ---------   -------------

   IXFTRECL       006-BYTE   CHARACTER   record length
   IXFTRECT       001-BYTE   CHARACTER   record type = 'T'
   IXFTNAML       003-BYTE   CHARACTER   name length
   IXFTNAME       256-BYTE   CHARACTER   name of data
   IXFTQULL       003-BYTE   CHARACTER   qualifier length
   IXFTQUAL       256-BYTE   CHARACTER   qualifier
   IXFTSRC        012-BYTE   CHARACTER   data source
   IXFTDATA       001-BYTE   CHARACTER   data convention = 'C'
   IXFTFORM       001-BYTE   CHARACTER   data format = 'M'
   IXFTMFRM       005-BYTE   CHARACTER   machine format = 'PC'
   IXFTLOC        001-BYTE   CHARACTER   data location = 'I'
   IXFTCCNT       005-BYTE   CHARACTER   'C' record count
   IXFTFIL1       002-BYTE   CHARACTER   reserved
   IXFTDESC       030-BYTE   CHARACTER   data description
   IXFTPKNM       257-BYTE   CHARACTER   primary key name
   IXFTDSPC       257-BYTE   CHARACTER   reserved
   IXFTISPC       257-BYTE   CHARACTER   reserved
   IXFTLSPC       257-BYTE   CHARACTER   reserved
```

The following fields are contained in the table record:

IXFTRECL
The record length indicator. A 6-byte character representation of an integer value specifying the length, in bytes, of the portion of the PC/IXF record that follows the record length indicator; that is, the total record size minus 6 bytes. The T record must be sufficiently long to include all of its defined fields.

IXFTRECT
The IXF record type, which is set to T for this record.

IXFTNAML
The length, in bytes, of the table name in the IXFTNAME field.

IXFTNAME
The name of the table. If each file has only one table, this is an informational field only. The database manager does not use this field

when importing data. When writing a PC/IXF file, the database manager writes the DOS file name (and possibly path information) to this field.

IXFTQULL
The length, in bytes, of the table name qualifier in the IXFTQUAL field.

IXFTQUAL
Table name qualifier, which identifies the creator of a table in a relational system. This is an informational field only. If a program writing a file has no data to write to this field, the preferred fill value is blanks. Programs reading a file might print or display this field, or store it in an informational field, but no computations should depend on the content of this field.

IXFTSRC
Used to indicate the original source of the data. This is an informational field only. If a program writing a file has no data to write to this field, the preferred fill value is blanks. Programs reading a file might print or display this field, or store it in an informational field, but no computations should depend on the content of this field.

IXFTDATA
Convention used to describe the data. This field must be set to `C` for import and export, indicating that individual column attributes are described in the following column descriptor (C) records, and that data follows PC/IXF conventions.

IXFTFORM
Convention used to store numeric data. This field must be set to `M`, indicating that numeric data in the data (D) records is stored in the machine (internal) format specified by the IXFTMFRM field.

IXFTMFRM
The format of any machine data in the PC/IXF file. The database manager will only read or write files if this field is set to `PC`*bbb*, where *b* represents a blank, and PC specifies that data in the PC/IXF file is in IBM® PC machine format.

IXFTLOC
The location of the data. The database manager only supports a value of I, meaning the data is internal to this file.

IXFTCCNT
The number of C records in this table. It is a right-aligned character representation of an integer value.

IXFTFIL1
Spare field set to two blanks to match a reserved field in host IXF files.

IXFTDESC
Descriptive data about the table. This is an informational field only. If a program writing a file has no data to write to this field, the preferred fill value is blanks. Programs reading a file might print or display this field, or store it in an informational field, but no computations should depend on the content of this field. This field contains `NOT NULL WITH DEFAULT` if the column was not null with default, and the table name came from a workstation database.

IXFTPKNM
The name of the primary key defined on the table (if any). The name is stored as a null-terminated string.

IXFTDSPC
This field is reserved for future use.

IXFTISPC
This field is reserved for future use.

IXFTLSPC
This field is reserved for future use.

COLUMN DESCRIPTOR RECORD

FIELD NAME	LENGTH	TYPE	COMMENTS
IXFCRECL	006-BYTE	CHARACTER	record length
IXFCRECT	001-BYTE	CHARACTER	record type = 'C'
IXFCNAML	003-BYTE	CHARACTER	column name length
IXFCNAME	256-BYTE	CHARACTER	column name
IXFCNULL	001-BYTE	CHARACTER	column allows nulls
IXFCDEF	001-BYTE	CHARACTER	column has defaults
IXFCSLCT	001-BYTE	CHARACTER	column selected flag
IXFCKPOS	002-BYTE	CHARACTER	position in primary key
IXFCCLAS	001-BYTE	CHARACTER	data class
IXFCTYPE	003-BYTE	CHARACTER	data type
IXFCSBCP	005-BYTE	CHARACTER	single byte code page
IXFCDBCP	005-BYTE	CHARACTER	double byte code page
IXFCLENG	005-BYTE	CHARACTER	column data length
IXFCDRID	003-BYTE	CHARACTER	'D' record identifier
IXFCPOSN	006-BYTE	CHARACTER	column position
IXFCDESC	030-BYTE	CHARACTER	column description
IXFCLOBL	020-BYTE	CHARACTER	lob column length
IXFCUDTL	003-BYTE	CHARACTER	UDT name length
IXFCUDTN	256-BYTE	CHARACTER	UDT name
IXFCDEFL	003-BYTE	CHARACTER	default value length
IXFCDEFV	254-BYTE	CHARACTER	default value
IXFCREF	001-BYTE	CHARACTER	reference type
IXFCNDIM	002-BYTE	CHARACTER	number of dimensions
IXFCDSIZ	varying	CHARACTER	size of each dimension

The following fields are contained in column descriptor records:

IXFCRECL
The record length indicator. A 6-byte character representation of an integer value specifying the length, in bytes, of the portion of the PC/IXF record that follows the record length indicator; that is, the total record size minus 6 bytes. The C record must be sufficiently long to include all of its defined fields.

IXFCRECT
The IXF record type, which is set to C for this record.

IXFCNAML
The length, in bytes, of the column name in the IXFCNAME field.

IXFCNAME
The name of the column.

IXFCNULL
Specifies if nulls are permitted in this column. Valid settings are `Y` or `N`.

IXFCDEF
Specifies if a default value is defined for this field. Valid settings are `Y` or `N`.

IXFCSLCT
An obsolete field whose intended purpose was to allow selection of a subset of columns in the data. Programs writing PC/IXF files should always store a Y in this field. Programs reading PC/IXF files should ignore the field.

IXFCKPOS
The position of the column as part of the primary key. Valid values range from `01` to `16`, or `N` if the column is not part of the primary key.

IXFCCLAS
The class of data types to be used in the IXFCTYPE field. The database manager only supports relational types (`R`).

IXFCTYPE
The data type for the column.

IXFCSBCP
Contains a single-byte character representation of a SBCS CPGID. This field specifies the CPGID for single-byte character data, which occurs with the IXFDCOLS field of the D records for this column.

The semantics of this field vary with the data type for the column (specified in the IXFCTYPE field).

- For a character string column, this field should normally contain a non-zero value equal to that of the IXFHSBCP field in the H record; however, other values are permitted. If this value is zero, the column is interpreted to contain bit string data.
- For a numeric column, this field is not meaningful. It is set to zero by the export utility, and ignored by the import utility.
- For a date or time column, this field is not meaningful. It is set to the value of the IXFHSBCP field by the export utility, and ignored by the import utility.
- For a graphic column, this field must be zero.

IXFCDBCP
Contains a single-byte character representation of a DBCS CPGID. This field specifies the CPGID for double-byte character data, which occurs with the IXFDCOLS field of the D records for this column.

The semantics of this field vary with the data type for the column (specified in the IXFCTYPE field).

- For a character string column, this field should either be zero, or contain a value equal to that of the IXFHDBCP field in the H record; however, other values are permitted. If the value in the IXFCSBCP field is zero, the value in this field must be zero.
- For a numeric column, this field is not meaningful. It is set to zero by the export utility, and ignored by the import utility.
- For a date or time column, this field is not meaningful. It is set to zero by the export utility, and ignored by the import utility.
- For a graphic column, this field must have a value equal to the value of the IXFHDBCP field.

IXFCLENG
Provides information about the size of the column being described. For some data types, this field is unused, and should contain blanks. For other data types, this field contains the right-aligned character representation of an integer specifying the column length. For yet other data types, this field is divided into two subfields: 3 bytes for precision, and 2 bytes for scale; both of these subfields are right-aligned character representations of integers. Starting with Version 9.7, for a timestamp data type this field contains the right-aligned character representation of an integer specifying the timestamp precision.

IXFCDRID
The D record identifier. This field contains the right-aligned character representation of an integer value. Several D records can be used to contain each row of data in the PC/IXF file. This field specifies which D record (of the several D records contributing to a row of data) contains the data for the column. A value of one (for example, `001`) indicates that the data for a column is in the first D record in a row of data. The first C record must have an IXFCDRID value of one. All subsequent C records must have an IXFCDRID value equal to the value in the preceding C record, or one higher.

IXFCPOSN
The value in this field is used to locate the data for the column within one of the D records representing a row of table data. It is the starting position of the data for this column within the IXFDCOLS field of the D record. If the column is nullable, IXFCPOSN points to the null indicator; otherwise, it points to the data itself. If a column contains varying length data, the data itself begins with the current length indicator. The IXFCPOSN value for the first byte in the IXFDCOLS field of the D record is one (not zero). If a column is in a new D record, the value of IXFCPOSN should be one; otherwise, IXFCPOSN values should increase from column to column to such a degree that the data values do not overlap.

IXFCDESC
Descriptive information about the column. This is an informational field only. If a program writing to a file has no data to write to this field, the preferred fill value is blanks. Programs reading a file might print or display this field, or store it in an informational field, but no computations should depend on the content of this field.

IXFCLOBL
The length, in bytes, of the long or the LOB defined in this column. If this column is not a long or a LOB, the value in this field is `000`.

IXFCUDTL
The length, in bytes, of the user defined type (UDT) name in the IXFCUDTN field. If the type of this column is not a UDT, the value in this field is `000`.

IXFCUDTN
The name of the user defined type that is used as the data type for this column.

IXFCDEFL
The length, in bytes, of the default value in the IXFCDEFV field. If this column does not have a default value, the value in this field is `000`.

IXFCDEFV
Specifies the default value for this column, if one has been defined.

IXFCREF
If the column is part of a hierarchy, this field specifies whether the column is a data column (`D`), or a reference column (`R`).

IXFCNDIM
The number of dimensions in the column. Arrays are not supported in this version of PC/IXF. This field must therefore contain a character representation of a zero integer value.

IXFCDSIZ
The size or range of each dimension. The length of this field is five bytes

per dimension. Since arrays are not supported (that is, the number of dimensions must be zero), this field has zero length, and does not actually exist.

```
DATA RECORD

   FIELD NAME     LENGTH    TYPE        COMMENTS
   ----------     -------   ---------   -------------
   IXFDRECL       06-BYTE   CHARACTER   record length
   IXFDRECT       01-BYTE   CHARACTER   record type = 'D'
   IXFDRID        03-BYTE   CHARACTER   'D' record identifier
   IXFDFIL1       04-BYTE   CHARACTER   reserved
   IXFDCOLS       varying   variable    columnar data
```

The following fields are contained in the data records:

IXFDRECL
The record length indicator. A 6-byte character representation of an integer value specifying the length, in bytes, of the portion of the PC/IXF record that follows the record length indicator; that is, the total record size minus 6 bytes. Each D record must be sufficiently long to include all significant data for the current occurrence of the last data column stored in the record.

IXFDRECT
The IXF record type, which is set to D for this record, indicating that it contains data values for the table.

IXFDRID
The record identifier, which identifies a particular D record within the sequence of several D records contributing to a row of data. For the first D record in a row of data, this field has a value of one; for the second D record in a row of data, this field has a value of two, and so on. In each row of data, all the D record identifiers called out in the C records must actually exist.

IXFDFIL1
Spare field set to four blanks to match reserved fields, and hold a place for a possible shift-out character, in host IXF files.

IXFDCOLS
The area for columnar data. The data area of a data record (D record) is composed of one or more column entries. There is one column entry for each column descriptor record, which has the same D record identifier as the D record. In the D record, the starting position of the column entries is indicated by the IXFCPOSN value in the C records.

The format of the column entry data depends on whether or not the column is nullable:

- If the column is nullable (the IXFCNULL field is set to `Y`), the column entry data includes a null indicator. If the column is not null, the indicator is followed by data type-specific information, including the actual database value. The null indicator is a two-byte value set to x'0000' for not null, and x'FFFF' for null.
- If the column is not nullable, the column entry data includes only data type-specific information, including the actual database value.

For varying-length data types, the data type-specific information includes a current length indicator. The current length indicators are 2-byte integers in a form specified by the IXFTMFRM field.

The length of the data area of a D record cannot exceed 32 771 bytes.

APPLICATION RECORD

FIELD NAME	LENGTH	TYPE	COMMENTS
IXFARECL	06-BYTE	CHARACTER	record length
IXFARECT	01-BYTE	CHARACTER	record type = 'A'
IXFAPPID	12-BYTE	CHARACTER	application identifier
IXFADATA	varying	variable	application-specific data

The following fields are contained in application records:

IXFARECL
The record length indicator. A 6-byte character representation of an integer value specifying the length, in bytes, of the portion of the PC/IXF record that follows the record length indicator; that is, the total record size minus 6 bytes. Each A record must be sufficiently long to include at least the entire IXFAPPID field.

IXFARECT
The IXF record type, which is set to A for this record, indicating that this is an application record. These records are ignored by programs which do not have particular knowledge about the content and the format of the data implied by the application identifier.

IXFAPPID
The application identifier, which identifies the application creating the A record. PC/IXF files created by the database manager can have A records with the first 6 characters of this field set to a constant identifying the database manager, and the last 6 characters identifying the release or version of the database manager or another application writing the A record.

IXFADATA
This field contains application dependent supplemental data, whose form and content are known only to the program creating the A record, and to other applications which are likely to process the A record.

DB2 INDEX RECORD

FIELD NAME	LENGTH	TYPE	COMMENTS
IXFARECL	006-BYTE	CHARACTER	record length
IXFARECT	001-BYTE	CHARACTER	record type = 'A'
IXFAPPID	012-BYTE	CHARACTER	application identifier = 'DB2 02.00'
IXFAITYP	001-BYTE	CHARACTER	application specific data type = 'I'
IXFADATE	008-BYTE	CHARACTER	date written from the 'H' record
IXFATIME	006-BYTE	CHARACTER	time written from the 'H' record
IXFANDXL	002-BYTE	SHORT INT	length of name of the index
IXFANDXN	256-BYTE	CHARACTER	name of the index
IXFANCL	002-BYTE	SHORT INT	length of name of the index creator
IXFANCN	256-BYTE	CHARACTER	name of the index creator
IXFATABL	002-BYTE	SHORT INT	length of name of the table
IXFATABN	256-BYTE	CHARACTER	name of the table
IXFATCL	002-BYTE	SHORT INT	length of name of the table creator
IXFATCN	256-BYTE	CHARACTER	name of the table creator
IXFAUNIQ	001-BYTE	CHARACTER	unique rule
IXFACCNT	002-BYTE	SHORT INT	column count
IXFAREVS	001-BYTE	CHARACTER	allow reverse scan flag
IXFAIDXT	001-BYTE	CHARACTER	type of index
IXFAPCTF	002-BYTE	CHARACTER	amount of pct free
IXFAPCTU	002-BYTE	CHARACTER	amount of minpctused
IXFAEXTI	001-BYTE	CHARACTER	reserved
IXFACNML	006-BYTE	SHORT INT	length of name of the columns
IXFACOLN	varying	CHARACTER	name of the columns in the index

One record of this type is specified for each user defined index. This record is located after all of the C records for the table. The following fields are contained in Db2 index records:

IXFARECL
The record length indicator. A 6-byte character representation of an integer value specifying the length, in bytes, of the portion of the PC/IXF record that follows the record length indicator; that is, the total record size minus 6 bytes. Each A record must be sufficiently long to include at least the entire IXFAPPID field.

IXFARECT
The IXF record type, which is set to A for this record, indicating that this is an application record. These records are ignored by programs which do not have particular knowledge about the content and the format of the data implied by the application identifier.

IXFAPPID
The application identifier, which identifies Db2 as the application creating this A record.

IXFAITYP
Specifies that this is subtype "I" of Db2 application records.

IXFADATE
The date on which the file was written, in the form *yyyymmdd*. This field must have the same value as IXFHDATE.

IXFATIME
The time at which the file was written, in the form *hhmmss*. This field must have the same value as IXFHTIME.

IXFANDXL
The length, in bytes, of the index name in the IXFANDXN field.

IXFANDXN
The name of the index.

IXFANCL
The length, in bytes, of the index creator name in the IXFANCN field.

IXFANCN
The name of the index creator.

IXFATABL
The length, in bytes, of the table name in the IXFATABN field.

IXFATABN
The name of the table.

IXFATCL
The length, in bytes, of the table creator name in the IXFATCN field.

IXFATCN
The name of the table creator.

IXFAUNIQ
Specifies the type of index. Valid values are `P` for a primary key, `U` for a unique index, and `D` for a non unique index.

IXFACCNT
Specifies the number of columns in the index definition.

IXFAREVS
Specifies whether reverse scan is allowed on this index. Valid values are `Y` for reverse scan, and `N` for no reverse scan.

IXFAIDXT
Specifies the index type. Valid values are `R` for a regular index, and `C` for a clustered index.

IXFAPCTF
Specifies the percentage of index pages to leave as free. Valid values range from `-1` to `99`. If a value of `-1` or zero is specified, the system default value is used.

IXFAPCTU
Specifies the minimum percentage of index pages that must be free before two index pages can be merged. Valid values range from `00` to `99`.

IXFAEXTI
Reserved for future use.

IXFACNML
The length, in bytes, of the column names in the IXFACOLN field.

IXFACOLN
The names of the columns that are part of this index. Valid values are in the form *+name-name...*, where + specifies an ascending sort on the column, and - specifies a descending sort on the column.

```
DB2 HIERARCHY RECORD

   FIELD NAME     LENGTH     TYPE        COMMENTS
   ----------     --------   ---------   -------------
   IXFARECL       006-BYTE   CHARACTER   record length
   IXFARECT       001-BYTE   CHARACTER   record type = 'A'
   IXFAPPID       012-BYTE   CHARACTER   application identifier = 'DB2 02.00'
   IXFAXTYP       001-BYTE   CHARACTER   application specific data type = 'X'
   IXFADATE       008-BYTE   CHARACTER   date written from the 'H' record
   IXFATIME       006-BYTE   CHARACTER   time written from the 'H' record
   IXFAYCNT       010-BYTE   CHARACTER   'Y' record count for this hierarchy
   IXFAYSTR       010-BYTE   CHARACTER   starting column of this hierarchy
```

One record of this type is used to describe a hierarchy. All subtable records (see the following list) must be located immediately after the hierarchy record, and hierarchy records are located after all of the C records for the table. The following fields are contained in Db2 hierarchy records:

IXFARECL
The record length indicator. A 6-byte character representation of an integer value specifying the length, in bytes, of the portion of the PC/IXF record that follows the record length indicator; that is, the total record size minus 6 bytes. Each A record must be sufficiently long to include at least the entire IXFAPPID field.

IXFARECT
The IXF record type, which is set to A for this record, indicating that this is an application record. These records are ignored by programs which do not have particular knowledge about the content and the format of the data implied by the application identifier.

IXFAPPID
The application identifier, which identifies Db2 as the application creating this A record.

IXFAXTYP
Specifies that this is subtype "X" of Db2 application records.

IXFADATE
The date on which the file was written, in the form *yyyymmdd*. This field must have the same value as IXFHDATE.

IXFATIME
The time at which the file was written, in the form *hhmmss*. This field must have the same value as IXFHTIME.

IXFAYCNT
Specifies the number of subtable records that are expected after this hierarchy record.

IXFAYSTR
Specifies the index of the subtable records at the beginning of the exported data. If export of a hierarchy was started from a non-root subtable, all parent tables of this subtable are exported. The position of this subtable inside of the IXF file is also stored in this field. The first X record represents the column with an index of zero.

```
DB2 SUBTABLE RECORD

   FIELD NAME     LENGTH     TYPE        COMMENTS
   ----------     --------   ---------   -------------
   IXFARECL       006-BYTE   CHARACTER   record length
   IXFARECT       001-BYTE   CHARACTER   record type = 'A'
   IXFAPPID       012-BYTE   CHARACTER   application identifier = 'DB2 02.00'
   IXFAYTYP       001-BYTE   CHARACTER   application specific data type = 'Y'
   IXFADATE       008-BYTE   CHARACTER   date written from the 'H' record
   IXFATIME       006-BYTE   CHARACTER   time written from the 'H' record
   IXFASCHL       003-BYTE   CHARACTER   type schema name length
   IXFASCHN       256-BYTE   CHARACTER   type schema name
   IXFATYPL       003-BYTE   CHARACTER   type name length
   IXFATYPN       256-BYTE   CHARACTER   type name
   IXFATABL       003-BYTE   CHARACTER   table name length
   IXFATABN       256-BYTE   CHARACTER   table name
   IXFAPNDX       010-BYTE   CHARACTER   subtable index of parent table
   IXFASNDX       005-BYTE   CHARACTER   starting column index of current table

   IXFAENDX       005-BYTE   CHARACTER   ending column index of current table
```

One record of this type is used to describe a subtable as part of a hierarchy. All subtable records belonging to a hierarchy must be stored together, and immediately after the corresponding hierarchy record. A subtable is composed of one or more columns, and each column is described in a column record. Each column in a subtable must be described in a consecutive set of C records. The following fields are contained in Db2 subtable records:

IXFARECL
The record length indicator. A 6-byte character representation of an integer value specifying the length, in bytes, of the portion of the PC/IXF record that follows the record length indicator; that is, the total record size minus 6 bytes. Each A record must be sufficiently long to include at least the entire IXFAPPID field.

IXFARECT
The IXF record type, which is set to A for this record, indicating that this is an application record. These records are ignored by programs which do not have particular knowledge about the content and the format of the data implied by the application identifier.

IXFAPPID
The application identifier, which identifies Db2 as the application creating this A record.

IXFAYTYP
Specifies that this is subtype "Y" of Db2 application records.

IXFADATE
The date on which the file was written, in the form *yyyymmdd*. This field must have the same value as IXFHDATE.

IXFATIME
The time at which the file was written, in the form *hhmmss*. This field must have the same value as IXFHTIME.

IXFASCHL
The length, in bytes, of the subtable schema name in the IXFASCHN field.

IXFASCHN
The name of the subtable schema.

IXFATYPL
The length, in bytes, of the subtable name in the IXFATYPN field.

IXFATYPN
The name of the subtable.

IXFATABL
The length, in bytes, of the table name in the IXFATABN field.

IXFATABN
The name of the table.

IXFAPNDX
Subtable record index of the parent subtable. If this subtable is the root of a hierarchy, this field contains the value -1.

IXFASNDX
Starting index of the column records that made up this subtable.

IXFAENDX
Ending index of the column records that made up this subtable.

```
DB2 CONTINUATION RECORD

   FIELD NAME     LENGTH     TYPE        COMMENTS
   ----------     --------   ---------   -------------
   IXFARECL       006-BYTE   CHARACTER   record length
   IXFARECT       001-BYTE   CHARACTER   record type = 'A'
   IXFAPPID       012-BYTE   CHARACTER   application identifier = 'DB2 02.00'
   IXFACTYP       001-BYTE   CHARACTER   application specific data type = 'C'
   IXFADATE       008-BYTE   CHARACTER   date written from the 'H' record
   IXFATIME       006-BYTE   CHARACTER   time written from the 'H' record
   IXFALAST       002-BYTE   SHORT INT   last diskette volume number
   IXFATHIS       002-BYTE   SHORT INT   this diskette volume number
   IXFANEXT       002-BYTE   SHORT INT   next diskette volume number
```

This record is found at the end of each file that is part of a multi-volume IXF file, unless that file is the final volume; it can also be found at the beginning of each file that is part of a multi-volume IXF file, unless that file is the first volume. The purpose of this record is to keep track of file order. The following fields are contained in Db2 continuation records:

IXFARECL
The record length indicator. A 6-byte character representation of an integer value specifying the length, in bytes, of the portion of the PC/IXF record that follows the record length indicator; that is, the total record size minus 6 bytes. Each A record must be sufficiently long to include at least the entire IXFAPPID field.

IXFARECT
The IXF record type, which is set to A for this record, indicating that this is an application record. These records are ignored by programs which do not have particular knowledge about the content and the format of the data implied by the application identifier.

IXFAPPID
The application identifier, which identifies Db2 as the application creating this A record.

IXFACTYP
Specifies that this is subtype "C" of Db2 application records.

IXFADATE
The date on which the file was written, in the form *yyyymmdd*. This field must have the same value as IXFHDATE.

IXFATIME
The time at which the file was written, in the form *hhmmss*. This field must have the same value as IXFHTIME.

IXFALAST
This field is a binary field, in little-endian format. The value should be one less than the value in IXFATHIS.

IXFATHIS
This field is a binary field, in little-endian format. The value in this field on consecutive volumes should also be consecutive. The first volume has a value of 1.

IXFANEXT
This field is a binary field, in little-endian format. The value should be one more than the value in IXFATHIS, unless the record is at the beginning of the file, in which case the value should be zero.

```
DB2 TERMINATE RECORD

   FIELD NAME     LENGTH     TYPE        COMMENTS
   ----------     --------   ---------   -------------
   IXFARECL       006-BYTE   CHARACTER   record length
   IXFARECT       001-BYTE   CHARACTER   record type = 'A'
   IXFAPPID       012-BYTE   CHARACTER   application identifier = 'DB2 02.00'
   IXFAETYP       001-BYTE   CHARACTER   application specific data type = 'E'
   IXFADATE       008-BYTE   CHARACTER   date written from the 'H' record
   IXFATIME       006-BYTE   CHARACTER   time written from the 'H' record
```

This record is the end-of-file marker found at the end of an IXF file. The following fields are contained in Db2 terminate records:

IXFARECL
The record length indicator. A 6-byte character representation of an integer value specifying the length, in bytes, of the portion of the PC/IXF record that follows the record length indicator; that is, the total record size minus 6 bytes. Each A record must be sufficiently long to include at least the entire IXFAPPID field.

IXFARECT
The IXF record type, which is set to A for this record, indicating that this is an application record. These records are ignored by programs which do not have particular knowledge about the content and the format of the data implied by the application identifier.

IXFAPPID
The application identifier, which identifies Db2 as the application creating this A record.

IXFAETYP
Specifies that this is subtype "E" of Db2 application records.

IXFADATE
The date on which the file was written, in the form *yyyymmdd*. This field must have the same value as IXFHDATE.

IXFATIME
The time at which the file was written, in the form *hhmmss*. This field must have the same value as IXFHTIME.

```
DB2 IDENTITY RECORD

   FIELD NAME     LENGTH    TYPE        COMMENTS
   ----------     -------   ---------   -------------
   IXFARECL       06-BYTE   CHARACTER   record length
   IXFARECT       01-BYTE   CHARACTER   record type = 'A'
   IXFAPPID       12-BYTE   CHARACTER   application identifier
   IXFATYPE       01-BYTE   CHARACTER   application specific record type = 'S'
   IXFADATE       08-BYTE   CHARACTER   application record creation date
   IXFATIME       06-BYTE   CHARACTER   application record creation time
   IXFACOLN       06-BYTE   CHARACTER   column number of the identity column
   IXFAITYP       01-BYTE   CHARACTER   generated always ('Y' or 'N')
   IXFASTRT       33-BYTE   CHARACTER   identity START AT value
   IXFAINCR       33-BYTE   CHARACTER   identity INCREMENT BY value
   IXFACACH       10-BYTE   CHARACTER   identity CACHE value
   IXFAMINV       33-BYTE   CHARACTER   identity MINVALUE
   IXFAMAXV       33-BYTE   CHARACTER   identity MAXVALUE
   IXFACYCL       01-BYTE   CHARACTER   identity CYCLE ('Y' or 'N')
   IXFAORDR       01-BYTE   CHARACTER   identity ORDER ('Y' or 'N')
   IXFARMRL       03-BYTE   CHARACTER   identity Remark length
   IXFARMRK      254-BYTE   CHARACTER   identity Remark value
```

The following fields are contained in Db2 identity records:

IXFARECL
The record length indicator. A 6-byte character representation of an integer value specifying the length, in bytes, of the portion of the PC/IXF record that follows the record length indicator; that is, the total record size minus 6 bytes. Each A record must be sufficiently long to include at least the entire IXFAPPID field.

IXFARECT
The IXF record type, which is set to A for this record, indicating that this is an application record. These records are ignored by programs which do not have particular knowledge about the content and the format of the data implied by the application identifier.

IXFAPPID
The application identifier, which identifies Db2 as the application creating this A record.

IXFATYPE
Application specific record type. This field should always have a value of "S".

IXFADATE
The date on which the file was written, in the form *yyyymmdd*. This field must have the same value as IXFHDATE.

IXFATIME
The time at which the file was written, in the form *hhmmss*. This field must have the same value as IXFHTIME.

IXFACOLN
Column number of the identity column in the table.

IXFAITYP
The type of the identity column. A value of "Y" indicates that the identity

column is always GENERATED. All other values are interpreted to mean that the column is of type GENERATED BY DEFAULT.

IXFASTRT
The START AT value for the identity column that was supplied to the CREATE TABLE statement at the time of table creation.

IXFAINCR
The INCREMENT BY value for the identity column that was supplied to the CREATE TABLE statement at the time of table creation.

IXFACACH
The CACHE value for the identity column that was supplied to the CREATE TABLE statement at the time of table creation. A value of "1" corresponds to the NO CACHE option.

IXFAMINV
The MINVALUE for the identity column that was supplied to the CREATE TABLE statement at the time of table creation.

IXFAMAXV
The MAXVALUE for the identity column that was supplied to the CREATE TABLE statement at the time of table creation.

IXFACYCL
The CYCLE value for the identity column that was supplied to the CREATE TABLE statement at the time of table creation. A value of "Y" corresponds to the CYCLE option, any other value corresponds to NO CYCLE.

IXFAORDR
The ORDER value for the identity column that was supplied to the CREATE TABLE statement at the time of table creation. A value of "Y" corresponds to the ORDER option, any other value corresponds to NO ORDER.

IXFARMRL
The length, in bytes, of the remark in IXFARMRK field.

IXFARMRK
This is the user-entered remark associated with the identity column. This is an informational field only. The database manager does not use this field when importing data.

```
DB2 SQLCA RECORD

   FIELD NAME     LENGTH    TYPE        COMMENTS
   ----------     -------   ---------   -------------
   IXFARECL       006-BYTE  CHARACTER   record length
   IXFARECT       001-BYTE  CHARACTER   record type = 'A'
   IXFAPPID       012-BYTE  CHARACTER   application identifier = 'DB2 02.00'
   IXFAITYP       001-BYTE  CHARACTER   application specific data type = 'A'
   IXFADATE       008-BYTE  CHARACTER   date written from the 'H' record
   IXFATIME       006-BYTE  CHARACTER   time written from the 'H' record
   IXFASLCA       136-BYTE  variable    sqlca - SQL communications area
```

One record of this type is used to indicate the IXF file cannot be used to re-create the table in a subsequent import operation. For more information, refer to the message and reason code returned in IXFASLCA.

The following fields are contained in Db2 SQLCA records:

IXFARECL
The record length indicator. A six-byte character representation of an

integer value specifying the length, in bytes, of the portion of the PC/IXF record that follows the record length indicator; that is, the total record size minus six bytes. Each 'A' record must be sufficiently long to include at least the entire IXFAPPID field.

IXFARECT
The IXF record type, which is set to 'A' for this record, indicating that this is an application record. These records are ignored by programs which do not have particular knowledge about the content and the format of the data implied by the application identifier.

IXFAPPID
The application identifier, which identifies Db2 as the application creating this 'A' record.

IXFAITYP
Specifies that this is subtype 'A' of Db2 application records.

IXFADATE
The date on which the file was written, in the form *yyyymmdd*. This field must have the same value as IXFHDATE.

IXFATIME
The time at which the file was written, in the form *hhmmss*. This field must have the same value as IXFHTIME.

IXFASLCA
SQL communications area, which contains the SQL27984W warning message, along with a reason code that explains why the IXF file does not contain all of the information required by the **IMPORT** command to re-create the table.

PC/IXF data types:

Table 7. PC/IXF Data Types

Name	IXFCTYPE Value	Description
BIGINT	492	An 8-byte integer in the form specified by IXFTMFRM. It represents a whole number between -9 223 372 036 854 775 808 and 9 223 372 036 854 775 807. IXFCSBCP and IXFCDBCP are not significant , and should be zero. IXFCLENG is not used, and should contain blanks.
BINARY	912	A fixed-length binary string. The string length is contained in the IXFCLENG field of the column descriptor record, and cannot exceed 254 bytes. The string data is binary data and should not be translated by any transformation program.

Table 7. PC/IXF Data Types (continued)

Name	IXFCTYPE Value	Description
BLOB, CLOB	404, 408	A variable-length character string. The maximum length of the string is contained in the IXFCLENG field of the column descriptor record, and cannot exceed 32 767 bytes. The string itself is preceded by a current length indicator, which is a 4-byte integer specifying the length of the string, in bytes. The string is in the code page indicated by IXFCSBCP. The following applies to BLOBs only: If IXFCSBCP is zero, the string is bit data, and should not be translated by any transformation program. The following applies to CLOBs only: If IXFCDBCP is non-zero, the string can also contain double-byte characters in the code page indicated by IXFCDBCP.
BLOB_LOCATION_ SPECIFIER and DBCLOB_ LOCATION_ SPECIFIER	960, 964, 968	A fixed-length field, which cannot exceed 255 bytes. The LOB Location Specifier (LLS)is located in the code page indicated by IXFCSBCP. If IXFCSBCP is zero, the LLS is bit data and should not be translated by any transformation program. If IXFCDBCP is non-zero, the string can also contain double-byte characters in the code page indicated by IXFCDBCP. Since the length of the LLS is stored in IXFCLENG, the actual length of the original LOB is lost. PC/IXF files with columns of this type should not be used to re-create the LOB field since the LOB will be created with the length of the LLS.
BLOB_FILE, CLOB_FILE, DBCLOB_FILE	916, 920, 924	A fixed-length field containing an SQLFILE structure with the *name_length* and the *name* fields filled in. The length of the structure is contained in the IXFCLENG field of the column descriptor record, and cannot exceed 255 bytes. The file name is in the code page indicated by IXFCSBCP. If IXFCDBCP is non-zero, the file name can also contain double-byte characters in the code page indicated by IXFCDBCP. If IXFCSBCP is zero, the file name is bit data and should not be translated by any transformation program. Since the length of the structure is stored in IXFCLENG, the actual length of the original LOB is lost. IXF files with columns of type BLOB_FILE, CLOB_FILE, or DBCLOB_FILE should not be used to re-create the LOB field, since the LOB will be created with a length of *sql_lobfile_len*.

Table 7. PC/IXF Data Types (continued)

Name	IXFCTYPE Value	Description
CHAR	452	A fixed-length character string. The string length is contained in the IXFCLENG field of the column descriptor record, and cannot exceed 254 bytes. The string is in the code page indicated by IXFCSBCP. If IXFCDBCP is non-zero, the string can also contain double-byte characters in the code page indicated by IXFCDBCP. If IXFCSBCP is zero, the string is bit data and should not be translated by any transformation program.
DATE	384	A point in time in accordance with the Gregorian calendar. Each date is a 10-byte character string in International Standards Organization (ISO) format: *yyyy-mm-dd*. The range of the year part is `0001` to `9999`. The range of the month part is `01` to `12`. The range of the day part is `01` to *n*, where *n* depends on the month, using the usual rules for days of the month and leap year. Leading zeros cannot be omitted from any part. IXFCLENG is not used, and should contain blanks. Valid characters within DATE are invariant in all PC ASCII code pages; therefore, IXFCSBCP and IXFCDBCP are not significant, and should be zero.
DBCLOB	412	A variable-length string of double-byte characters. The IXFCLENG field in the column descriptor record specifies the maximum number of double-byte characters in the string, and cannot exceed 16 383. The string itself is preceded by a current length indicator, which is a 4-byte integer specifying the length of the string in double-byte characters (that is, the value of this integer is one half the length of the string, in bytes). The string is in the DBCS code page, as specified by IXFCDBCP in the C record. Since the string consists of double-byte character data only, IXFCSBCP should be zero. There are no surrounding shift-in or shift-out characters.
DECIMAL	484	A packed decimal number with precision P (as specified by the first three bytes of IXFCLENG in the column descriptor record) and scale S (as specified by the last two bytes of IXFCLENG). The length, in bytes, of a packed decimal number is (P+2)/2. The precision must be an odd number between 1 and 31, inclusive. The packed decimal number is in the internal format specified by IXFTMFRM, where packed decimal for the PC is defined to be the same as packed decimal for the System/370. IXFCSBCP and IXFCDBCP are not significant, and should be zero.

Table 7. PC/IXF Data Types (continued)

Name	IXFCTYPE Value	Description
DECFLOAT	996	A decimal floating-point value is an IEEE 754r number with a decimal point. The position of the decimal point is stored in each decimal floating point value. The range of a decimal floating-point number is a number with either 16 or 34 digits of precision, and an exponent range of 10-383 to 10+384 or 10-6143 to 10+6144, respectively. The storage length of the 16 digit value is 8 bytes, and the storage length of the 34 digit value is 16 bytes.
FLOATING POINT	480	Either a long (8-byte) or short (4-byte) floating point number, depending on whether IXFCLENG is set to eight or to four. The data is in the internal machine form, as specified by IXFTMFRM. IXFCSBCP and IXFCDBCP are not significant, and should be zero. Four-byte floating point is not supported by the database manager.
GRAPHIC	468	A fixed-length string of double-byte characters. The IXFCLENG field in the column descriptor record specifies the number of double-byte characters in the string, and cannot exceed 127. The actual length of the string is twice the value of the IXFCLENG field, in bytes. The string is in the DBCS code page, as specified by IXFCDBCP in the C record. Since the string consists of double-byte character data only, IXFCSBCP should be zero. There are no surrounding shift-in or shift-out characters.
INTEGER	496	A 4-byte integer in the form specified by IXFTMFRM. It represents a whole number between -2 147 483 648 and +2 147 483 647. IXFCSBCP and IXFCDBCP are not significant, and should be zero. IXFCLENG is not used, and should contain blanks.
LONGVARCHAR	456	A variable-length character string. The maximum length of the string is contained in the IXFCLENG field of the column descriptor record, and cannot exceed 32 767 bytes. The string itself is preceded by a current length indicator, which is a 2-byte integer specifying the length of the string, in bytes. The string is in the code page indicated by IXFCSBCP. If IXFCDBCP is non-zero, the string can also contain double-byte characters in the code page indicated by IXFCDBCP. If IXFCSBCP is zero, the string is bit data and should not be translated by any transformation program.

Table 7. PC/IXF Data Types (continued)

Name	IXFCTYPE Value	Description
LONG VARGRAPHIC	472	A variable-length string of double-byte characters. The IXFCLENG field in the column descriptor record specifies the maximum number of double-byte characters for the string, and cannot exceed 16 383. The string itself is preceded by a current length indicator, which is a 2-byte integer specifying the length of the string in double-byte characters (that is, the value of this integer is one half the length of the string, in bytes). The string is in the DBCS code page, as specified by IXFCDBCP in the C record. Since the string consists of double-byte character data only, IXFCSBCP should be zero. There are no surrounding shift-in or shift-out characters.
SMALLINT	500	A 2-byte integer in the form specified by IXFTMFRM. It represents a whole number between -32 768 and +32 767. IXFCSBCP and IXFCDBCP are not significant, and should be zero. IXFCLENG is not used, and should contain blanks.
TIME	388	A point in time in accordance with the 24-hour clock. Each time is an 8-byte character string in ISO format: *hh.mm.ss*. The range of the hour part is `00` to `24`, and the range of the other parts is `00` to `59`. If the hour is `24`, the other parts are `00`. The smallest time is `00.00.00`, and the largest is `24.00.00`. Leading zeros cannot be omitted from any part. IXFCLENG is not used, and should contain blanks. Valid characters within TIME are invariant in all PC ASCII code pages; therefore, IXFCSBCP and IXFCDBCP are not significant, and should be zero.
TIMESTAMP	392	The date and time with fractional second precision. Each time stamp is a character string of the form *yyyy-mm-dd-hh.mm.ss.nnnnnn* (year month day hour minutes seconds fractional seconds). Starting with Version 9.7, the timestamp precision is contained in the IXFCLENG field of the column descriptor record, and cannot exceed 12. before Version 9.7, IXFCLENG is not used, and should contain blanks. Valid characters within TIMESTAMP are invariant in all PC ASCII code pages; therefore, IXFCSBCP and IXFCDBCP are not significant, and should be zero.

Table 7. PC/IXF Data Types (continued)

Name	IXFCTYPE Value	Description
VARBINARY	908	A variable-length binary string. The maximum length of the string, in bytes, is contained in the IXFCLENG field of the column descriptor record, and cannot exceed 32672 bytes. The string itself is preceded by a current length indicator, which is a two-byte integer specifying the length of the string, in bytes. The string data is binary data and should not be translated by any transformation program.
VARCHAR	448	A variable-length character string. The maximum length of the string, in bytes, is contained in the IXFCLENG field of the column descriptor record, and cannot exceed 32672 bytes. The string itself is preceded by a current length indicator, which is a two-byte integer specifying the length of the string, in bytes. The string is in the code page indicated by IXFCSBCP. If IXFCDBCP is non-zero, the string can also contain double-byte characters in the code page indicated by IXFCDBCP. If IXFCSBCP is zero, the string is bit data and should not be translated by any transformation program.
VARGRAPHIC	464	A variable-length string of double-byte characters. The IXFCLENG field in the column descriptor record specifies the maximum number of double-byte characters in the string, and cannot exceed 127. The string itself is preceded by a current length indicator, which is a 2-byte integer specifying the length of the string in double-byte characters (that is, the value of this integer is one half the length of the string, in bytes). The string is in the DBCS code page, as specified by IXFCDBCP in the C record. Since the string consists of double-byte character data only, IXFCSBCP should be zero. There are no surrounding shift-in or shift-out characters.

Not all combinations of IXFCSBCP and IXFCDBCP values for PC/IXF character or graphic columns are valid. A PC/IXF character or graphic column with an invalid (IXFCSBCP,IXFCDBCP) combination is an invalid data type.

Table 8. Valid PC/IXF Data Types

PC/IXF Data Type	Valid (IXFCSBCP,IXFCDBCP) Pairs	Invalid (IXFCSBCP,IXFCDBCP) Pairs
CHAR, VARCHAR, or LONG VARCHAR	(0,0), (x,0), or (x,y)[1]	(0,y)[1]
BLOB	(0,0)	(x,0), (0,y), or (x,y)[1]
CLOB	(x,0), (x,y)[1]	(0,0), (0,y)[1]

Table 8. Valid PC/IXF Data Types (continued)

PC/IXF Data Type	Valid (IXFCSBCP,IXFCDBCP) Pairs	Invalid (IXFCSBCP,IXFCDBCP) Pairs
GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, or DBCLOB	(0,y)[1]	(0,0), (x,0), or (x,y)[1]
BINARY, VARBINARY	(0,0)	(x,0), (0,y), or (x,y)[1]

Note: [1] Neither x nor y is 0.

PC/IXF data type descriptions:

The following table lists the data types and the acceptable forms for each one for the import and load utilities.

Table 9. Acceptable data type forms for the PC/IXF file format

Data type	Form in files created by the export utility	Form acceptable to the import and load utilities
BIGINT	A BIGINT column, identical to the database column, is created.	A column in any numeric type (SMALLINT, INTEGER, BIGINT, DECIMAL, or FLOAT) is accepted. Individual values are rejected if they are not in the range -9 223 372 036 854 775 808 to 9 223 372 036 854 775 807.
BINARY	A PC/IXF BINARY column is created. The database column length, the SBCS CPGID value, and the DBCS CPGID value are copied to the PC/IXF column descriptor record.	A PC/IXF CHAR or VARCHAR column is acceptable if the PC/IXF column single-byte code page values and double-byte code page values are both zero. Otherwise, a BINARY or VARBINARY column is acceptable. If the PC/IXF column is of fixed length, its length must be compatible with the length of the database column. The data is padded on the right with hexadecimal zero characters (x'00'), if necessary.

Table 9. Acceptable data type forms for the PC/IXF file format (continued)

Data type	Form in files created by the export utility	Form acceptable to the import and load utilities
BLOB	A PC/IXF BLOB column is created. The maximum length of the database column, the SBCS CPGID value, and the DBCS CPGID value are copied to the column descriptor record.	A PC/IXF CHAR, VARCHAR, LONG VARCHAR, BLOB, BLOB_FILE, or BLOB_LOCATION_SPECIFIER column is acceptable if: • The database column is marked FOR BIT DATA • The PC/IXF column single-byte code page value equals the SBCS CPGID of the database column, and the PC/IXF column double-byte code page value equals zero, or the DBCS CPGID of the database column. A PC/IXF GRAPHIC, VARGRAPHIC, or LONG VARGRAPHIC BLOB column is also acceptable. If the PC/IXF column is of fixed length, its length must be compatible with the maximum length of the database column.
BOOLEAN	A Boolean value is converted to a SMALLINT: • FALSE is converted to 0. • TRUE is converted to 1. • NULL remains NULL.	A non-zero SMALLINT value is converted to TRUE, zero is converted to FALSE, and NULL remains NULL.
CHAR	A PC/IXF CHAR column is created. The database column length, the SBCS CPGID value, and the DBCS CPGID value are copied to the PC/IXF column descriptor record.	A PC/IXF CHAR, VARCHAR, or LONG VARCHAR column is acceptable if: • The database column is marked FOR BIT DATA • The PC/IXF column single-byte code page value equals the SBCS CPGID of the database column, and the PC/IXF column double-byte code page value equals zero, or the DBCS CPGID of the database column. A PC/IXF GRAPHIC, VARGRAPHIC, or LONG VARGRAPHIC column is also acceptable if the database column is marked FOR BIT DATA. In any case, if the PC/IXF column is of fixed length, its length must be compatible with the length of the database column. If required to match the width of the target column, the character string is leading truncated or padded with trailing spaces (X'20').

Table 9. Acceptable data type forms for the PC/IXF file format (continued)

Data type	Form in files created by the export utility	Form acceptable to the import and load utilities
CLOB	A PC/IXF CLOB column is created. The maximum length of the database column, the SBCS CPGID value, and the DBCS CPGID value are copied to the column descriptor record.	A PC/IXF CHAR, VARCHAR, LONG VARCHAR, CLOB, CLOB_FILE, or CLOB_LOCATION_SPECIFIER column is acceptable if the PC/IXF column single-byte code page value equals the SBCS CPGID of the database column, and the PC/IXF column double-byte code page value equals zero, or the DBCS CPGID of the database column. If the PC/IXF column is of fixed length, its length must be compatible with the maximum length of the database column.
DATE	A DATE column, identical to the database column, is created.	A PC/IXF column of type DATE is the usual input. The import utility also attempts to accept columns in any of the character types, except those with incompatible lengths. The character column in the PC/IXF file must contain dates in a format consistent with the territory code of the target database.
DBCLOB	A PC/IXF DBCLOB column is created. The maximum length of the database column, the SBCS CPGID value, and the DBCS CPGID value are copied to the column descriptor record.	A PC/IXF GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, DBCLOB, DBCLOB_FILE, or DBCLOB_LOCATION_SPECIFIER column is acceptable if the PC/IXF column double-byte code page value equals that of the database column. If the PC/IXF column is of fixed length, its length must be compatible with the maximum length of the database column.
DECIMAL	A DECIMAL column, identical to the database column, is created. The precision and scale of the column is stored in the column descriptor record.	A column in any numeric type (SMALLINT, INTEGER, BIGINT, DECIMAL, or FLOAT) is accepted. Individual values are rejected if they are not in the range of the DECIMAL column into which they are being imported.
DECFLOAT	A DECFLOAT column, identical to the database column, is created. The precision of the column is stored in the column descriptor record.	A column in the following types: SMALLINT, INTEGER, BIGINT (only into DECFLOAT(34)), DECIMAL, FLOAT, REAL, DOUBLE, or DECFLOAT(16) (only into DECFLOAT(34)) is accepted. Other numeric column types are valid for DECFLOAT, but if the value does not fit within the target precision, it is rounded.
FLOAT	A FLOAT column, identical to the database column, is created.	A column in any numeric type (SMALLINT, INTEGER, BIGINT, DECIMAL, or FLOAT) is accepted. All values are within range.

Table 9. Acceptable data type forms for the PC/IXF file format (continued)

Data type	Form in files created by the export utility	Form acceptable to the import and load utilities
GRAPHIC (DBCS only)	A PC/IXF GRAPHIC column is created. The database column length, the SBCS CPGID value, and the DBCS CPGID value are copied to the column descriptor record.	A PC/IXF GRAPHIC, VARGRAPHIC, or LONG VARGRAPHIC column is acceptable if the PC/IXF column double-byte code page value equals that of the database column. If the PC/IXF column is of fixed length, its length must be compatible with the database column length. The data is padded on the right with double-byte spaces (x'8140'), if necessary.
INTEGER	An INTEGER column, identical to the database column, is created.	A column in any numeric type (SMALLINT, INTEGER, BIGINT, DECIMAL, or FLOAT) is accepted. Individual values are rejected if they are not in the range -2 147 483 648 to 2 147 483 647.
LONG VARCHAR	A PC/IXF LONG VARCHAR column is created. The maximum length of the database column, the SBCS CPGID value, and the DBCS CPGID value are copied to the column descriptor record.	A PC/IXF CHAR, VARCHAR, or LONG VARCHAR column is acceptable if: • The database column is marked FOR BIT DATA • The PC/IXF column single-byte code page value equals the SBCS CPGID of the database column, and the PC/IXF column double-byte code page value equals zero, or the DBCS CPGID of the database column. A PC/IXF GRAPHIC, VARGRAPHIC, or LONG VARGRAPHIC column is also acceptable if the database column is marked FOR BIT DATA. In any case, if the PC/IXF column is of fixed length, its length must be compatible with the maximum length of the database column.
LONG VARGRAPHIC (DBCS only)	A PC/IXF LONG VARGRAPHIC column is created. The maximum length of the database column, the SBCS CPGID value, and the DBCS CPGID value are copied to the column descriptor record.	A PC/IXF GRAPHIC, VARGRAPHIC, or LONG VARGRAPHIC column is acceptable if the PC/IXF column double-byte code page value equals that of the database column. If the PC/IXF column is of fixed length, its length must be compatible with the maximum length of the database column.
SMALLINT	A SMALLINT column, identical to the database column, is created.	A column in any numeric type (SMALLINT, INTEGER, BIGINT, DECIMAL, or FLOAT) is accepted. Individual values are rejected if they are not in the range -32 768 to 32 767.

Table 9. Acceptable data type forms for the PC/IXF file format (continued)

Data type	Form in files created by the export utility	Form acceptable to the import and load utilities
TIME	A TIME column, identical to the database column, is created.	A PC/IXF column of type TIME is the usual input. The import utility also attempts to accept columns in any of the character types, except those with incompatible lengths. The character column in the PC/IXF file must contain time data in a format consistent with the territory code of the target database.
TIMESTAMP	A TIMESTAMP column, identical to the database column, is created.	A PC/IXF column of type TIMESTAMP is the usual input. The import utility also attempts to accept columns in any of the character types, except those with incompatible lengths. The character column in the PC/IXF file must contain data in the input format for time stamps.
VARBINARY	A PC/IXF VARBINARY column is created. The database column length, the SBCS CPGID value, and the DBCS CPGID value are copied to the PC/IXF column descriptor record.	A PC/IXF CHAR or VARCHAR column is acceptable if the PC/IXF column single-byte code page values and double-byte code page values are both zero. Otherwise, a BINARY or VARBINARY column is acceptable.
VARCHAR	If the maximum length of the database column is = 32672, a PC/IXF VARCHAR column is created. If the maximum length of the database column is > 32672, a PC/IXF LONG VARCHAR column is created. The maximum length of the database column, the SBCS CPGID value, and the DBCS CPGID value are copied to the column descriptor record.	A PC/IXF CHAR, VARCHAR, or LONG VARCHAR column is acceptable if: • The database column is marked FOR BIT DATA • The PC/IXF column single-byte code page value equals the SBCS CPGID of the database column, and the PC/IXF column double-byte code page value equals zero, or the DBCS CPGID of the database column. A PC/IXF GRAPHIC, VARGRAPHIC, or LONG VARGRAPHIC column is also acceptable if the database column is marked FOR BIT DATA. In any case, if the PC/IXF column is of fixed length, its length must be compatible with the maximum length of the database column. If required to match the maximum length of the target column, the character string is leading truncated. If the ASC *truncate blanks* option is in effect, trailing blanks are stripped from the original or the truncated string.

Table 9. Acceptable data type forms for the PC/IXF file format (continued)

Data type	Form in files created by the export utility	Form acceptable to the import and load utilities
VARGRAPHIC (DBCS only)	If the maximum length of the database column is = 127, a PC/IXF VARGRAPHIC column is created. If the maximum length of the database column is > 127, a PC/IXF LONG VARGRAPHIC column is created. The maximum length of the database column, the SBCS CPGID value, and the DBCS CPGID value are copied to the column descriptor record.	A PC/IXF GRAPHIC, VARGRAPHIC, or LONG VARGRAPHIC column is acceptable if the PC/IXF column double-byte code page value equals that of the database column. If the PC/IXF column is of fixed length, its length must be compatible with the maximum length of the database column.

General rules governing PC/IXF file import into databases:
The database manager import utility applies the following general rules when importing a PC/IXF file in either an SBCS or a DBCS environment:

- The import utility accepts PC/IXF format files only (IXFHID = 'IXF'). IXF files of other formats cannot be imported.
- The import utility rejects a PC/IXF file with more than 1024 columns.
- When exporting to the IXF format, if identifiers exceed the maximum size supported by the IXF format, the export operation succeeds, but the resulting data file cannot be used by a subsequent import operation using the `CREATE` mode. SQL27984W is returned.

  Note: The `CREATE` and `REPLACE_CREATE` options of the IMPORT command are deprecated and might be removed in a future release.
- The value of IXFHSBCP in the PC/IXF H record must equal the SBCS CPGID, or there must be a conversion table between the IXFHSBCP/IXFHDBCP and the SBCS/DBCS CPGID of the target database. The value of IXFHDBCP must equal either '00000', or the DBCS CPGID of the target database. If either of these conditions is not satisfied, the import utility rejects the PC/IXF file, unless the `FORCEIN` option is specified.
- Invalid data types - new tables

  Import of a PC/IXF file into a *new* table is specified by the `CREATE` or the `REPLACE_CREATE` keywords in the IMPORT command. If a PC/IXF column of an invalid data type is selected for import into a new table, the import utility terminates. The entire PC/IXF file is rejected, no table is created, and no data is imported.
- Invalid data types - existing tables

  Import of a PC/IXF file into an *existing* table is specified by the `INSERT`, the `INSERT_UPDATE`, the `REPLACE` or the `REPLACE_CREATE` keywords in the **`IMPORT`** command. If a PC/IXF column of an invalid data type is selected for import into an existing table, one of two actions is possible:
  - If the target table column is nullable, all values for the invalid PC/IXF column are ignored, and the table column values are set to NULL
  - If the target table column is not nullable, the import utility terminates. The entire PC/IXF file is rejected, and no data is imported. The existing table remains unaltered.

- When importing into a new table, nullable PC/IXF columns generate nullable database columns, and not nullable PC/IXF columns generate not nullable database columns.
- A not nullable PC/IXF column can be imported into a nullable database column.
- A nullable PC/IXF column can be imported into a not nullable database column. If a NULL value is encountered in the PC/IXF column, the import utility rejects the values of all columns in the PC/IXF row that contains the NULL value (the entire row is rejected), and processing continues with the next PC/IXF row. That is, no data is imported from a PC/IXF row that contains a NULL value if a target table column (for the NULL) is not nullable.
- Incompatible Columns - New Table

  If, during import to a *new* database table, a PC/IXF column is selected that is incompatible with the target database column, the import utility terminates. The entire PC/IXF file is rejected, no table is created, and no data is imported.

  Note: **IMPORT**'s FORCEIN option extends the scope of compatible columns.
- Incompatible columns - existing table

  If, during import to an *existing* database table, a PC/IXF column is selected that is incompatible with the target database column, one of two actions is possible:
  - If the target table column is nullable, all values for the PC/IXF column are ignored, and the table column values are set to NULL
  - If the target table column is not nullable, the import utility terminates. The entire PC/IXF file is rejected, and no data is imported. The existing table remains unaltered.

  Note: **IMPORT**'s FORCEIN option extends the scope of compatible columns.
- Invalid values

  If, during import, a PC/IXF column value is encountered that is not valid for the target database column, the import utility rejects the values of all columns in the PC/IXF row that contains the invalid value (the entire row is rejected), and processing continues with the next PC/IXF row.

Data type-specific rules governing PC/IXF file import into databases:

- A valid PC/IXF numeric column can be imported into any compatible numeric database column. PC/IXF columns containing 4-byte floating point data are not imported, because this is an invalid data type.
- Database date/time columns can accept values from matching PC/IXF date/time columns (DATE, TIME, and TIMESTAMP), as well as from PC/IXF character columns (CHAR, VARCHAR, and LONG VARCHAR), subject to column length and value compatibility restrictions.
- A valid PC/IXF character column (CHAR, VARCHAR, or LONG VARCHAR) can always be imported into an *existing* database character column marked FOR BIT DATA; otherwise:
  - IXFCSBCP and the SBCS CPGID must agree
  - There must be a conversion table for the IXFCSBCP/IXFCDBCP and the SBCS/DBCS
  - One set must be all zeros (FOR BIT DATA).

  If IXFCSBCP is not zero, the value of IXFCDBCP must equal either zero or the DBCS CPGID of the target database column.

  If either of these conditions is not satisfied, the PC/IXF and database columns are incompatible.

When importing a valid PC/IXF character column into a *new* database table, the value of IXFCSBCP must equal either zero or the SBCS CPGID of the database, or there must be a conversion table. If IXFCSBCP is zero, IXFCDBCP must also be zero (otherwise the PC/IXF column is an invalid data type); IMPORT creates a character column marked FOR BIT DATA in the new table. If IXFCSBCP is not zero, and equals the SBCS CPGID of the database, the value of IXFCDBCP must equal either zero or the DBCS CPGID of the database; in this case, the utility creates a character column in the new table with SBCS and DBCS CPGID values equal to those of the database. If these conditions are not satisfied, the PC/IXF and database columns are incompatible.

The FORCEIN option can be used to override code page equality checks. However, a PC/IXF character column with IXFCSBCP equal to zero and IXFCDBCP not equal to zero is an invalid data type, and cannot be imported, even if FORCEIN is specified.

- A valid PC/IXF graphic column (GRAPHIC, VARGRAPHIC, or LONG VARGRAPHIC) can always be imported into an *existing* database character column marked FOR BIT DATA, but is incompatible with all other database columns. The FORCEIN option can be used to relax this restriction. However, a PC/IXF graphic column with IXFCSBCP not equal to zero, or IXFCDBCP equal to zero, is an invalid data type, and cannot be imported, even if FORCEIN is specified.

  When importing a valid PC/IXF graphic column into a database graphic column, the value of IXFCDBCP must equal the DBCS CPGID of the target database column (that is, the double-byte code pages of the two columns must agree).
- If, during import of a PC/IXF file into an existing database table, a fixed-length string column (CHAR or GRAPHIC) is selected whose length is greater than the maximum length of the target column, the columns are incompatible.
- If, during import of a PC/IXF file into an existing database table, a variable-length string column (VARCHAR, LONG VARCHAR, VARGRAPHIC, or LONG VARGRAPHIC) is selected whose length is greater than the maximum length of the target column, the columns *are* compatible. Individual values are processed according to the compatibility rules governing the database manager INSERT statement, and PC/IXF values which are too long for the target database column are invalid.
- PC/IXF values imported into a fixed-length database *character* column (that is, a CHAR column) are padded on the right with single-byte spaces (0x20), if necessary, to obtain values whose length equals that of the database column. PC/IXF values imported into a fixed-length database *graphic* column (that is, a GRAPHIC column) are padded on the right with double-byte spaces (0x8140), if necessary, to obtain values whose length equals that of the database column.
- Since PC/IXF VARCHAR columns have a maximum length of 254 bytes, a database VARCHAR column of maximum length *n*, with 254 *n* 4001, must be exported into a PC/IXF LONG VARCHAR column of maximum length *n*.
- Although PC/IXF LONG VARCHAR columns have a maximum length of 32 767 bytes, and database LONG VARCHAR columns have a maximum length restriction of 32 700 bytes, PC/IXF LONG VARCHAR columns of length greater than 32 700 bytes (but less than 32 768 bytes) are still valid, and can be imported into database LONG VARCHAR columns, but data might be lost.
- Since PC/IXF VARGRAPHIC columns have a maximum length of 127 bytes, a database VARGRAPHIC column of maximum length *n*, with 127 *n* 2001, must be exported into a PC/IXF LONG VARGRAPHIC column of maximum length *n*.

- Although PC/IXF LONG VARGRAPHIC columns have a maximum length of 16 383 bytes, and database LONG VARGRAPHIC columns have a maximum length restriction of 16 350, PC/IXF LONG VARGRAPHIC columns of length greater than 16 350 bytes (but less than 16 384 bytes) are still valid, and can be imported into database LONG VARGRAPHIC columns, but data might be lost.

Table 10 and Table 11 summarize PC/IXF file import into new or existing database tables without the FORCEIN option.

Table 10. Summary of PC/IXF file import without FORCEIN option-numeric types

PC/IXF COLUMN DATA TYPE	DATABASE COLUMN DATA TYPE					
	SMALL INT	INT	BIGINT	DEC	DFP	FLT
-SMALLINT	N					
	E	E	E	E[a]	E	E
-INTEGER		N				
	E[a]	E	E	E[a]	E	E
-BIGINT			N			
	E[a]	E[a]	E	E[a]	E	E
-DECIMAL				N		
	E[a]	E[a]	E[a]	E[a]	E	E
-DECFLOAT					N	
	E[a]	E[a]	E[a]	E[a]	E	E[a]
-FLOAT						N
	E[a]	E[a]	E[a]	E[a]	E	E

[a] Individual values are rejected if they are out of range for the target numeric data type.

Table 11. Summary of PC/IXF file import without FORCEIN option-character, graphic, and date/time types

PC/IXF COLUMN DATA TYPE	DATABASE COLUMN DATA TYPE						
	(0,0)	(SBCS, 0)[d]	(SBCS, DBCS)[b]	GRAPH[b]	DATE	TIME	TIME STAMP
-(0,0)	N						
	E				E[c]	E[c]	E[c]
-(SBCS,0)		N	N				
	E	E	E		E[c]	E[c]	E[c]
-(SBCS, DBCS)			N		E[c]	E[c]	E[c]
	E		E				
-GRAPHIC				N			
	E			E			
-DATE					N		
					E		
-TIME						N	
						E	
-TIME STAMP							N
							E

[b] Data type is available only in DBCS environments.
[c] Individual values are rejected if they are not valid date or time values.
[d] Data type is not available in DBCS environments.

Note:

1. The table is a matrix of all valid PC/IXF and database manager data types. If a PC/IXF column can be imported into a database column, a letter is displayed in the matrix cell at the intersection of the PC/IXF data type matrix row and the database manager data type matrix column. An 'N' indicates that the utility is creating a new database table (a database column of the indicated data type is created). An 'E' indicates that the utility is importing data to an existing database table (a database column of the indicated data type is a valid target).
2. Character string data types are distinguished by code page attributes. These attributes are shown as an ordered pair (SBCS,DBCS), where:
   - SBCS is either zero or denotes a non-zero value of the single-byte code page attribute of the character data type
   - DBCS is either zero or denotes a non-zero value of the double-byte code page attribute of the character data type.
3. If the table indicates that a PC/IXF character column can be imported into a database character column, the values of their respective code page attribute pairs satisfy the rules governing code page equality.

Differences between PC/IXF and Version 0 System/370 IXF:
The following describes differences between PC/IXF, used by the database manager, and Version 0 System/370 IXF, used by several host database products:

- PC/IXF files are ASCII, rather than EBCDIC oriented. PC/IXF files have significantly expanded code page identification, including new code page identifiers in the H record, and the use of actual code page values in the column descriptor records. There is also a mechanism for marking columns of character data as FOR BIT DATA. FOR BIT DATA columns are of special significance, because transforms which convert a PC/IXF file format to or from any other IXF or database file format cannot perform any code page translation on the values contained in FOR BIT DATA columns.
- Only the machine data form is permitted; that is, the IXFTFORM field must always contain the value `M`. Furthermore, the machine data must be in PC forms; that is, the IXFTMFRM field must contain the value `PC`. This means that integers, floating point numbers, and decimal numbers in data portions of PC/IXF data records must be in PC forms.
- Application (A) records are permitted anywhere after the H record in a PC/IXF file. They are not counted when the value of the IXFHHCNT field is computed.
- Every PC/IXF record begins with a record length indicator. This is a 6-byte character representation of an integer value containing the length, in bytes, of the PC/IXF record not including the record length indicator itself; that is, the total record length minus 6 bytes. The purpose of the record length field is to enable PC programs to identify record boundaries.
- To facilitate the compact storage of variable-length data, and to avoid complex processing when a field is split into multiple records, PC/IXF does not support Version 0 IXF X records, but does support D record identifiers. Whenever a variable-length field or a nullable field is the last field in a data D record, it is not necessary to write the entire maximum length of the field to the PC/IXF file.

FORCEIN option:

The `forcein` file type modifier permits import of a PC/IXF file despite code page differences between data in the PC/IXF file and the target database. It offers additional flexibility in the definition of compatible columns.

General semantics of forcein

The following general semantics apply when using the `forcein` file type modifier in either an SBCS or a DBCS environment:

- The `forcein` file type modifier should be used with caution. It is usually advisable to attempt an import without this option enabled. However, because of the generic nature of the PC/IXF data interchange architecture, some PC/IXF files might contain data types or values that cannot be imported without intervention.
- Import with `forcein` to a *new* table might yield a different result than import to an existing table. An existing table has predefined target data types for each PC/IXF data type.
- When LOB data is exported with the `lobsinfile` file type modifier, and the files move to another client with a different code page, then, unlike other data, the CLOBS and DBCLOBS in the separate files are not converted to the client code page when imported or loaded into a database.

Code page semantics for forcein

The following code page semantics apply when using the `forcein` file type modifier in either an SBCS or a DBCS environment:

- The `forcein` file type modifier disables all import utility code page comparisons.

  This rule applies to code page comparisons at the column level and at the file level as well, when importing to a new or an existing database table. At the column (for example, data type) level, this rule applies only to the following database manager and PC/IXF data types: character (CHAR, VARCHAR, and LONG VARCHAR), and graphic (GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC). The restriction follows from the fact that code page attributes of other data types are not relevant to the interpretation of data type values.
- `forcein` does not disable inspection of code page attributes to determine data types.

  For example, the database manager allows a CHAR column to be declared with the FOR BIT DATA attribute. Such a declaration sets both the SBCS CPGID and the DBCS CPGID of the column to zero; it is the zero value of these CPGIDs that identifies the column values as bit strings (rather than character strings).
- `forcein` does not imply code page translation.

  Values of data types that are sensitive to the `forcein` file type modifier are copied "as is". No code point mappings are employed to account for a change of code page environments. Padding of the imported value with spaces might be necessary in the case of fixed length target columns.
- When data is imported to an *existing* table using `forcein`:
  - The code page value of the target database table and columns always prevails.
  - The code page value of the PC/IXF file and columns is ignored.

  This rule applies whether or not `forcein` is used. The database manager does not permit changes to a database or a column code page value once a database is created.

- When importing to a *new* table using `forcein`:
  - The code page value of the target database prevails.
  - PC/IXF character columns with IXFCSBCP = IXFCDBCP = 0 generate table columns marked FOR BIT DATA.
  - All other PC/IXF character columns generate table character columns with SBCS and DBCS CPGID values equal to those of the database.
  - PC/IXF graphic columns generate table graphic columns with an SBCS CPGID of "undefined", and a DBCS CPGID equal to that of the database (DBCS environment only).

Consider a PC/IXF CHAR column with IXFCSBCP = '00897' and IXFCDBCP = '00301'. This column is to be imported into a database CHAR column whose SBCS CPGID = '00850' and DBCS CPGID = '00000'. Without `forcein`, the utility terminates, and no data is imported, or the PC/IXF column values are ignored, and the database column contains NULLs (if the database column is nullable). With `forcein`, the utility proceeds, ignoring code page incompatibilities. If there are no other data type incompatibilities (such as length, for example), the values of the PC/IXF column are imported "as is", and become available for interpretation under the database column code page environment.

The following two tables show:

- The code page attributes of a column created in a *new* database table when a PC/IXF file data type with specified code page attributes is imported.
- That the import utility rejects PC/IXF data types if they are invalid or incompatible.

Table 12. Summary of Import Utility Code Page Semantics (New Table) for SBCS

CODE PAGE ATTRIBUTES of PC/IXF DATA TYPE	CODE PAGE ATTRIBUTES OF DATABASE TABLE COLUMN	
	Without `forcein`	With `forcein`
(0,0)	(0,0)	(0,0)
(a,0)	(a,0)	(a,0)
(x,0)	reject	(a,0)
(x,y)	reject	(a,0)
(a,y)	reject	(a,0)
(0,y)	reject	(0,0)

Note:

1. This table assumes there is no conversion table between a and x. If there were, items 3 and 4 would work successfully without `forcein`.
2. See the notes for Table 13.

Table 13. Summary of Import Utility Code Page Semantics (New Table) for DBCS

CODE PAGE ATTRIBUTES of PC/IXF DATA TYPE	CODE PAGE ATTRIBUTES OF DATABASE TABLE COLUMN	
	Without `forcein`	With `forcein`
(0,0)	(0,0)	(0,0)
(a,0)	(a,b)	(a,b)
(x,0)	reject	(a,b)

Table 13. Summary of Import Utility Code Page Semantics (New Table) for DBCS (continued)

CODE PAGE ATTRIBUTES of PC/IXF DATA TYPE	CODE PAGE ATTRIBUTES OF DATABASE TABLE COLUMN	
	Without `forcein`	With `forcein`
(a,b)	(a,b)	(a,b)
(x,y)	reject	(a,b)
(a,y)	reject	(a,b)
(x,b)	reject	(a,b)
(0,b)	(-,b)	(-,b)
(0,y)	reject	(-,b)

Note:

1. This table assumes there is no conversion table between a and x.
2. Code page attributes of a PC/IXF data type are shown as an ordered pair, where x represents a non-zero single-byte code page value, and y represents a non-zero double-byte code page value. A '-' represents an undefined code page value.
3. The use of different letters in various code page attribute pairs is deliberate. Different letters imply different values. For example, if a PC/IXF data type is shown as (x,y), and the database column as (a,y), x does not equal a, but the PC/IXF file and the database have the same double-byte code page value y.
4. Only character and graphic data types are affected by the `forcein` code page semantics.
5. It is assumed that the database containing the new table has code page attributes of (a,0); therefore, all character columns in the new table must have code page attributes of either (0,0) or (a,0).

   In a DBCS environment, it is assumed that the database containing the new table has code page attributes of (a,b); therefore, all graphic columns in the new table must have code page attributes of (-,b), and all character columns must have code page attributes of (a,b). The SBCS CPGID is shown as '-', because it is undefined for graphic data types.
6. The data type of the result is determined by the rules described in Data type semantics for `forcein`.
7. The `reject` result is a reflection of the rules for invalid or incompatible data types.

The following two tables show:

- That the import utility accepts PC/IXF data types with various code page attributes into an *existing* table column (the *target* column) having the specified code page attributes.
- That the import utility does not permit a PC/IXF data type with certain code page attributes to be imported into an *existing* table column having the code page attributes shown. The utility rejects PC/IXF data types if they are invalid or incompatible.

Table 14. Summary of Import Utility Code Page Semantics (Existing Table) for SBCS

CODE PAGE ATTRIBUTES OF PC/IXF DATA TYPE	CODE PAGE ATTRIBUTES OF TARGET DATABASE COLUMN	RESULTS OF IMPORT	
		Without `forcein`	With `forcein`
(0,0)	(0,0)	accept	accept
(a,0)	(0,0)	accept	accept
(x,0)	(0,0)	accept	accept
(x,y)	(0,0)	accept	accept
(a,y)	(0,0)	accept	accept
(0,y)	(0,0)	accept	accept
(0,0)	(a,0)	null or reject	accept
(a,0)	(a,0)	accept	accept
(x,0)	(a,0)	null or reject	accept
(x,y)	(a,0)	null or reject	accept
(a,y)	(a,0)	null or reject	accept
(0,y)	(a,0)	null or reject	null or reject

Note:

- This table assumes there is no conversion table between a and x.
- See the notes for Table 12 on page 49.
- The `null` or `reject` result is a reflection of the rules for invalid or incompatible data types.

Table 15. Summary of Import Utility Code Page Semantics (Existing Table) for DBCS

CODE PAGE ATTRIBUTES OF PC/IXF DATA TYPE	CODE PAGE ATTRIBUTES OF TARGET DATABASE COLUMN	RESULTS OF IMPORT	
		Without `forcein`	With `forcein`
(0,0)	(0,0)	accept	accept
(a,0)	(0,0)	accept	accept
(x,0)	(0,0)	accept	accept
(a,b)	(0,0)	accept	accept
(x,y)	(0,0)	accept	accept
(a,y)	(0,0)	accept	accept
(x,b)	(0,0)	accept	accept
(0,b)	(0,0)	accept	accept
(0,y)	(0,0)	accept	accept
(0,0)	(a,b)	null or reject	accept
(a,0)	(a,b)	accept	accept
(x,0)	(a,b)	null or reject	accept
(a,b)	(a,b)	accept	accept
(x,y)	(a,b)	null or reject	accept

Table 15. Summary of Import Utility Code Page Semantics (Existing Table) for DBCS (continued)

CODE PAGE ATTRIBUTES OF PC/IXF DATA TYPE	CODE PAGE ATTRIBUTES OF TARGET DATABASE COLUMN	RESULTS OF IMPORT	
		Without `forcein`	With `forcein`
(a,y)	(a,b)	null or reject	accept
(x,b)	(a,b)	null or reject	accept
(0,b)	(a,b)	null or reject	null or reject
(0,y)	(a,b)	null or reject	null or reject
(0,0)	(-,b)	null or reject	accept
(a,0)	(-,b)	null or reject	null or reject
(x,0)	(-,b)	null or reject	null or reject
(a,b)	(-,b)	null or reject	null or reject
(x,y)	(-,b)	null or reject	null or reject
(a,y)	(-,b)	null or reject	null or reject
(x,b)	(-,b)	null or reject	null or reject
(0,b)	(-,b)	accept	accept
(0,y)	(-,b)	null or reject	accept

Note:

- This table assumes there is no conversion table between a and x.
- See the notes for Table 12 on page 49.
- The `null or reject` result is a reflection of the rules for invalid or incompatible data types.

Data type semantics for forcein

The `forcein` file type modifier permits import of certain PC/IXF columns into target database columns of unequal and otherwise incompatible data types. The following data type semantics apply when using `forcein` in either an SBCS or a DBCS environment (except where noted):

- In SBCS environments, `forcein` permits import of:
  - A PC/IXF BIT data type (IXFCSBCP = 0 = IXFCDBCP for a PC/IXF character column) into a database character column (non-zero SBCS CPGID, and DBCS CPGID = 0); existing tables only
  - A PC/IXF MIXED data type (non-zero IXFCSBCP and IXFCDBCP) into a database character column; both new and existing tables
  - A PC/IXF GRAPHIC data type into a database FOR BIT DATA column (SBCS CPGID = 0 = DBCS CPGID); new tables only (this is always permitted for existing tables).
- The `forcein` file type modifier does not extend the scope of valid PC/IXF data types.

  PC/IXF columns with data types not defined as valid PC/IXF data types are invalid for import with or without `forcein`.
- In DBCS environments, `forcein` permits import of:
  - A PC/IXF BIT data type into a database character column

– A PC/IXF BIT data type into a database graphic column; however, if the PC/IXF BIT column is of fixed length, that length must be even. A fixed length PC/IXF BIT column of odd length is not compatible with a database graphic column. A varying-length PC/IXF BIT column *is* compatible whether its length is odd or even, although an odd-length value from a varying-length column is an invalid value for import into a database graphic column
– A PC/IXF MIXED data type into a database character column.

Table 16 summarizes PC/IXF file import into new or existing database tables with `forcein` specified.

Table 16. Summary of PC/IXF File Import with `forcein`

PC/IXF COLUMN DATA TYPE	DATABASE COLUMN DATA TYPE											
	SMALL INT	INT	BIGINT	DEC	FLT	(0,0)	(SBCS, 0)[e]	(SBCS, DBCS)[b]	GRAPH[b]	DATE	TIME	TIME STAMP
-SMALLINT	N											
	E	E	E	E[a]	E							
-INTEGER		N										
	E[a]	E	E	E[a]	E							
-BIGINT			N									
	E[a]	E[a]	E	E[a]	E							
-DECIMAL				N								
	E[a]	E[a]	E[a]	E[a]	E							
-FLOAT					N							
	E[a]	E[a]	E[a]	E[a]	E							
-(0,0)												
						N						
-(SBCS,0)						E	E w/F	E w/F	E w/F	E[c]	E[c]	E[c]
							N	N				
-(SBCS, DBCS)						E	E	E		E[c]	E[c]	E[c]
							N w/F[d]	N		E[c]	E[c]	E[c]
						E	E w/F	E				
-GRAPHIC						N w/F[d]			N			
						E			E			
-DATE										N		
										E		
-TIME											N	
											E	
-TIME STAMP												N
												E

[a] Individual values are rejected if they are out of range for the target numeric data type.
[b] Data type is available only in DBCS environments.
[c] Individual values are rejected if they are not valid date or time values.
[d] Applies only if the source PC/IXF data type is not supported by the target database.
[e] Data type is not available in DBCS environments.

Note: If a PC/IXF column can be imported into a database column only with `forcein`, the string 'w/F' is displayed together with an 'N' or an 'E'. An 'N' indicates that the utility is creating a new database table; an 'E' indicates that the

utility is importing data to an existing database table. The `forcein` file type modifier affects compatibility of character and graphic data types only.

Unicode considerations for data movement

The export, import, and load utilities are not supported when they are used with a Unicode client connected to a non-Unicode database.

The DEL, ASC, and PC/IXF file formats are supported for a Unicode database, as described in this section.

When exporting from a Unicode database to an ASCII delimited (DEL) file, all character data is converted to the application code page. Both character string and graphic string data are converted to the same SBCS or MBCS code page of the client. This is expected behavior for the export of any database, and cannot be changed, because the entire delimited ASCII file can have only one code page. Therefore, if you export to a delimited ASCII file, only those UCS-2 characters that exist in your application code page will be saved. Other characters are replaced with the default substitution character for the application code page. For UTF-8 clients (code page 1208), there is no data loss, because all UCS-2 characters are supported by UTF-8 clients.

When importing from an ASCII file (DEL or ASC) to a Unicode database, character string data is converted from the application code page to UTF-8, and graphic string data is converted from the application code page to UCS-2. There is no data loss. If you want to import ASCII data that has been saved under a different code page, you should change the data file code page before issuing the IMPORT command. You can specify the code page of the data file by setting the **DB2CODEPAGE** registry variable to the code page of the ASCII data file or by using the `codepage` file type modifier.

The range of valid ASCII delimiters for SBCS and MBCS clients is identical to what is currently supported by Db2 for those clients. The range of valid delimiters for UTF-8 clients is X'01' to X'7F', with the usual restrictions.

When exporting from a Unicode database to a PC/IXF file, character string data is converted to the SBCS/MBCS code page of the client. Graphic string data is not converted, and is stored in UCS-2 (code page 1200). There is no data loss.

When importing from a PC/IXF file to a Unicode database, character string data is assumed to be in the SBCS/MBCS code page stored in the PC/IXF header, and graphic string data is assumed to be in the DBCS code page stored in the PC/IXF header. Character string data is converted by the import utility from the code page specified in the PC/IXF header to the code page of the client, and then from the client code page to UTF-8 (by the INSERT statement). Graphic string data is converted by the import utility from the DBCS code page specified in the PC/IXF header directly to UCS-2 (code page 1200).

The load utility places the data directly into the database and, by default, assumes data in ASC or DEL files to be in the code page of the database. Therefore, by default, no code page conversion takes place for ASCII files. When the code page for the data file has been explicitly specified (using the `codepage` file type modifier), the load utility uses this information to convert from the specified code page to the database code page before loading the data. For PC/IXF files, the load utility always converts from the code pages specified in the IXF header to the database code page (1208 for CHAR, and 1200 for GRAPHIC).

The code page for DBCLOB files is always 1200 for UCS-2. The code page for CLOB files is the same as the code page for the data files being imported, loaded or exported. For example, when loading or importing data using the PC/IXF format, the CLOB file is assumed to be in the code page specified by the PC/IXF header. If the DBCLOB file is in ASC or DEL format, the load utility assumes that CLOB data is in the code page of the database, while the import utility assumes it to be in the code page of the client application.

The `nochecklengths` modifier is always specified for a Unicode database, because:

- Any SBCS can be connected to a database for which there is no DBCS code page
- Character strings in UTF-8 format usually have different lengths than those in client code pages.

Considerations for code page 1394, 1392, and 5488

The import, export and load utilities can be used to transfer data from the Chinese code page GB18030 (code page identifier 1392 and 5488) and the Japanese code page ShiftJISX 0213 (code page identifier 1394) to Db2 Unicode databases. In addition, the export utility can be used to transfer data from Db2 Unicode databases to GB18030 or ShiftJIS X0213 code page data.

For example, the following command will load the Shift JIS X0213 data file `u/jp/user/x0213/data.del` residing on a remotely connected client into MYTABLE:

```
db2 load client from /u/jp/user/x0213/data.del
of del modified by codepage=1394 insert into mytable
```

where MYTABLE is located on a Db2 Unicode database.

Since only connections between a Unicode client and a Unicode server are supported, you need to use either a Unicode client or set the Db2 registry variable **DB2CODEPAGE** to 1208 before using the load, import, or export utilities.

Conversion from code page 1394 to Unicode can result in expansion. For example, a 2-byte character can be stored as two 16-bit Unicode characters in the GRAPHIC columns. You need to ensure the target columns in the Unicode database are wide enough to contain any expanded Unicode byte.

Incompatibilities

For applications connected to a Unicode database, graphic string data is always in UCS-2 (code page 1200). For applications connected to non-Unicode databases, the graphic string data is in the DBCS code page of the application, or not allowed if the application code page is SBCS. For example, when a 932 client is connected to a Japanese non-Unicode database, the graphic string data is in code page 301. For the 932 client applications connected to a Unicode database, the graphic string data is in UCS-2 encoding.

Character set and national language support

The Db2 data movement utilities offer the following national language support:

- The import and the export utilities provide automatic code page conversion from a client code page to the server code page.
- For the load utility, data can be converted from any code page to the server code page by using the `codepage` modifier with DEL and ASC files.

- For all utilities, IXF data is automatically converted from its original code page (as stored in the IXF file) to the server code page.

Unequal code page situations, involving expansion or contraction of the character data, can sometimes occur. For example, Japanese or Traditional-Chinese Extended UNIX Code (EUC) and double-byte character sets (DBCS) might encode different lengths for the same character. Normally, comparison of input data length to target column length is performed before reading in any data. If the input length is greater than the target length, NULLs are inserted into that column if it is nullable. Otherwise, the request is rejected. If the `nochecklengths` file type modifier is specified, no initial comparison is performed, and an attempt is made to import or load the data. If the data is too long after translation is complete, the row is rejected. Otherwise, the data is imported or loaded.

XML data movement

Support for XML data movement is provided by the load, import and export utilities. Support for moving tables that contain XML columns without taking the tables offline is provided by the ADMIN_MOVE_TABLE stored procedure.

Importing XML data

The import utility can be used to insert XML documents into a regular relational table. Only well-formed XML documents can be imported.

Use the XML FROM option of the IMPORT command to specify the location of the XML documents to import. The XMLVALIDATE option specifies how imported documents should be validated. You can select to have the imported XML data validated against a schema specified with the IMPORT command, against a schema identified by a schema location hint inside of the source XML document, or by the schema identified by the XML Data Specifier in the main data file. You can also use the XMLPARSE option to specify how whitespace is handled when the XML document is imported. The `xmlchar` and `xmlgraphic` file type modifiers allow you to specify the encoding characteristics for the imported XML data.

Loading XML data

The load utility offers an efficient way to insert large volumes of XML data into a table. This utility also allows certain options unavailable with the import utility, such as the ability to load from a user-defined cursor.

Like the IMPORT command, with the LOAD command you can specify the location of the XML data to load, validation options for the XML data, and how whitespace is handled. As with IMPORT, you can use the `xmlchar` and `xmlgraphic` file type modifiers to specify the encoding characteristics for the loaded XML data.

Exporting XML data

Data may be exported from tables that include one or more columns with an XML data type. Exported XML data is stored in files separate from the main data file containing the exported relational data. Information about each exported XML document is represented in the main exported data file by an XML data specifier (XDS). The XDS is a string that specifies the name of the system file in which the XML document is stored, the exact location and length of the XML document inside of this file, and the XML schema used to validate the XML document.

You can use the XMLFILE, XML TO, and XMLSAVESCHEMA parameters of the EXPORT command to specify details about how exported XML documents are stored. The `xmlinsepfiles`, `xmlnodeclaration`, `xmlchar`, and `xmlgraphic` file type modifiers allow you to specify further details about the storage location and the encoding of the exported XML data.

Moving tables online

The ADMIN_MOVE_TABLE stored procedure moves the data in an active table into a new table object with the same name, while the data remains online and available for access. The table can include one or more columns with an XML data type. Use an online table move instead of an offline table move if you value availability more than cost, space, move performance, and transaction overhead

You can call the procedure once or multiple times, one call for each operation performed by the procedure. Using multiple calls provides you with additional options, such as cancelling the move or controlling when the target table is taken offline to be updated.

Important considerations for XML data movement

There are a number of restrictions, prerequisites, and reminders to consider when importing or exporting XML data. Review these considerations before importing or exporting XML data.

Keep the following consideration in mind when exporting or importing XML data:

- Exported XML data is always stored separately from the main data file containing exported relational data.
- By default, the export utility writes XML data in Unicode. Use the `xmlchar` file type modifier to have XML data written in the character code page, or use the `xmlgraphic` file type modifier to have XML data written in UTF-16 (the graphic code page) regardless of the application code page.
- XML data can be stored in non-Unicode databases, and the data inserted into an XML column is converted from the database codepage to UTF-8 before insertion. In order to avoid the possible introduction of substitution characters during XML parsing, character data to be inserted should consist only of code points that are part of the database codepage. Setting the `enable_xmlchar` configuration parameter to `no` blocks the insertion of character data types during XML parsing, restricting insertion to data types that do not undergo codepage conversion, such as BIT DATA, BLOB, or XML.
- When importing or loading XML data, the XML data is assumed to be in Unicode unless the XML document to import contains a declaration tag that includes an encoding attribute. You can use the `xmlchar` file type modifier to indicate that XML documents to import are encoded in the character code page, while the `xmlgraphic` file type modifier indicates that XML documents to import are encoded in UTF-16.
- The import and load utilities reject rows that contain documents that are not well-formed.
- If the `XMLVALIDATE` option is specified for the import utility or the load utility, documents that successfully validate against their matching schema are annotated with information about the schema used for validation as they are inserted into a table. Rows containing documents that fail to validate against their matching schema are rejected.
- If the `XMLVALIDATE` option is specified for an import or load utility and multiple XML schemas are used to validate XML documents, you might need to increase

the catalog cache size configuration parameter **catalogcache_sz**. If increasing the value of **catalogcache_sz** is not feasible or possible, you can separate the single import or load command into multiple commands that use fewer schema documents.

- When you export XML data specifying an XQuery statement, you might export Query and XPath Data Model (XDM) instances that are not well-formed XML documents. Exported XML documents that are not well-formed cannot be imported directly into an XML column, because columns defined with the XML data type can contain only complete, well formed XML documents.
- The **CPU_PARALLELISM** setting during a load is reduced to 1 if statistics are being collected.
- An XML load operation requires the use of shared sort memory to proceed. Enable **SHEAPTHRES_SHR** or **INTRA_PARALLEL**, or turn on the connection concentrator. By default, **SHEAPTHRES_SHR** is set, so shared sort memory is available on the default configuration.
- You cannot specify the **SOURCEUSEREXIT** option or **SAVECOUNT** parameter of the LOAD command when loading a table containing an XML column.
- As with LOB files, XML files have to reside on the server side when using the LOAD command.
- When loading XML data to multiple database partitions in a partitioned database environment, the files containing the XML data must be accessible to all database partitions. For example, you can copy the files or create an NFS mount to make the files accessible.

LOB and XML file behavior when importing and exporting

LOB and XML files share certain behaviors and compatibilities that can be used when importing and exporting data.

Export When exporting data, if one or more LOB paths are specified with the LOBS TO option, the export utility will cycle between the paths to write each successive LOB value to the appropriate LOB file. Similarly, if one or more XML paths are specified with the XML TO option, the export utility will cycle between the paths to write each successive XQuery and XPath Data Model (XDM) instance to the appropriate XML file. By default, LOB values and XDM instances are written to the same path to which the exported relational data is written. Unless the LOBSINSEPFILES or XMLINSEPFILES file type modifier is set, both LOB files and XML files can have multiple values concatenated to the same file.

The LOBFILE option provides a means to specify the base name of the LOB files generated by the export utility. Similarly, the XMLFILE option provides a means to specify the base name of the XML files generated by the export utility. The default LOB file base name is the name of the exported data file, with the extension `.lob`. The default XML file base name is the name of the exported data file, with the extension `.xml`. The full name of the exported LOB file or XML file therefore consists of the base name, followed by a number extension that is padded to three digits, and the extension `.lob` or `.xml`.

Import

When importing data, a LOB Location Specifier (LLS) is compatible with an XML target column, and an XML Data Specifier (XDS) is compatible with a LOB target column. If the LOBS FROM option is not specified, the LOB files to import are assumed to reside in the same path as the input

relational data file. Similarly, if the XML FROM option is not specified, the XML files to import are assumed to reside in the same path as the input relational data file.

Export examples

In the following example, all LOB values are written to the file /mypath/t1export.del.001.lob, and all XDM instances are written to the file /mypath/t1export.del.001.xml:

```
EXPORT TO /mypath/t1export.del OF DEL MODIFIED BY LOBSINFILE
SELECT * FROM USER.T1
```

In the following example, the first LOB value is written to the file /lob1/t1export.del.001.lob, the second is written to the file /lob2/t1export.del.002.lob, the third is appended to /lob1/t1export.del.001.lob, the fourth is appended to /lob2/t1export.del.002.lob, and so on:

```
EXPORT TO /mypath/t1export.del OF DEL LOBS TO /lob1,/lob2
MODIFIED BY LOBSINFILE SELECT * FROM USER.T1
```

In the following example, the first XDM instance is written to the file /xml1/xmlbase.001.xml, the second is written to the file /xml2/xmlbase.002.xml, the third is written to /xml1/xmlbase.003.xml, the fourth is written to /xml2/xmlbase.004.xml, and so on:

```
EXPORT TO /mypath/t1export.del OF DEL XML TO /xml1,/xml2 XMLFILE xmlbase
   MODIFIED BY XMLINSEPFILES SELECT * FROM USER.T1
```

Import examples

For a table "mytable" that contains a single XML column, and the following IMPORT command:

```
IMPORT FROM myfile.del of del LOBS FROM /lobpath XML FROM /xmlpath
MODIFIED BY LOBSINFILE XMLCHAR replace into mytable
```

If "myfile.del" contains the following data:

```
mylobfile.001.lob.123.456/
```

The import utility will try to import an XML document from the file /lobpath/mylobfile.001.lob, starting at file offset 123, with its length being 456 bytes.

The file "mylobfile.001.lob" is assumed to be in the LOB path, as opposed to the XML path, since the value is referred to by a LOB Location Specifier (LLS) instead of an XML Data Specifier (XDS).

The document is assumed to be encoded in the character codepage, since the XMLCHAR file type modifier is specified.

XML data specifier

XML data moved with the export, import and load utilities must be stored in files separate from the main data file. The XML data is represented in the main data file with an XML data specifier (XDS).

The XDS is a string represented as an XML tag named "XDS", which has attributes that describe information about the actual XML data in the column; such

information includes the name of the file that contains the actual XML data, and the offset and length of the XML data within that file. The attributes of the XDS are described in the following list.

FIL The name of the file that contains the XML data. You cannot specify a named pipe. Importing or loading XML documents from a named pipe is not supported.

OFF The byte offset of the XML data in the file named by the FIL attribute, where the offset begins from 0.

LEN The length in bytes of the XML data in the file named by the FIL attribute.

SCH The fully qualified SQL identifier of the XML schema that is used to validate this XML document. The schema and name components of the SQL identifier are stored as the "OBJECTSCHEMA" and "OBJECTNAME" values, respectively, of the row in the SYSCAT.XSROBJECTS catalog table that corresponds to this XML schema.

The XDS is interpreted as a character field in the data file and is subject to the parsing behavior for character columns of the file format. For the delimited ASCII file format (DEL), for example, if the character delimiter is present in the XDS, it must be doubled. The special characters <, >, &, ', " within the attribute values must always be escaped. Case-sensitive object names must be placed between " character entities.

Examples

Consider a FIL attribute with the value `abc&"def".del`. To include this XDS in a delimited ASCII file, where the character delimiter is the " character, the " characters are doubled and special characters are escaped.

```
<XDS FIL=""abc&amp;&quot;def&quot;.del"" />
```

The following example shows an XDS as it would appear in a delimited ASCII data file. XML data is stored in the file `xmldocs.xml.001` beginning at byte offset 100 with a length of 300 bytes. Because this XDS is within an ASCII file delimited with double quotation marks, the double quotation marks within the XDS tag itself must be doubled.

```
"<XDS FIL = ""xmldocs.xml.001"" OFF=""100"" LEN=""300"" />"
```

The following example shows the fully qualified SQL identifier `ANTHONY.purchaseOrderTest`. The case-sensitive portion of the identifier must be placed between " character entities in the XDS:

```
"<XDS FIL='/home/db2inst1/xmlload/a.xml' OFF='0' LEN='6758'
   SCH='ANTHONY.&quot;purchaseOrderTest&quot;' />"
```

Query and XPath Data Model

You can access XML data in tables either by using the XQuery functions available in SQL, or by invoking XQuery directly. An instance of the Query and XPath Data Model (XDM) can be well-formed XML documents, sequences of nodes or atomic values, or any combination of nodes and atomic values.

Individual XDM instances can be written to one or more XML files by means of the EXPORT command.

Export utility

Export utility overview

The export utility extracts data using an SQL select or an XQuery statement, and places that information into a file. You can use the output file to move data for a future import or load operation or to make the data accessible for analysis.

The export utility is a relatively simple, yet flexible data movement utility. You can activate it by issuing the **EXPORT** command in the CLP, by calling the ADMIN_CMD stored procedure, or by calling the db2Export API through a user application.

The following items are mandatory for a basic export operation:

- The path and name of the operating system file in which you want to store the exported data
- The format of the data in the input file
  Export supports IXF and DEL data formats for the output files.
- A specification of the data that is to be exported
  For the majority of export operations, you need to provide a SELECT statement that specifies the data to be retrieved for export. When exporting typed tables, you don't need to issue the SELECT statement explicitly; you only need to specify the subtable traverse order within the hierarchy

You can use the export utility with Db2 Connect if you need to move data in IXF format.

Additional options

There are a number of parameters that allow you to customize an export operation. File type modifiers offer many options such as allowing you to change the format of the data, date and time stamps, or code page, or have certain data types written to separate files. Using the **METHOD** parameters, you can specify different column names to be used for the exported data.

You can export from tables that include one or more columns with an XML data type. Use the **XMLFILE**, **XML TO**, and **XMLSAVESCHEMA** parameters to specify details about how those exported documents are stored.

There are a few ways to improve the export utility's performance. As the export utility is an embedded SQL application and does SQL fetches internally, optimizations that apply to SQL operations apply to the export utility as well. Consider taking advantage of large buffer pools, indexing, and sort heaps. In addition, try to minimize device contention on the output files by placing them away from the containers and log devices.

The messages file

The export utility writes error, warning, and informational messages to standard ASCII text message files. For all interfaces except the CLP, you must specify the name of these files in advance with the **MESSAGES** parameter. If you are using the CLP and do not specify a messages file, the export utility writes the messages to standard output.

In IBM Data Studio Version 3.1 or later, you can use the task assistant for exporting data. Task assistants can guide you through the process of setting options, reviewing the automatically generated commands to perform the task, and running these commands. For more details, see Administering databases with task assistants.

Privileges and authorities required to use the export utility

Privileges enable you to create, update, delete, or access database resources. Authority levels provide a method of mapping privileges to higher-level database manager maintenance and utility operations.

Together, privileges and authorities control access to the database manager and its database objects. You can access only those objects for which you have the appropriate authorization: that is, the required privilege or authority.

You must have DATAACCESS authority or the CONTROL or SELECT privilege for each table or view participating in the export operation.

When you are exporting LBAC-protected data, the session authorization ID must be allowed to read the rows or columns that you are trying to export. Protected rows that the session authorization ID is not authorized to read are not exported. If the SELECT statement includes any protected columns that the session authorization ID is not allowed to read, the export utility fails, and an error (SQLSTATE 42512) is returned.

Exporting data

Use the export utility to export data from a database to a file. The file can have one of several external file formats. You can specify the data to be exported by supplying an SQL SELECT statement or by providing hierarchical information for typed tables.

Before you begin

You need DATAACCESS authority, the CONTROL privilege, or the SELECT privilege on each participating table or view to export data from a database

Before running the export utility, you must be connected (or be able to implicitly connect) to the database from which you want to export the data. If implicit connect is enabled, a connection to the default database is established. Utility access to Linux, UNIX, or Windows database servers from Linux, UNIX, or Windows clients must be through a direct connection through the engine and not through a Db2 Connect gateway or loop back environment.

Because the utility issues a COMMIT statement, complete all transactions and release all locks by issuing a COMMIT or a ROLLBACK statement before running the export utility. There is no requirement for applications accessing the table and using separate connections to disconnect.

You cannot export tables with structured type columns.

Procedure

To run the export utility:

- Specify the **EXPORT** command in the command line processor (CLP).
- Call the db2Export application programming interface (API).
- Open the task assistant in IBM Data Studio for the **EXPORT** command.

Example

A simple export operation requires you to specify only a target file, a file format, and a source file for the SELECT statement.

For example:

```
db2 export to filename of ixf select * from table
```

where *filename* is the name of the output file that you want to create and export, `ixf` is the file format, and *table* is the name of the table that contains the data you want to copy.

However, you might also want to specify a messages file to which warning and error messages are written. To do that, add the **MESSAGES** parameter and a message file name (in this case, `msg.txt`). For example:

```
db2 export to filename of ixf messages msgs.txt select * from table
```

Exporting XML data

When exporting XML data, the resulting QDM (XQuery Data Model) instances are written to a file or files separate from the main data file containing exported relational data. This is true even if neither the XMLFILE nor the XML TO option is specified.

By default, exported QDM instances are all concatenated to the same XML file. You can use the `XMLINSEPFILES` file type modifier to specify that each QDM instance be written to a separate file.

The XML data, however, is represented in the main data file with an XML data specifier (XDS). The XDS is a string represented as an XML tag named "XDS", which has attributes that describe information about the actual XML data in the column; such information includes the name of the file that contains the actual XML data, and the offset and length of the XML data within that file.

The destination paths and base names of the exported XML files can be specified with the XML TO and XMLFILE options. If the XML TO or XMLFILE option is specified, the format of the exported XML file names, stored in the FIL attribute of the XDS, is `xmlfilespec.xxx.xml`, where `xmlfilespec` is the value specified for the XMLFILE option, and `xxx` is a sequence number for xml files produced by the export utility. Otherwise, the format of the exported XML file names is: `exportfilename.xxx.xml`, where `exportfilename` is the name of the exported output file specified for the EXPORT command, and `xxx` is a sequence number for xml files produced by the export utility.

By default, exported XML files are written to the path of the exported data file. The default base name for exported XML files is the name of the exported data file, with an appending 3-digit sequence number, and the `.xml` extension.

Examples

For the following examples, imagine a table USER.T1 containing four columns and two rows:

```
C1 INTEGER
C2 XML
C3 VARCHAR(10)
C4 XML
```

Table 17. USER.T1

C1	C2	C3	C4
2	<?xml version="1.0" encoding="UTF-8" ?><note time="12:00:00"><to>You</ to><from> Me</ from><heading>note1</heading> <body>Hello World!</body></ note>	'char1'	<?xml version="1.0" encoding="UTF-8" ?><note time="13:00:00"><to>Him</ to><from> Her</ from><heading>note2</heading>< body>Hello World!</body></note>
4	NULL	'char2'	?xml version="1.0" encoding="UTF-8" ?><note time="14:00:00">to>Us</ to><from> Them</ from><heading>note3</heading> <body>Hello World!</body></note>

Example 1

The following command exports the contents of USER.T1 in Delimited ASCII (DEL) format to the file "/mypath/t1export.del". Because the XML TO and XMLFILE options are not specified, the XML documents contained in columns C2 and C4 are written to the same path as the main exported file "/mypath". The base name for these files is "t1export.del.xml". The XMLSAVESCHEMA option indicates that XML schema information is saved during the export procedure.

```
EXPORT TO /mypath/t1export.del OF DEL XMLSAVESCHEMA SELECT * FROM USER.T1
```

The exported file "/mypath/t1export.del" contains:

```
2,"<XDS FIL='t1export.del.001.xml' OFF='0' LEN='144' />","char1",
"<XDS FIL='t1export.del.001.xml' OFF='144' LEN='145' />"
4,,"char2","<XDS FIL='t1export.del.001.xml' OFF='289'
LEN='145' SCH='S1.SCHEMA_A' />"
```

The exported XML file "/mypath/t1export.del.001.xml" contains:

```
<?xml version="1.0" encoding="UTF-8" ?><note time="12:00:00"><to>You</to>
   <from>Me</from><heading>note1</heading><body>Hello World!</body>
   </note><?xml version="1.0" encoding="UTF-8" ?><note time="13:00:00"><to>Him
   </to><from>Her</from><heading>note2</heading><body>Hello World!
   </body></note><?xml version="1.0" encoding="UTF-8" ?><note time="14:00:00">
   <to>Us</to><from>Them</from>heading>note3</heading><body>
   Hello World!</body></note>
```

Example 2

The following command exports the contents of USER.T1 in DEL format to the file "t1export.del". XML documents contained in columns C2 and C4 are written to the path "/home/user/xmlpath". The XML files are named with the base name "xmldocs", with multiple exported XML documents written to the same XML file. The XMLSAVESCHEMA option indicates that XML schema information is saved during the export procedure.

```
EXPORT TO /mypath/t1export.del OF DEL XML TO /home/user/xmlpath
XMLFILE xmldocs XMLSAVESCHEMA SELECT * FROM USER.T1
```

The exported DEL file "/home/user/t1export.del" contains:

```
2,"<XDS FIL='xmldocs.001.xml' OFF='0' LEN='144' />","char1",
"<XDS FIL='xmldocs.001.xml' OFF='144' LEN='145' />"
4,,"char2","<XDS FIL='xmldocs.001.xml' OFF='289'
LEN='145' SCH='S1.SCHEMA_A' />"
```

The exported XML file "/home/user/xmlpath/xmldocs.001.xml" contains:

```
<?xml version="1.0" encoding="UTF-8" ?><note time="12:00:00"><to>You</to>
   <from>Me</from><heading>note1</heading><body>Hello World!</body>
   </note><?xml version="1.0" encoding="UTF-8" ?><note time="13:00:00">
   <to>Him</to><from>Her</from><heading>note2</heading><body>
   Hello World!</body></note><?xml version="1.0" encoding="UTF-8" ?>
   <note time="14:00:00"><to>Us</to><from>Them</from><heading>
   note3</heading><body>Hello World!</body></note>
```

Example 3

The following command is similar to Example 2, except that each exported XML document is written to a separate XML file.

```
EXPORT TO /mypath/t1export.del OF DEL XML TO /home/user/xmlpath
XMLFILE xmldocs MODIFIED BY XMLINSEPFILES XMLSAVESCHEMA
SELECT * FROM USER.T1
```

The exported file "/mypath/t1export.del" contains:

```
2,"<XDS FIL='xmldocs.001.xml' />","char1","XDS FIL='xmldocs.002.xml' />"
4,,"char2","<XDS FIL='xmldocs.004.xml' SCH='S1.SCHEMA_A' />"
```

The exported XML file "/home/user/xmlpath/xmldocs.001.xml" contains:

```
<?xml version="1.0" encoding="UTF-8" ?><note time="12:00:00"><to>You</to>
   <from>Me</from><heading>note1</heading><body>Hello World!</body>
   </note>
```

The exported XML file "/home/user/xmlpath/xmldocs.002.xml" contains:

```
?xml version="1.0" encoding="UTF-8" ?>note time="13:00:00">to>Him/to>
   from>Her/from>heading>note2/heading>body>Hello World!/body>
   /note>
```

The exported XML file "/home/user/xmlpath/xmldocs.004.xml" contains:

```
<?xml version="1.0" encoding="UTF-8" ?><note time="14:00:00"><to>Us</to>
   <from>Them</from><heading>note3</heading><body>Hello World!</body>
   </note>
```

Example 4

The following command writes the result of an XQuery to an XML file.

```
EXPORT TO /mypath/t1export.del OF DEL XML TO /home/user/xmlpath
XMLFILE xmldocs MODIFIED BY XMLNODECLARATION select
xmlquery( '$m/note/from/text()' passing by ref c4 as "m" returning sequence)
   from USER.T1
```

The exported DEL file "/mypath/t1export.del" contains:

```
"<XDS FIL='xmldocs.001.xml' OFF='0' LEN='3' />"
"<XDS FIL='xmldocs.001.xml' OFF='3' LEN='4' />"
```

The exported XML file "/home/user/xmlpath/xmldocs.001.xml" contains:

```
HerThem
```

Note: The result of this particular XQuery does not produce well-formed XML documents. Therefore, the file exported in this example, could not be directly imported into an XML column.

Export sessions - CLP examples

Example 1

The following example shows how to export information from the STAFF table in

the SAMPLE database (to which the user must be connected) to myfile.ixf, with the output in IXF format. If the database connection is not through Db2 Connect, the index definitions (if any) will be stored in the output file; otherwise, only the data will be stored:

```
db2 export to myfile.ixf of ixf messages msgs.txt select * from staff
```

Example 2
The following example shows how to export the information about employees in Department 20 from the STAFF table in the SAMPLE database (to which the user must be connected) to awards.ixf, with the output in IXF format:

```
db2 export to awards.ixf of ixf messages msgs.txt select * from staff
   where dept = 20
```

Example 3
The following example shows how to export LOBs to a DEL file:

```
db2 export to myfile.del of del lobs to mylobs/
   lobfile lobs1, lobs2 modified by lobsinfile
   select * from emp_photo
```

Example 4
The following example shows how to export LOBs to a DEL file, specifying a second directory for files that might not fit into the first directory:

```
db2 export to myfile.del of del
   lobs to /db2exp1/, /db2exp2/ modified by lobsinfile
   select * from emp_photo
```

Example 5
The following example shows how to export data to a DEL file, using a single quotation mark as the string delimiter, a semicolon as the column delimiter, and a comma as the decimal point. The same convention should be used when importing data back into the database:

```
db2 export to myfile.del of del
   modified by chardel'' coldel; decpt,
   select * from staff
```

LBAC-protected data export considerations

When you export data that is protected by label-based access control (LBAC), the data that is exported is limited to the data that your LBAC credentials allow you to read.

If your LBAC credentials do not allow you to read a row, that row is not exported, but no error is returned. If your LBAC credentials do not allow you to read a column, the export utility fails, and an error (SQLSTATE 42512) is returned.

A value from a column with a data type of DB2SECURITYLABEL is exported as raw data enclosed in character delimiters. If a character delimiter is included in the original data, it is doubled. No other changes are made to the bytes that make up the exported value. This means that a data file that contains DB2SECURITYLABEL data can contain newlines, formfeeds, or other non-printable ASCII characters.

If you want the values of columns with a data type of DB2SECURITYLABEL to be exported in a human-readable form, you can use the SECLABEL_TO_CHAR scalar function in the SELECT statement to convert the values to the security label string format.

Examples

In the following examples, output is in DEL format and is written to the file `myfile.del`. The data is exported from a table named REPS, which was created with the following statement:

```
create table reps (row_label db2securitylabel,
id integer,
name char(30))
security policy data_access_policy
```

This example exports the values of the row_label column in the default format:

```
db2 export to myfile.del of del select * from reps
```

The data file is not very readable in most text editors because the values for the row_label column are likely to contain several ASCII control characters.

The following example exports the values of the row_label column in the security label string format:

```
db2 export to myfile.del of del select SECLABEL_TO_CHAR
(row_label,'DATA_ACCESS_POLICY'), id, name from reps
```

Here is an excerpt of the data file created by the previous example. Notice that the format of the security label is readable:

```
...
"Secret:():Epsilon 37", 2005, "Susan Liu"
"Secret:():(Epsilon 37,Megaphone,Cloverleaf)", 2006, "Johnny Cogent"
"Secret:():(Megaphone,Cloverleaf)", 2007, "Ron Imron"
...
```

Table export considerations

A typical export operation involves the outputting of selected data that is inserted or loaded into existing tables. However, it is also possible to export an entire table for subsequent re-creation using the import utility.

To export a table, you must specify the PC/IXF file format. You can then re-create your saved table (including its indexes) using the import utility in `CREATE` mode. However, some information is not saved to the exported IXF file if any of the following conditions exist:

- The index column names contain hexadecimal values of 0x2B or 0x2D.
- The table contains XML columns.
- The table is multidimensional clustered (MDC).
- The table contains a table partitioning key.
- The index name is longer than 128 bytes due to code page conversion.
- The table is protected.
- The **EXPORT** command contains action strings other than SELECT * FROM *tablename*
- You specify the **METHOD N** parameter for the export utility.

For a list of table attributes that are lost, see "Table import considerations." If any information is not saved, warning SQL27984W is returned when the table is re-created.

Note: Import's `CREATE` mode is being deprecated. Use the **db2look** utility to capture and re-create your tables.

Index information

If the column names specified in the index contain either - or + characters, the index information is not collected, and warning SQL27984W is returned. The export utility completes its processing, and the data exported is unaffected. However, the index information is not saved in the IXF file. As a result, you must create the indexes separately using the **db2look** utility.

Space limitations

The export operation fails if the data that you are exporting exceeds the space available on the file system on which the exported file is created. In this case, you should limit the amount of data selected by specifying conditions on the WHERE clause so that the exported file fits on the target file system. You can run the export utility multiple times to export all of the data.

Tables with other file formats

If you do not export using the IXF file format, the output files do not contain descriptions of the target table, but they contain the record data. To re-create a table and its data, create the target table, then use the load or import utility to populate the table. You can use the **db2look** utility to capture the original table definitions and to generate the corresponding data definition language (DDL).

Typed table export considerations

You can use the Db2 export utility can be used to move data out of typed tables for a later import. Export moves data from one hierarchical structure of typed tables to another by following a specific order and creating an intermediate flat file.

When working with typed tables, the export utility controls what is placed in the output file; specify only the target table name and, optionally, the WHERE clause. You can express subselect statements only by specifying the target table name and the WHERE clause. You cannot specify a fullselect or select-statement when exporting a hierarchy.

Preservation of hierarchies using traverse order

Typed tables can be in a hierarchy. There are several ways you can move data across hierarchies:

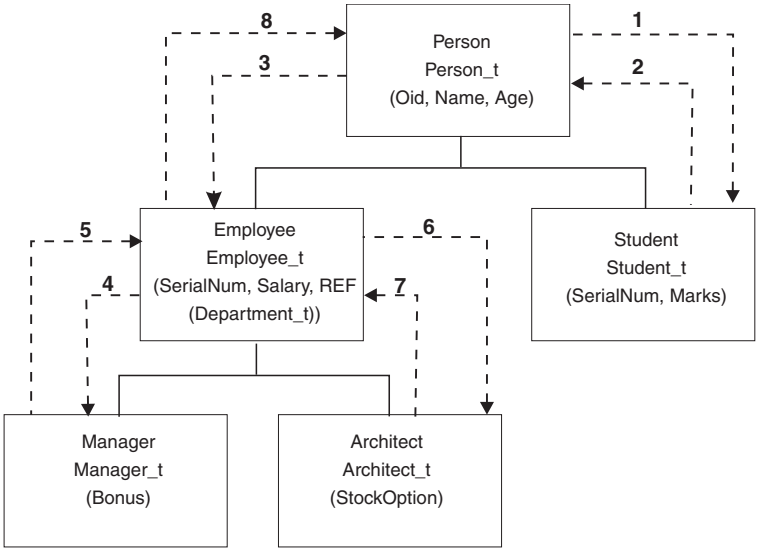
- Movement from one hierarchy to an identical hierarchy
- Movement from one hierarchy to a subsection of a larger hierarchy
- Movement from a subsection of a large hierarchy to a separate hierarchy

Identification of types in a hierarchy is database dependent, meaning that in different databases, the same type has a different identifier. Therefore, when moving data between these databases, a mapping of the same types must be done to ensure that the data is moved correctly.

The mapping used for typed tables is known as the *traverse order*, the order of proceeding top-to-bottom, left-to-right through all of the supertables and subtables in the hierarchy. Before each typed row is written out during an export operation, an identifier is translated into an index value. This index value can be any number from one to the number of relevant types in the hierarchy. Index values are generated by numbering each type when moving through the hierarchy in a specific order-the traverse order. Figure 1 shows a hierarchy with four valid traverse orders:

- Person, Employee, Manager, Architect, Student

- Person, Student, Employee, Manager, Architect
- Person, Employee, Architect, Manager, Student
- Person, Student, Employee, Architect, Manager



(artname: 00002440.gif)
Figure 1. An example of a hierarchy

The traverse order is important when moving data between table hierarchies because it determines where the data is moved in relation to other data. There are two types of traverse order: *default* and *user specified*.

Default traverse order

With the default traverse order, all relevant types refer to all reachable types in the hierarchy from a given starting point in the hierarchy. The default order includes all tables in the hierarchy, and each table is ordered by the scheme used in the OUTER order predicate. For instance, the default traverse order of Figure 1, indicated by the dotted line, would be Person, Student, Employee, Manager, Architect.

The default traverse order behaves differently when used with different file formats. Exporting data to the PC/IXF file format creates a record of all relevant types, their definitions, and relevant tables. The export utility also completes the mapping of an index value to each table. When working with the PC/IXF file format, you should use the default traverse order.

With the ASC or DEL file format, the order in which the typed rows and the typed tables are created could be different, even though the source and target hierarchies might be structurally identical. This results in time differences that the default traverse order identifies when proceeding through the hierarchies. The creation time of each type determines the order used to move through the hierarchy at both the source and the target when using the default traverse order. Ensure that the creation order of each type in both the source and the target hierarchies is identical and that there is structural identity between the source and the target. If these conditions cannot be met, select a user-specified traverse order.

User-specified traverse order

With the user-specified traverse order, you define (in a traverse order list) the relevant types to be used. This order outlines how to traverse the hierarchy and what sub-tables to export, whereas with the default traverse order, all tables in the hierarchy are exported.

Although you determine the starting point and the path down the hierarchy when defining the traverse order, remember that the subtables must be traversed in *pre-order* fashion. Each branch in the hierarchy must be traversed to the bottom before a new branch can be started. The export utility looks for violations of this condition within the specified traverse order. One method of ensuring that the condition is met is to proceed from the top of the hierarchy (or the root table), down the hierarchy (subtables) to the bottom subtable, then back up to its supertable, down to the next "right-most" subtable, then back up to next higher supertable, down to its subtables, and so on.

If you want to control the traverse order through the hierarchies, ensure that the same traverse order is used for both the export and the import utilities.

Example 1

The following examples are based on the hierarchical structure in Figure 1. To export the entire hierarchy, enter the following commands:

```
DB2 CONNECT TO Source_db
DB2 EXPORT TO entire_hierarchy.ixf OF IXF HIERARCHY STARTING Person
```

Note that setting the parameter **HIERARCHY STARTING** to `Person` indicates that the default traverse order starting from the table PERSON.

Example 2

To export the entire hierarchy, but only the data for those people over the age of 20, you would enter the following commands:

```
DB2 CONNECT TO Source_db
   DB2 EXPORT TO entire_hierarchy.del OF DEL HIERARCHY (Person,
      Employee, Manager, Architect, Student) WHERE Age>=20
```

Note that setting the parameter **HIERARCHY** to `Person, Employee, Manager, Architect, Student` indicates a user-specified traverse order.

Identity column export considerations

You can use the export utility to export data from a table containing an identity column. However, the identity column limits your choice of output file format.

If the SELECT statement that you specify for the export operation is of the form SELECT * FROM *tablename* and you do not use the `METHOD` option, exporting identity column properties to IXF files is supported. You can then use the `REPLACE_CREATE` and the `CREATE` options of the **IMPORT** command to re-create the table, including its identity column properties. If you create the exported IXF file from a table containing an identity column of type GENERATED ALWAYS, the only way that you can successfully import the data file is to specify the `identityignore` file type modifier during the import operation. Otherwise, all rows are rejected (SQL3550W is issued).

Note: The `CREATE` and `REPLACE_CREATE` options of the IMPORT command are deprecated and might be removed in a future release.

LOB export considerations

When exporting tables with large object (LOB) columns, the default action is to export a maximum of 32 KB per LOB value and to place it in the same file as the rest of the column data. If you are exporting LOB values that exceed 32 KB, you should have the LOB data written to a separate file to avoid truncation.

To specify that LOB should be written to its own file, use the `lobsinfile` file type modifier. This modifier instructs the export utility to place the LOB data in the directories specified by the `LOBS TO` clause. Using `LOBS TO` or `LOBFILE` implicitly activates the `lobsinfile` file type modifier. By default, LOB values are written to the same path to which the exported relational data is written. If one or more paths are specified with the LOBS TO option, the export utility cycles between the paths to write each successful LOB value to the appropriate LOB file. You can also specify names for the output LOB files using the `LOBFILE` option. If the LOBFILE option is specified, the format of lobfilename is `lobfilespec.xxx.lob`, where `lobfilespec` is the value specified for the LOBFILE option, and `xxx` is a sequence number for LOB files produced by the export utility. Otherwise, lobfilename is of the format: `exportfilename.xxx.lob`, where `exportfilename` is the name of the exported output file specified for the EXPORT command, and `xxx` is a sequence number for LOB files produced by the export utility.

By default, LOBs are written to a single file, but you can also specify that the individual LOBs are to be stored in separate files. The export utility generates a LOB Location Specifier (LLS) to enable the storage of multiple LOBs in one file. The LLS, which is written to the export output file, is a string that indicates where the LOB data is stored within the file. The format of the LLS is `lobfilename.ext.nnn.mmm/`, where `lobfilename.ext` is the name of the file that contains the LOB, `nnn` is the offset of the LOB within the file (measured in bytes), and mmm is the length of the LOB (measured in bytes). For example, an LLS of `db2exp.001.123.456/` indicates that the LOB is located in the file `db2exp.001`, begins at an offset of 123 bytes into the file, and is 456 bytes long. If the indicated size in the LLS is 0, the LOB is considered to have a length of 0. If the length is -1, the LOB is considered to be NULL and the offset and file name are ignored.

If you don't want individual LOB data concatenated to the same file, use the `lobsinsepfiles` file type modifier to write each LOB to a separate file.

Note: The IXF file format does not store the LOB options of the column, such as whether or not the LOB column is logged. This means that the import utility cannot re-create a table containing a LOB column that is defined to be 1 GB or larger.

Example 1

The following example shows how to export LOBs (where the exported LOB files have the specified base name `lobs1`) to a DEL file:

```
db2 export to myfile.del of del lobs to mylobs/
   lobfile lobs1 modified by lobsinfile
   select * from emp_photo
```

Example 2

The following example shows how to export LOBs to a DEL file, where each LOB value is written to a separate file and lobfiles are written to two directories:

```
db2 export to myfile.del of del
lobs to /db2exp1/, /db2exp2/ modified by lobsinfile
select * from emp_photo
```

Import utility

Import overview

The import utility populates a table, typed table, or view with data using an SQL INSERT statement. If the table or view receiving the imported data already contains data, the input data can either replace or be appended to the existing data.

Like export, import is a relatively simple data movement utility. It can be activated by issuing CLP commands, by calling the ADMIN_CMD stored procedure, or by calling its API, db2Import, through a user application.

There are a number of data formats that import supports, as well as features that can be used with import:

- Import supports IXF, ASC, and DEL data formats.
- Import can be used with file type modifiers to customize the import operation.
- Import can be used to move hierarchical data and typed tables.
- Import logs all activity, updates indexes, verifies constraints, and fires triggers.
- Import allows you to specify the names of the columns within the table or view into which the data is to be inserted.
- Import can be used with Db2 Connect.

Import modes

Import has five modes which determine the method in which the data is imported. The first three, `INSERT`, `INSERT_UPDATE`, and `REPLACE` are used when the target tables already exist. All three support IXF, ASC, and DEL data formats. However, only `INSERT` and `INSERT_UPDATE` can be used with nicknames.

Table 18. Overview of INSERT, INSERT_UPDATE, and REPLACE import modes

Mode	Best practice usage
INSERT	Inserts input data into target table without changing existing data
INSERT_UPDATE	Updates rows with matching primary key values with values of input rows Where there's no matching row, inserts imported row into the table
REPLACE	Deletes all existing data and inserts imported data, while keeping table and index definitions

The other two modes, `REPLACE_CREATE` and `CREATE`, are used when the target tables do not exist. They can only be used with input files in the PC/IXF format, which contains a structured description of the table that is to be created. Imports cannot be performed in these modes if the object table has any dependents other than itself.

Note: Import's `CREATE` and `REPLACE_CREATE` modes are being deprecated. Use the **db2look** utility instead.

Table 19. Overview of REPLACE_CREATE and CREATE import modes

Mode	Best practice usage
REPLACE_CREATE	Deletes all existing data and inserts imported data, while keeping table and index definitions Creates target table and index if they don't exist
CREATE	Creates target table and index Can specify the name of the table space where the new table is created

In IBM Data Studio Version 3.1 or later, you can use the task assistant for importing data. Task assistants can guide you through the process of setting options, reviewing the automatically generated commands to perform the task, and running these commands. For more details, see Administering databases with task assistants.

How import works

The number of steps and the amount of time required for an import depend on the amount of data being moved and the options that you specify. An import operation follows these steps:

1. Locking tables
   Import acquires either an exclusive (X) lock or a nonexclusive (IX) lock on existing target tables, depending on whether you allow concurrent access to the table.
2. Locating and retrieving data
   Import uses the FROM clause to locate the input data. If your command indicates that XML or LOB data is present, import will locate this data.
3. Inserting data
   Import either replaces existing data or adds new rows of data to the table.
4. Checking constraints and firing triggers
   As the data is written, import ensures that each inserted row complies with the constraints defined on the target table. Information about rejected rows is written to the messages file. Import also fires existing triggers.
5. Committing the operation
   Import saves the changes made and releases the locks on the target table. You can also specify that periodic take place during the import.

The following items are mandatory for a basic import operation:

- The path and the name of the input file
- The name or alias of the target table or view
- The format of the data in the input file
- The method by which the data is to be imported
- The traverse order, when importing hierarchical data
- The subtable list, when importing typed tables

Additional options

There are a number of options that allow you to customize an import operation. You can specify file type modifiers in the MODIFIED BY clause to change the format of the data, tell the import utility what to do with the data, and to improve performance.

The import utility, by default, does not perform commits until the end of a successful import, except in the case of some `ALLOW WRITE ACCESS` imports. This improves the speed of an import, but for the sake of concurrency, restartability, and active log space considerations, it might be preferable to specify that commits take place during the import. One way of doing so is to set the **`COMMITCOUNT`** parameter to "automatic," which instructs import to internally determine when it should perform a commit. Alternatively, you can set **`COMMITCOUNT`** to a specific number, which instructs import to perform a commit once that specified number of records has been imported.

There are a few ways to improve import's performance. As the import utility is an embedded SQL application and does SQL fetches internally, optimizations that apply to SQL operations apply to import as well. You can use the `compound` file type modifier to perform a specified number of rows to insert at a time, rather than the default row-by-row insertion. If you anticipate that a large number of warnings will be generated (and, therefore, slow down the operation) during the import, you can also specify the `norowwarnings` file type modifier to suppress warnings about rejected rows.

Messages file

During an import, standard ASCII text message files are written to contain the error, warning, and informational messages associated with that operation. If the utility is invoked through the application programming interface (API) db2Import, you must specify the name of these files in advance with the **`MESSAGES`** parameter, otherwise it is optional. The messages file is a convenient way of monitoring the progress of an import, as you can access is while the import is in progress. In the event of a failed import operation, message files can be used to determine a restarting point by indicating the last row that was successfully imported.

Note: If the volume of output messages generated by an import operation against a remote database exceeds 60 KB, the utility will keep the first 30 KB and the last 30 KB.

Privileges and authorities required to use import

Privileges enable users to create or access database resources. Authority levels provide a method of grouping privileges and higher-level database manager maintenance and utility operations. Together, these act to control access to the database manager and its database objects.

Users can access only those objects for which they have the appropriate authorization; that is, the required privilege or authority.

With DATAACCESS authority, you can perform any type of import operation. The following table lists the other authorities on each participating table, view or nickname that enable you to perform the corresponding type of import.

Table 20. Authorities required to perform import operations

Mode	Required authority
INSERT	CONTROL or INSERT and SELECT
INSERT_UPDATE	CONTROL or INSERT, SELECT, UPDATE, and DELETE

Table 20. Authorities required to perform import operations (continued)

Mode	Required authority
REPLACE	CONTROL or INSERT, SELECT, and DELETE
REPLACE_CREATE	When the target table exists: CONTROL or INSERT, SELECT, and DELETE When the target table doesn't exist: CREATETAB (on the database), USE (on the table space), and when the schema does not exist: IMPLICIT_SCHEMA (on the database), or when the schema exists: CREATEIN (on the schema)
CREATE	CREATETAB (on the database), USE (on the table space), and when the schema does not exist: IMPLICIT_SCHEMA (on the database), or when the schema exists: CREATEIN (on the schema)

Note: The **CREATE** and **REPLACE_CREATE** options of the **IMPORT** command are deprecated and might be removed in a future release.
As well, to use the **REPLACE** or **REPLACE_CREATE** option on a table, the session authorization ID must have the authority to drop the table.

If you want to import to a hierarchy, the required authority also depends on the mode. For existing hierarchies, CONTROL privilege on every subtable in the hierarchy is sufficient for a **REPLACE** operation. For hierarchies that don't exist, CONTROL privilege on every subtable in the hierarchy, along with CREATETAB and USE, is sufficient for a **REPLACE_CREATE** operation.

In addition, there a few considerations for importing into tables with label-based access control (LBAC) security labels defined on them. To import data into a table that has protected columns, the session authorization ID must have LBAC credentials that allow write access to all protected columns in the table. To import data into a table that has protected rows, the session authorization ID must have been granted a security label for write access that is part of the security policy protecting the table.

Importing data

The import utility inserts data from an external file with a supported file format into a table, hierarchy, view, or nickname.

The load utility is a faster alternative, but the load utility does not support loading data at the hierarchy level.

Before you begin

Before invoking the import utility, you must be connected to (or be able to implicitly connect to) the database into which you want to import the data. If implicit connect is enabled, a connection to the default database is established.

Utility access to Db2 for Linux, UNIX, or Windows database servers from Db2 for Linux, UNIX, or Windows clients must be a direct connection through the engine. Utility access cannot be through a Db2 Connect gateway or loop back environment.

Since the utility issues a COMMIT or a ROLLBACK statement, complete all transactions and release all locks by issuing a COMMIT statement or a ROLLBACK operation before invoking import.

Note: The **CREATE** and **REPLACE_CREATE** parameters of the **IMPORT** command are deprecated and might be removed in a future release.

Restrictions

The following restrictions apply to the import utility:

- If the existing table is a parent table containing a primary key that is referenced by a foreign key in a dependent table, its data cannot be replaced, only appended to.
- You cannot perform an import replace operation into an underlying table of a materialized query table defined in refresh immediate mode.
- You cannot import data into a system table, a summary table, or a table with a structured type column.
- You cannot import data into declared temporary tables.
- Views cannot be created through the import utility.
- Referential constraints and foreign key definitions are not preserved when creating tables from PC/IXF files. (Primary key definitions *are* preserved if the data was previously exported by using `SELECT *`.)
- Because the import utility generates its own SQL statements, the maximum statement size of 2 MB might, in some cases, be exceeded.
- You cannot re-create a partitioned table or a multidimensional clustered table (MDC) by using the **CREATE** or **REPLACE_CREATE** import parameters.
- You cannot re-create tables containing XML columns.
- You cannot import encrypted data.
- The import replace operation does not honor the Not Logged Initially clause. The **REPLACE** parameter for the **IMPORT** command does not honor the NOT LOGGED INITIALLY (NLI) clause for the CREATE TABLE statement clause or the ACTIVATE NOT LOGGED INITIALLY clause for the ALTER TABLE statement. If an import with the **REPLACE** action is performed within the same transaction as a CREATE TABLE or ALTER TABLE statement where the NLI clause is invoked, the import does not honor the NLI clause. In this scenario, all inserts are logged.

  Workaround 1: Delete the contents of the table by using the DELETE statement, then invoke the import with INSERT statement.

  Workaround 2: Drop the table and re-create it, then invoke the import with INSERT statement.

The following limitation applies to the import utility: If the volume of output messages generated by an import operation against a remote database exceeds 60 KB, the utility keeps the first 30 KB and the last 30 KB.

Procedure

To invoke the import utility:

- Issue an **IMPORT** command in the command line processor (CLP).
- Call the db2Import application programming interface (API) from a client application.
- Open the task assistant in IBM Data Studio for the **IMPORT** command.

Example

A simple import operation requires you to specify only an input file, a file format, an import mode, and a target table (or the name of the table that is to be created).

For example, to import data from the CLP, enter the **IMPORT** command:

```
db2 import from filename of fileformat import_mode into table
```

where *filename* is the name of the input file that contains the data you want to import, *fileformat* is the file format, *import_mode* is the mode, and *table* is the name of the table that you want to insert the data into.

However, you might also want to specify a messages file to which warning and error messages are written. To do that, add the **MESSAGES** parameter and a message file name. For example:

```
db2 import from filename of fileformat messages messagefile import_mode into table
```

Importing XML data

The import utility can be used to import XML data into an XML table column using either the table name or a nickname for a Db2 source data object.

When importing data into an XML table column, you can use the XML FROM option to specify the paths of the input XML data file or files. For example, for an XML file "/home/user/xmlpath/xmldocs.001.xml" that had previously been exported, the following command could be used to import the data back into the table.

```
IMPORT FROM t1export.del OF DEL XML FROM /home/user/xmlpath INSERT INTO USER.T1
```

Validating inserted documents against schemas

The XMLVALIDATE option allows XML documents to be validated against XML schemas as they are imported. In the following example, incoming XML documents are validated against schema information that was saved when the XML documents were exported:

```
IMPORT FROM t1export.del OF DEL XML FROM /home/user/xmlpath XMLVALIDATE
     USING XDS INSERT INTO USER.T1
```

Specifying parse options

You can use the XMLPARSE option to specify whether whitespace in the imported XML documents is preserved or stripped. In the following example, all imported XML documents are validated against XML schema information that was saved when the XML documents were exported, and these documents are parsed with whitespace preserved.

```
IMPORT FROM t1export.del OF DEL XML FROM /home/user/xmlpath XMLPARSE PRESERVE
WHITESPACE XMLVALIDATE USING XDS INSERT INTO USER.T1
```

Import sessions - CLP examples

Example 1

The following example shows how to import information from`myfile.ixf` to the STAFF table:

```
db2 import from myfile.ixf of ixf messages msg.txt insert into staff

SQL3150N  The H record in the PC/IXF file has product "Db2    01.00", date
"19970220", and time "140848".
```

```
SQL3153N  The T record in the PC/IXF file has name "myfile",
qualifier "        ", and source "            ".

SQL3109N  The utility is beginning to load data from file "myfile".

SQL3110N  The utility has completed processing.  "58" rows were read from the
input file.

SQL3221W  ...Begin COMMIT WORK. Input Record Count = "58".

SQL3222W  ...COMMIT of any database changes was successful.

SQL3149N  "58" rows were processed from the input file.  "58" rows were
successfully inserted into the table.  "0" rows were rejected.
```

Example 2

The following example shows how to import into a table that has identity columns:

TABLE1 has 4 columns:

- C1 VARCHAR(30)
- C2 INT GENERATED BY DEFAULT AS IDENTITY
- C3 DECIMAL(7,2)
- C4 CHAR(1)

TABLE2 is the same as TABLE1, except that C2 is a GENERATED ALWAYS identity column.

Data records in DATAFILE1 (DEL format):

```
"Liszt"
"Hummel",,187.43, H
"Grieg",100, 66.34, G
"Satie",101, 818.23, I
```

Data records in DATAFILE2 (DEL format):

```
"Liszt", 74.49, A
"Hummel", 0.01, H
"Grieg", 66.34, G
"Satie", 818.23, I
```

The following command generates identity values for rows 1 and 2, since no identity values are supplied in DATAFILE1 for those rows. Rows 3 and 4, however, are assigned the user-supplied identity values of 100 and 101, respectively.

```
db2 import from datafile1.del of del replace into table1
```

To import DATAFILE1 into TABLE1 so that identity values are generated for all rows, issue one of the following commands:

```
db2 import from datafile1.del of del method P(1, 3, 4)
   replace into table1 (c1, c3, c4)
db2 import from datafile1.del of del modified by identityignore
   replace into table1
```

To import DATAFILE2 into TABLE1 so that identity values are generated for each row, issue one of the following commands:

```
db2 import from datafile2.del of del replace into table1 (c1, c3, c4)
db2 import from datafile2.del of del modified by identitymissing
   replace into table1
```

If DATAFILE1 is imported into TABLE2 without using any of the identity-related file type modifiers, rows 1 and 2 will be inserted, but rows 3 and 4 will be rejected, because they supply their own non-NULL values, and the identity column is GENERATED ALWAYS.

Example 3

The following example shows how to import into a table that has null indicators:

TABLE1 has 5 columns:

- COL1 VARCHAR 20 NOT NULL WITH DEFAULT
- COL2 SMALLINT
- COL3 CHAR 4
- COL4 CHAR 2 NOT NULL WITH DEFAULT
- COL5 CHAR 2 NOT NULL

ASCFILE1 has 6 elements:

- ELE1 positions 01 to 20
- ELE2 positions 21 to 22
- ELE5 positions 23 to 23
- ELE3 positions 24 to 27
- ELE4 positions 28 to 31
- ELE6 positions 32 to 32
- ELE6 positions 33 to 40

Data Records:

```
1...5....10...15...20...25...30...35...40
Test data 1         XXN 123abcdN
Test data 2 and 3   QQY    wxyzN
Test data 4,5 and 6 WWN6789    Y
```

The following command imports records from ASCFILE1 into TABLE1:

```
db2 import from ascfile1 of asc
method L (1 20, 21 22, 24 27, 28 31)
null indicators (0, 0, 23, 32)
insert into table1 (col1, col5, col2, col3)
```

Note:

1. Because COL4 is not provided in the input file, it will be inserted into TABLE1 with its default value (it is defined NOT NULL WITH DEFAULT).
2. Positions 23 and 32 are used to indicate whether COL2 and COL3 of TABLE1 will be loaded NULL for a given row. If there is a `Y` in the column's null indicator position for a given record, the column will be NULL. If there is an `N`, the data values in the column's data positions of the input record (as defined in L(.....)) are used as the source of column data for the row. In this example, neither column in row 1 is NULL; COL2 in row 2 is NULL; and COL3 in row 3 is NULL.
3. In this example, the NULL INDICATORS for COL1 and COL5 are specified as 0 (zero), indicating that the data is not nullable.

4. The NULL INDICATOR for a given column can be anywhere in the input record, but the position must be specified, and the `Y` or `N` values must be supplied.

Imported table re-creation

You can use the import utility's `CREATE` mode to re-create a table that was saved through the export utility. However, there are a number of limitations on the process, as many of the input table's attributes are not retained.

For import to be able to re-create the table, the export operation must meet some requirements. The original table must have been exported to an IXF file. If you export files with DEL or ASC file formats, the output files do not contain descriptions of the target table, but they contain the record data. To re-create a table with data stored in these file formats, create the target table, then use the load or import utility to populate the table from these files. You can use the **db2look** utility to capture the original table definitions and to generate the corresponding data definition language (DDL). As well, the SELECT statement used during the export can only contain certain action strings. For example, no column names can be used in the SELECT clause and only SELECT * is permitted.

Note: Import's `CREATE` mode is being deprecated. Use the **db2look** utility to capture and re-create your tables.

Retained attributes

The re-created table will retain the following attributes of the original table:

- The primary key name, and definition
- Column information, including:
  - Column name
  - Column data type, including user-defined distinct types, which are preserved as their base type
  - Identity properties
  - Lengths (except for lob_file types)
  - Code page (if applicable)
  - Identity options
  - Whether the column is defined as nullable or not nullable
  - Default values for constants, if any, but not other types of default values
- Index information, including:
  - Index name
  - Index creator name
  - Column names, and whether each column is sorted in ascending or descending order
  - Whether the index is defined as unique
  - Whether the index is clustered
  - Whether the index allows reverse scans
  - PCTFREE values
  - MINPCTUSED values

Note: No index information is retained if the column names in the index contain the characters - or +, in which case SQL27984W is returned.

Lost attributes

The re-created table does not retain several attributes of the original table, including:

- Whether the source was a normal table, a materialized query table (MQT), a view, or a set of columns from any or all of these sources
- Unique constraints and other types of constraints or triggers (not including primary key constraints)
- Table information, including:
  - MQT definition (if applicable)
  - MQT options (if applicable)
  - Table space options; however, this information can be specified through the **IMPORT** command
  - Multidimensional clustering (MDC) dimensions
  - Partitioned table dimensions
  - Table partitioning key
  - NOT LOGGED INITIALLY property
  - Check constraints
  - Table code page
  - Protected table properties
  - Table or value compression options
- Column information, including:
  - Any default value except constant values
  - LOB options (if any)
  - XML properties
  - References clause of the CREATE TABLE statement (if any)
  - Referential constraints (if any)
  - Check constraints (if any)
  - Generated column options (if any)
  - Columns dependent on database scope sequences
  - Implicitly hidden property
- Index information, including:
  - INCLUDE columns (if any)
  - Index name, if the index is a primary key index
  - Descending order of keys, if the index is a primary key index (ascending is the default)
  - Index column names that contain hexadecimal values of 0x2B or 0x2D
  - Index names that contain more than 128 bytes after code page conversion
  - PCTFREE2 value
  - Unique constraints

Note: This list is not exhaustive, use with care.

If the import fails and SQL3311N is returned, you can still re-create the table using the file type modifier `forcecreate`. This modifier allows you to create the table with missing or limited information.

Typed table import considerations

The import utility can be used to move data both from and into typed tables while preserving the data's preexisting hierarchy. If desired, import can also be used to create the table hierarchy and the type hierarchy.

The movement of data from one hierarchical structure of typed tables to another is done through a specific traverse order and the creation of an intermediate flat file during an export operation. In turn, the import utility controls the size and the placement of the hierarchy being moved, using the **CREATE**, **INTO** *table-name*, **UNDER**, and **AS ROOT TABLE** parameters. As well, import determines what is placed in the target database. For example, it can specify an attributes list at the end of each subtable name to restrict the attributes that are moved to the target database. If no attributes list is used, all of the columns in each subtable are moved.

Table re-creation

The type of import you are able to perform depends on the file format of the input file. When working with ASC or DEL data, the target table or hierarchy must exist before the data can be imported. However, data from a PC/IXF file can be imported even if the table or hierarchy does not already exist if you specify an import **CREATE** operation. It must be noted that if the **CREATE** option is specified, import cannot alter subtable definitions.

Traverse order

The traverse order contained in the input file enables the hierarchies in the data to be maintained. Therefore, the same traverse order must be used when invoking the export utility and the import utility.

For the PC/IXF file format, one need only specify the target subtable name, and use the default traverse order stored in the file.

When using options other than **CREATE** with typed tables, the traverse order list enables one to specify the traverse order. This user-specified traverse order must match the one used during the export operation. The import utility guarantees the accurate movement of data to the target database given the following:

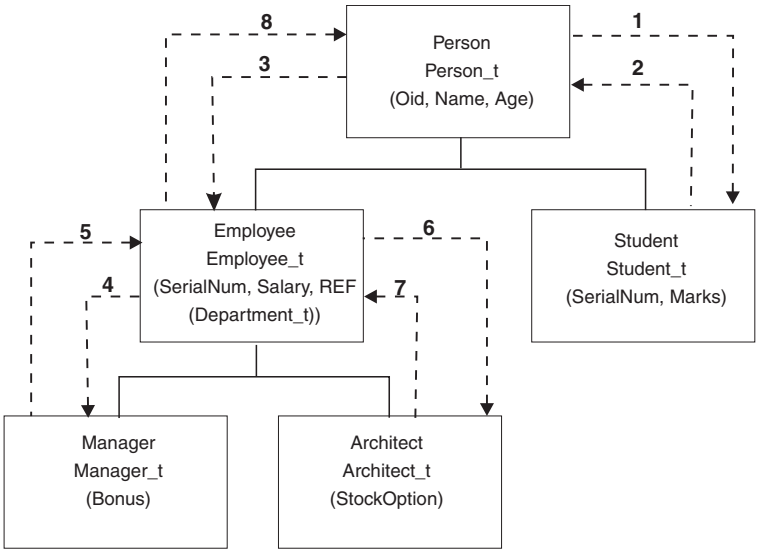
- An identical definition of subtables in both the source and the target databases
- An identical hierarchical relationship among the subtables in both the source and target databases
- An identical traverse order

Although you determine the starting point and the path down the hierarchy when defining the traverse order, each branch must be traversed to the end before the next branch in the hierarchy can be started. The import utility looks for violations of this condition within the specified traverse order.

Examples

Examples in this section are based on the following hierarchical structure with four valid traverse orders:

- Person, Employee, Manager, Architect, Student
- Person, Student, Employee, Manager, Architect
- Person, Employee, Architect, Manager, Student
- Person, Student, Employee, Architect, Manager



(artname: 00002440.gif)
Figure 2. An example of a hierarchy

Example 1
To re-create an entire hierarchy (contained in the data file `entire_hierarchy.ixf` created by a prior export operation) using import, you would enter the following commands:

```
DB2 CONNECT TO Target_db
DB2 IMPORT FROM entire_hierarchy.ixf OF IXF CREATE INTO
   HIERARCHY STARTING Person AS ROOT TABLE
```

Each type in the hierarchy is created if it does not exist. If these types already exist, they must have the same definition in the target database as in the source database. An SQL error (SQL20013N) is returned if they are not the same. Since a new hierarchy is being created, none of the subtables defined in the data file being moved to the target database (Target_db) can exist. Each of the tables in the source database hierarchy is created. Data from the source database is imported into the correct subtables of the target database.

Example 2
To re-create the entire hierarchy of the source database and import it to the target database, while only keeping selected data, you would enter the following commands:

```
DB2 CONNECT TO Target_db
DB2 IMPORT FROM entire_hierarchy.del OF DEL INSERT INTO (Person,
   Employee(Salary), Architect) IN HIERARCHY (Person, Employee,
   Manager, Architect, Student)
```

The target tables PERSON, EMPLOYEE, and ARCHITECT must all exist. Data is imported into the PERSON, EMPLOYEE, and ARCHITECT subtables. That is, the following will be imported:

- All columns in PERSON into PERSON
- All columns in PERSON plus SALARY in EMPLOYEE into EMPLOYEE
- All columns in PERSON plus SALARY in EMPLOYEE, plus all columns in ARCHITECT into ARCHITECT

Columns SerialNum and REF(Employee_t) are not imported into EMPLOYEE or its subtables (that is, ARCHITECT, which is the only subtable having data imported into it).

Note: Because ARCHITECT is a subtable of EMPLOYEE, and the only import column specified for EMPLOYEE is SALARY, SALARY is also the only Employee-specific column imported into ARCHITECT. That is, neither SerialNum nor REF(Employee_t) columns are imported into either EMPLOYEE or ARCHITECT rows.
Data for the MANAGER and the STUDENT tables is not imported.

Example 3
This example shows how to export from a regular table, and import as a single subtable in a hierarchy. The **EXPORT** command operates on regular (non-typed) tables, so there is no Type_id column in the data file. The file type modifier `no_type_id` is used to indicate this, so that the import utility does not expect the first column to be the Type_id column.

```
DB2 CONNECT TO Source_db
DB2 EXPORT TO Student_sub_table.del OF DEL SELECT * FROM
   Regular_Student
DB2 CONNECT TO Target_db
DB2 IMPORT FROM Student_sub_table.del OF DEL METHOD P(1,2,3,5,4)
   MODIFIED BY NO_TYPE_ID INSERT INTO HIERARCHY (Student)
```

In this example, the target table STUDENT must exist. Since STUDENT is a subtable, the modifier `no_type_id` is used to indicate that there is no Type_id in the first column. However, you must ensure that there is an existing Object_id column, in addition to all of the other attributes that exist in the STUDENT table. Object-id is expected to be the first column in each row imported into the STUDENT table. The **METHOD** clause reverses the order of the last two attributes.

LBAC-protected data import considerations

For a successful import operation into a table with protected rows, you must have LBAC (label-based access control) credentials. You must also provide a valid security label, or a security label that can be converted to a valid label, for the security policy currently associated with the target table.

If you do not have valid LBAC credentials, the import fails and an error (SQLSTATE 42512) is returned. In cases where the input data does not contain a security label or that security label is not in its internal binary format, you can use several file type modifiers to allow your import to proceed.

When you import data into a table with protected rows, the target table has one column with a data type of DB2SECURITYLABEL. If the input row of data does not contain a value for that column, that row is rejected unless the `usedefaults` file type modifier is specified in the import command, in which case the security label you hold for write access from the security policy protecting the table is used. If you do not hold a security label for write access, the row is rejected and processing continues on to the next row.

When you import data into a table that has protected rows and the input data does include a value for the column with a data type of DB2SECURITYLABEL, the same rules are followed as when you insert data into that table. If the security label protecting the row being imported (the one in that row of the data file) is one that you are able to write to, then that security label is used to protect the row. (In other words, it is written to the column that has a data type of

DB2SECURITYLABEL.) If you are not able to write to a row protected by that security label, what happens depends on how the security policy protecting the source table was created:

- If the CREATE SECURITY POLICY statement that created the policy included the option `RESTRICT NOT AUTHORIZED WRITE SECURITY LABEL`, the insert fails and an error is returned.
- If the CREATE SECURITY POLICY statement did not include the option or if it instead included the `OVERRIDE NOT AUTHORIZED WRITE SECURITY LABEL` option, the security label in the data file for that row is ignored and the security label you hold for write access is used to protect that row. No error or warning is issued in this case. If you do not hold a security label for write access, the row is rejected and processing continues on to the next row.

Delimiter considerations

When importing data into a column with a data type of DB2SECURITYLABEL, the value in the data file is assumed by default to be the actual bytes that make up the internal representation of that security label. However, some raw data might contain newline characters which could be misinterpreted by the IMPORT command as delimiting the row. If you have this problem, use the `delprioritychar` file type modifier to ensure that the character delimiter takes precedence over the row delimiter. When you use `delprioritychar`, any record or column delimiters that are contained within character delimiters are not recognized as being delimiters. Using the `delprioritychar` file type modifier is safe to do even if none of the values contain a newline character, but it does slow the import down slightly.

If the data being imported is in ASC format, you might want to take an extra step in order to prevent any trailing white space from being included in the imported security labels and security label names. ASCII format uses column positions as delimiters, so this might occur when importing into variable-length fields. Use the `striptblanks` file type modifier to truncate any trailing blank spaces.

Nonstandard security label values

You can also import data files in which the values for the security labels are strings containing the values of the components in the security label, for example, *S:(ALPHA,BETA)*. To do so you must use the file type modifier `seclabelchar`. When you use `seclabelchar`, a value for a column with a data type of DB2SECURITYLABEL is assumed to be a string constant containing the security label in the string format for security labels. If a string is not in the proper format, the row is not inserted and a warning (SQLSTATE 01H53) is returned. If the string does not represent a valid security label that is part of the security policy protecting the table, the row is not inserted and a warning (SQLSTATE 01H53) is returned.

You can also import a data file in which the values of the security label column are security label names. To import this sort of file you must use the file type modifier `seclabelname`. When you use `seclabelname`, all values for columns with a data type of DB2SECURITYLABEL are assumed to be string constants containing the names of existing security labels. If no security label exists with the indicated name for the security policy protecting the table, the row is not inserted and a warning (SQLSTATE 01H53) is returned.

Examples

For all examples, the input data file `myfile.del` is in DEL format. All are importing data into a table named REPS, which was created with this statement:

```
create table reps (row_label db2securitylabel,
id integer,
name char(30))
security policy data_access_policy
```

For this example, the input file is assumed to contain security labels in the default format:

```
db2 import from myfile.del of del modified by delprioritychar insert into reps
```

For this example, the input file is assumed to contain security labels in the security label string format:

```
db2 import from myfile.del of del modified by seclabelchar insert into reps
```

For this example, the input file is assumed to contain security labels names for the security label column:

```
db2 import from myfile.del of del modified by seclabelname insert into reps
```

Buffered-insert imports

In a partitioned database environment, the import utility can be enabled to use buffered inserts. This reduces the messaging that occurs when data is imported, resulting in better performance.

The buffered inserts option should only be enabled if you are not concerned about error reporting, since details about a failed buffered insert are not returned.

When buffered inserts are used, import sets a default **WARNINGCOUNT** value to 1. As a result, the operation will fail if any rows are rejected. If a record is rejected, the utility will roll back the current transaction. The number of committed records can be used to determine which records were successfully inserted into the database. The number of committed records can be non zero only if the `COMMITCOUNT` option was specified.

If a different **WARNINGCOUNT** value is explicitly specified on the import command, and some rows are rejected, the row summary output by the utility can be incorrect. This is due to a combination of the asynchronous error reporting used with buffered inserts and the fact that an error detected during the insertion of a group of rows causes all the rows of that group to be backed out. Since the utility would not reliably report which input records were rejected, it would be difficult to determine which records were committed and which records need to be re-inserted into the database.

Use the Db2 bind utility to request buffered inserts. The import package, `db2uimpm.bnd`, must be rebound against the database using the `INSERT BUF` option. For example:

```
db2 connect to your_database
db2 bind db2uimpm.bnd insert buf
```

Buffered inserts feature cannot be used in conjunction with import operations in the `INSERT_UPDATE` mode. The bind file `db2uImpInsUpdate.bnd` enforces this restriction. This file should never be bound with the `INSERT BUF` option. This

causes the import operations in the INSERT_UPDATE mode to fail. Import operations in the INSERT, REPLACE, or REPLACE_CREATE modes are not affected by the binding of the new file.

Identity column import considerations

The import utility can be used to import data into a table containing an identity column whether or not the input data has identity column values.

If no identity-related file type modifiers are used, the utility works according to the following rules:

- If the identity column is GENERATED ALWAYS, an identity value is generated for a table row whenever the corresponding row in the input file is missing a value for the identity column, or a NULL value is explicitly given. If a non-NULL value is specified for the identity column, the row is rejected (SQL3550W).
- If the identity column is GENERATED BY DEFAULT, the import utility makes use of user-supplied values, if they are provided; if the data is missing or explicitly NULL, a value is generated.

The import utility does not perform any extra validation of user-supplied identity values beyond what is normally done for values of the identity column's data type (that is, SMALLINT, INT, BIGINT, or DECIMAL). Duplicate values will not be reported. In addition, the compound=*x* modifier cannot be used when importing data into a table with an identity column.

There are two ways you can simplify the import of data into tables that contain an identity column: the identitymissing and the identityignore file type modifiers.

Importing data without an identity column

The identitymissing modifier makes importing a table with an identity column more convenient if the input data file does not contain any values (not even NULLS) for the identity column. For example, consider a table defined with the following SQL statement:

```
create table table1 (c1 char(30),
                     c2 int generated by default as identity,
                     c3 real,
                     c4 char(1))
```

A user might want to import data from a file (import.del) into TABLE1, and this data might have been exported from a table that does not have an identity column. The following is an example of such a file:

```
Robert, 45.2, J
Mike, 76.9, K
Leo, 23.4, I
```

One way to import this file would be to explicitly list the columns to be imported through the **IMPORT** command as follows:

```
db2 import from import.del of del replace into table1 (c1, c3, c4)
```

For a table with many columns, however, this syntax might be cumbersome and prone to error. An alternate method of importing the file is to use the identitymissing file type modifier as follows:

```
db2 import from import.del of del modified by identitymissing
   replace into table1
```

Importing data with an identity column
The identityignore modifier is in some ways the opposite of the identitymissing modifier: it indicates to the import utility that even though the input data file contains data for the identity column, the data should be ignored, and an identity value should be generated for each row. For example, a user might want to import the following data from a file (import.del) into TABLE1, as defined previously:

```
Robert, 1, 45.2, J
Mike, 2, 76.9, K
Leo, 3, 23.4, I
```

If the user-supplied values of 1, 2, and 3 are not to be used for the identity column, the user could issue the following **IMPORT** command:

```
db2 import from import.del of del method P(1, 3, 4)
   replace into table1 (c1, c3, c4)
```

Again, this approach might be cumbersome and prone to error if the table has many columns. The identityignore modifier simplifies the syntax as follows:

```
db2 import from import.del of del modified by identityignore
   replace into table1
```

When a table with an identity column is exported to an IXF file, the REPLACE_CREATE and the CREATE options of the **IMPORT** command can be used to re-create the table, including its identity column properties. If such an IXF file is created from a table containing an identity column of type GENERATED ALWAYS, the only way that the data file can be successfully imported is to specify the identityignore modifier. Otherwise, all rows will be rejected (SQL3550W).

Note: The CREATE and REPLACE_CREATE options of the IMPORT command are deprecated and might be removed in a future release.

Generated column import considerations

The import utility can be used to import data into a table containing (nonidentity) generated columns whether or not the input data has generated column values.

If no generated column-related file type modifiers are used, the import utility works according to the following rules:

- A value is generated for a generated column whenever the corresponding row in the input file is missing a value for the column, or a NULL value is explicitly given. If a non-NULL value is supplied for a generated column, the row is rejected (SQL3550W).
- If the server generates a NULL value for a generated column that is not nullable, the row of data to which this field belongs is rejected (SQL0407N). This could happen, for example, if a non-nullable generated column were defined as the sum of two table columns that have NULL values supplied to them in the input file.

There are two ways you can simplify the import of data into tables that contain a generated column: the generatedmissing and the generatedignore file type modifiers.

Importing data without generated columns
The generatedmissing modifier makes importing data into a table with generated columns more convenient if the input data file does not contain any values (not even NULLS) for all generated columns present in the table. For example, consider a table defined with the following SQL statement:

```
create table table1 (c1 int,
                     c2 int,
                     g1 int generated always as (c1 + c2),
                     g2 int generated always as (2 * c1),
                     c3 char(1))
```

A user might want to import data from a file (`load.del`) into TABLE1, and this data might have been exported from a table that does not have any generated columns. The following is an example of such a file:

```
1, 5, J
2, 6, K
3, 7, I
```

One way to import this file would be to explicitly list the columns to be imported through the **IMPORT** command as follows:

```
db2 import from import.del of del replace into table1 (c1, c2, c3)
```

For a table with many columns, however, this syntax might be cumbersome and prone to error. An alternate method of importing the file is to use the `generatedmissing` file type modifier as follows:

```
db2 import from import.del of del modified by generatedmissing
   replace into table1
```

Importing data with generated columns

The `generatedignore` modifier is in some ways the opposite of the `generatedmissing` modifier: it indicates to the import utility that even though the input data file contains data for all generated columns, the data should be ignored, and values should be generated for each row. For example, a user might want to import the following data from a file (`import.del`) into TABLE1, as defined previously:

```
1, 5, 10, 15, J
2, 6, 11, 16, K
3, 7, 12, 17, I
```

The user-supplied, non-NULL values of `10`, `11`, and `12` (for g1), and `15`, `16`, and `17` (for g2) result in the row being rejected (SQL3550W). To avoid this, the user could issue the following **IMPORT** command:

```
db2 import from import.del of del method P(1, 2, 5)
   replace into table1 (c1, c2, c3)
```

Again, this approach might be cumbersome and prone to error if the table has many columns. The `generatedignore` modifier simplifies the syntax as follows:

```
db2 import from import.del of del modified by generatedignore
   replace into table1
```

For an `INSERT_UPDATE`, if the generated column is also a primary key and the `generatedignore` modifier is specified, the **IMPORT** command honors the `generatedignore` modifier. The **IMPORT** command does not substitute the user-supplied value for this column in the WHERE clause of the UPDATE statement.

LOB import considerations

Since the import utility restricts the size of a single column value to 32 KB, extra considerations need to be taken when importing LOBs.

The import utility, by default, treats data in the input file as data to load into the column. However, when large object (LOB) data is stored in the main input data

file, the size of the data is limited to 32 KB. Therefore, to prevent loss of data, LOB data should be stored separate from the main datafile and the `lobsinfile` file type modifier should be specified when importing LOBs.

The LOBS FROM clause implicitly activates `lobsinfile`. The LOBS FROM clause conveys to the import utility the list of paths to search for the LOB files while importing the data. If LOBS FROM option is not specified, the LOB files to import are assumed to reside in the same path as the input relational data file.

Indicating where LOB data is stored

The LOB Location Specifier (LLS) can be used to store multiple LOBs in a single file when importing the LOB information. The export utility generates and stores it in the export output file when `lobsinfile` is specified, and it indicates where LOB data can be found. When data with the modified by `lobsinfile` option specified is being imported, the database will expect an LLS for each of the corresponding LOB columns. If something other than an LLS is encountered for a LOB column, the database will treat it as a LOB file and will load the entire file as the LOB.

For an import in `CREATE` mode, you can specify that the LOB data be created and stored in a separate table space by using the LONG IN clause.

The following example shows how you would import an DEL file which has its LOBs stored in separate files:

```
IMPORT FROM inputfile.del OF DEL
 LOBS FROM /tmp/data
 MODIFIED BY lobsinfile
 INSERT INTO newtable
```

User-defined distinct types import considerations

The import utility casts user-defined distinct types (UDTs) to similar base data types automatically. This saves you from having to explicitly cast UDTs to the base data types. Casting allows for comparisons between UDTs and the base data types in SQL.

Additional considerations for import

Client/server environments and import

When you import a file to a remote database, a stored procedure can be called to perform the import on the server.

A stored procedure cannot be called when:

- The application and database code pages are different.
- The file being imported is a multiple-part PC/IXF file.
- The method used for importing the data is either column name or relative column position.
- The target column list provided is longer than 4 KB.
- The LOBS FROM clause or the `lobsinfile` modifier is specified.
- The NULL INDICATORS clause is specified for ASC files.

When import uses a stored procedure, messages are created in the message file using the default language installed on the server. The messages are in the language of the application if the language at the client and the server are the same.

The import utility creates two temporary files in the tmp subdirectory of the sqllib directory (or the directory indicated by the **DB2INSTPROF** registry variable, if specified). One file is for data, and the other file is for messages generated by the import utility.

If you receive an error about writing or opening data on the server, ensure that:

- The directory exists.
- There is sufficient disk space for the files.
- The instance owner has write permission in the directory.

Table locking modes supported by the import utility

The import utility supports two table locking modes: offline, or ALLOW NO ACCESS, mode; and online, or ALLOW WRITE ACCESS mode.

ALLOW NO ACCESS mode prevents concurrent applications from accessing table data. ALLOW WRITE ACCESS mode allows concurrent applications both read and write access to the import target table. If no mode is explicitly specified, import runs in the default mode, ALLOW NO ACCESS. As well, the import utility is, by default, bound to the database with isolation level RS (read stability).

Offline import (ALLOW NO ACCESS)

In ALLOW NO ACCESS mode, import acquires an exclusive (X) lock on the target table is before inserting any rows. Holding a lock on a table has two implications:

- First, if there are other applications holding a table lock or row locks on the import target table, the import utility waits for those applications to commit or roll back their changes.
- Second, while import is running, any other application requesting locks waits for the import operation to complete.

Note: You can specify a locktimeout value, which prevents applications (including the import utility) from waiting indefinitely for a lock.
By requesting an exclusive lock at the beginning of the operation, import prevents deadlocks from occurring as a result of other applications working and holding row locks on the same target table.

Online import (ALLOW WRITE ACCESS)

In ALLOW WRITE ACCESS mode, the import utility acquires a nonexclusive (IX) lock on the target table. Holding this lock on the table has the following implications:

- If there are other applications holding an incompatible table lock, the import utility does not start inserting data until all of these applications commit or roll back their changes.
- While import is running, any other application requesting an incompatible table lock waits until the import commits or rolls back the current transaction. Note that import's table lock does not persist across a transaction boundary. As a result, online import has to request and potentially wait for a table lock after every commit.
- If there are other applications holding an incompatible row lock, the import utility stops inserting data until all of these applications commit or roll back their changes.
- While import is running, any other application requesting an incompatible row lock waits until the import operation commits or rolls back the current transaction.

To preserve the online properties, and to reduce the chance of a deadlock, an ALLOW WRITE ACCESS import periodically commits the current transaction and releases all row locks before escalating to an exclusive table lock. If you have not explicitly set a commit frequency, import performs commits as if COMMITCOUNT AUTOMATIC has been specified. No commits are performed if COMMITCOUNT is set to 0.

ALLOW WRITE ACCESS mode is not compatible with the following:

- Imports in REPLACE, CREATE, or REPLACE_CREATE mode
- Imports with buffered inserts
- Imports into a target view
- Imports into a hierarchy table
- Imports into a table with its lock granularity is set at the table level (set by using the LOCKSIZE parameter of the ALTER TABLE statement)

Load utility

Load overview

The load utility is capable of efficiently moving large quantities of data into newly created tables, or into tables that already contain data.

The utility can handle most data types, including XML, large objects (LOBs), and user-defined types (UDTs).

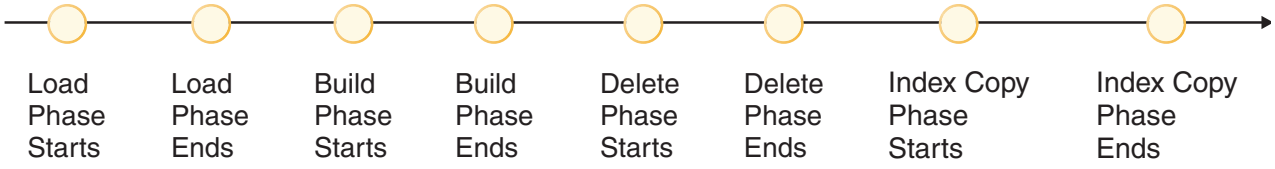
The load utility is faster than the import utility, because it writes formatted pages directly into the database, while the import utility performs SQL INSERTs.

The load utility does not fire triggers, and does not perform referential or table constraints checking (other than validating the uniqueness of the indexes).

The load process has several distinct phases (see Figure 3 on page 93):

1. **Analyze**
   When data is being loaded into a column-organized table, the first phase is the analyze phase, which is unique to column-organized tables. The analyze phase occurs only if a column compression dictionary needs to be built, which happens during a **LOAD REPLACE** operation, a **LOAD REPLACE RESETDICTIONARY** operation, a **LOAD REPLACE RESETDICTIONARYONLY** operation, or a **LOAD INSERT** operation (if the column-organized table is empty). For column-organized tables, this phase is followed by the load, build, and delete phases. The index copy phase applies to row-organized tables only.
2. **Load**
   During the load phase, data is loaded into the table, and index keys and table statistics are collected, if necessary. *Save points*, or points of consistency, are established at intervals specified through the **SAVECOUNT** parameter in the **LOAD** command. Messages are generated, indicating how many input rows were successfully loaded at the time of the save point.
3. **Build**
   During the build phase, indexes are produced based on the index keys collected during the load phase. The index keys are sorted during the load phase, and index statistics are collected (if the **STATISTICS USE PROFILE** option was specified, and profile indicates collecting index statistics). The statistics are similar to those collected through the **RUNSTATS** command.

4. **Delete**
   During the delete phase, the rows that caused a unique or primary key violation are removed from the table. These deleted rows are stored in the load exception table, if one was specified.
5. **Index copy**
   During the index copy phase, the index data is copied from a system temporary table space to the original table space. This will only occur if a system temporary table space was specified for index creation during a load operation with the **READ ACCESS** option specified.



(artname: 00002429.gif)
Figure 3. Phases of the Load Process for Row-organized Tables

Note: After you invoke the load utility, you can use the **LIST UTILITIES** command to monitor the progress of the load operation.

The following information is required when loading data:

- The path and the name of the input file, named pipe, or device.
- The name or alias of the target table.
- The format of the input source. This format can be DEL, ASC, PC/IXF, or CURSOR.
- Whether the input data is to be appended to the table, or is to replace the existing data in the table.
- A message file name, if the utility is invoked through the application programming interface (API), db2Load.

Load modes

- **INSERT**
  In this mode, load appends input data to the table without making any changes to the existing data.
- **REPLACE**
  In this mode, load deletes existing data from the table and populates it with the input data.
- **RESTART**
  In this mode, an interrupted load is resumed. In most cases, the load is resumed from the phase it failed in. If that phase was the load phase, the load is resumed from the last successful consistency point.
- **TERMINATE**
  In this mode, a failed load operation is rolled back.

The options you can specify include:

- That the data to be loaded resides on the client, if the load utility is invoked from a remotely connected client. Note that XML and LOB data are always read from the server, even you specify the **CLIENT** option.
- The method to use for loading the data: column location, column name, or relative column position.

- How often the utility is to establish consistency points.
- The names of the table columns into which the data is to be inserted.
- Whether or not preexisting data in the table can be queried while the load operation is in progress.
- Whether the load operation should wait for other utilities or applications to finish using the table or force the other applications off before proceeding.
- An alternate system temporary table space in which to build the index.
- The paths and the names of the input files in which LOBs are stored.

  Note: The load utility does not honor the **COMPACT** lob option.
- A message file name. During load operations, you can specify that message files be created to contain the error, warning, and informational messages associated with those operations. Specify the name of these files with the MESSAGES parameter.

  Note:

  1. You can only view the contents of a message file after the operation is finished. If you want to view load messages while a load operation is running, you can use the **LOAD QUERY** command.
  2. Each message in a message file begins on a new line and contains information provided by the Db2 message retrieval facility.
- Whether column values being loaded have implied decimal points.
- Whether the utility should modify the amount of free space available after a table is loaded.
- Whether statistics are to be gathered during the load process. This option is only supported if the load operation is running in REPLACE mode. Statistics are collected according to the profile defined for the table. The profile must be created by the **RUNSTATS** command before the **LOAD** command is executed. If the profile does not exist and the load operation is instructed to collect statistics according to the profile, the load will fail, and an error message will be returned.

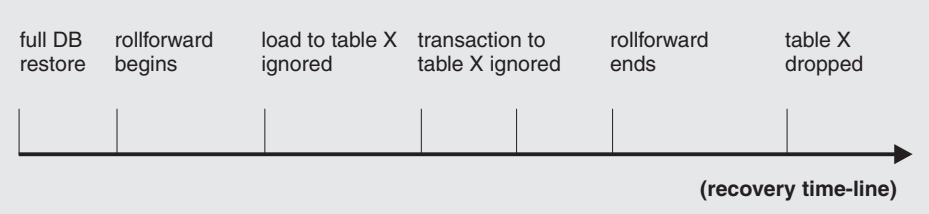
  If data is appended to a table, statistics are not collected. To collect current statistics on an appended table, invoke the **RUNSTATS** utility following completion of the load process. If gathering statistics on a table with a unique index, and duplicate keys are deleted during the delete phase, statistics are not updated to account for the deleted records. If you expect to have a significant number of duplicate records, do not collect statistics during the load operation. Instead, invoke the **RUNSTATS** utility following completion of the load process.
- Whether to keep a copy of the changes made. This is done to enable rollforward recovery of the database. This option is not supported if rollforward recovery is disabled for the database; that is, if the database configuration parameters **logarchmeth1** and **logarchmeth2** are set to OFF. If no copy is made, and rollforward recovery is enabled, the table space is left in Backup Pending state at the completion of the load operation.

  Logging is required for fully recoverable databases. The load utility almost completely eliminates the logging associated with the loading of data. In place of logging, you have the option of making a copy of the loaded portion of the table. If you have a database environment that allows for database recovery following a failure, you can do one of the following:
  - Explicitly request that a copy of the loaded portion of the table be made.
  - Take a backup of the table spaces in which the table resides immediately after the completion of the load operation.

If the database configuration parameter **logindexbuild** is set, and if the load operation is invoked with the **COPY YES** recoverability option and the INCREMENTAL indexing option, the load logs all index modifications. The benefit of using these options is that when you roll forward through the log records for this load, you also recover the indexes (whereas normally the indexes are not recovered unless the load uses the REBUILD indexing mode).

If you are loading a table that already contains data, and the database is non-recoverable, ensure that you have a backed-up copy of the database, or the table spaces for the table being loaded, before invoking the load utility, so that you can recover from errors.

If you want to perform a sequence of multiple load operations on a recoverable database, the sequence of operations will be faster if you specify that each load operation is non-recoverable, and take a backup at the end of the load sequence, than if you invoke each of the load operations with the **COPY YES** option. You can use the NONRECOVERABLE option to specify that a load transaction is to be marked as non-recoverable, and that it will not be possible to recover it by a subsequent rollforward operation. The rollforward utility will skip the transaction, and will mark the table into which data was being loaded as "invalid". The utility will also ignore any subsequent transactions against that table. After the rollforward operation is completed, such a table can only be dropped (see Figure 4). With this option, table spaces are not put in backup pending state following the load operation, and a copy of the loaded data does not have to be made during the load operation.



(artname: 00002436.gif)
Figure 4. Non-recoverable Processing During a Roll Forward Operation

- The fully qualified path to be used when creating temporary files during a load operation. The name is specified by the **TEMPFILES PATH** parameter of the **LOAD** command. The default value is the database path. The path resides on the server machine, and is accessed by the Db2 instance exclusively. Therefore, any path name qualification given to this parameter must reflect the directory structure of the server, not the client, and the Db2 instance owner must have read and write permission on the path.

Privileges and authorities required to use load

Privileges enable users to create or access database resources. Authority levels provide a method of grouping privileges and higher-level database manager maintenance and utility operations. Together, these act to control access to the database manager and its database objects. Users can access only those objects for which they have the appropriate authorization; that is, the required privilege or authority.

To load data into a table, you must have one of the following:

- DATAACCESS authority
- LOAD or DBADM authority on the database and

- INSERT privilege on the table when the load utility is invoked in INSERT mode, TERMINATE mode (to terminate a previous load insert operation), or RESTART mode (to restart a previous load insert operation)
- INSERT and DELETE privilege on the table when the load utility is invoked in REPLACE mode, TERMINATE mode (to terminate a previous load replace operation), or RESTART mode (to restart a previous load replace operation)
- INSERT privilege on the exception table, if such a table is used as part of the load operation.
- SELECT privilege on SYSCAT.TABLES is required in some cases where LOAD queries the catalog tables.

Since all load processes (and all Db2 server processes, in general), are owned by the instance owner, and all of these processes use the identification of the instance owner to access needed files, the instance owner must have read access to input data files. These input data files must be readable by the instance owner, regardless of who invokes the command.

If the REPLACE option is specified, the session authorization ID must have the authority to drop the table.

On Windows, and Windows.NET operating systems where Db2 is running as a Windows service, if you are loading data from files that reside on a network drive, you must configure the Db2 service to run under a user account that has read access to these files.

Note:

- To load data into a table that has protected columns, the session authorization ID must have LBAC credentials that allow write access to all protected columns in the table.
- To load data into a table that has protected rows, the session authorization ID must have been granted a security label for write access that is part of the security policy protecting the table.

LOAD authority

Users having LOAD authority at the database level, as well as INSERT privilege on a table, can use the **LOAD** command to load data into a table.

Note: Having DATAACCESS authority gives a user full access to the LOAD command.

Users having LOAD authority at the database level, as well as INSERT privilege on a table, can **LOAD RESTART** or **LOAD TERMINATE** if the previous load operation is a load to insert data.

Users having LOAD authority at the database level, as well as the INSERT and DELETE privileges on a table, can use the **LOAD REPLACE** command.

If the previous load operation was a load replace, the DELETE privilege must also have been granted to that user before the user can **LOAD RESTART** or **LOAD TERMINATE**.

If the exception tables are used as part of a load operation, the user must have INSERT privilege on the exception tables.

The user with this authority can perform **QUIESCE TABLESPACES FOR TABLE**, **RUNSTATS**, and **LIST TABLESPACES** commands.

Loading data

The load utility efficiently moves large quantities of data into newly created tables, or into tables that already contain data.

Before you begin

Before invoking the load utility, you must be connected to (or be able to implicitly connect to) the database into which you want to load the data.

Since the utility issues a COMMIT statement, complete all transactions and release all locks by issuing either a COMMIT or a ROLLBACK statement before invoking the load utility.

Data is loaded in the sequence that appears in the input file, except when using multidimensional clustering (MDC) tables, partitioned tables, or the **anyorder** file type modifier. If you want a particular sequence, sort the data before attempting a load operation. If clustering is required, the data should be sorted on the clustering index before loading. When loading data into multidimensional clustered tables (MDC), sorting is not required before the load operation, and data is clustered according to the MDC table definition. When loading data into partitioned tables, sorting is not required before the load operation, and data is partitioned according to the table definition.

Restrictions

These are some of the restrictions that apply to the load utility:

- Loading data into nicknames is not supported.
- Loading data into typed tables, or tables with structured type columns, is not supported.
- Loading data into declared temporary tables and created temporary tables is not supported.
- XML data can only be read from the server side; if you want to have the XML files read from the client, use the import utility.
- You cannot create or drop tables in a table space that is in Backup Pending state.
- You cannot load data into a database accessed through Db2 Connect or a server level before Db2 Version 2. Options that are only available with the current cannot be used with a server from the previous release.
- If an error occurs during a **LOAD REPLACE** operation, the original data in the table is lost. Retain a copy of the input data to allow the load operation to be restarted.
- Triggers are not activated on newly loaded rows. Business rules associated with triggers are not enforced by the load utility.
- Loading encrypted data is not supported.

These are some of the restrictions that apply to the load utility when loading into a partitioned table:

- Consistency points are not supported when the number of partitioning agents is greater than one.
- Loading data into a subset of data partitions while keeping the remaining data partitions fully online is not supported.

- The exception table used by a load operation or a set integrity pending operation cannot be partitioned.
- A unique index cannot be rebuilt when the load utility is running in insert mode or restart mode, and the load target table has any detached dependents.

Procedure

To invoke the load utility:

- Issue a **LOAD** command in the command line processor (CLP).
- Call the db2Load application programming interface (API) from a client application.
- Open the task assistant in IBM Data Studio for the **LOAD** command.

Example

The following is an example of a **LOAD** command issued through the CLP:

```
db2 load from stafftab.ixf of ixf messages staff.msgs
   insert into userid.staff copy yes use tsm data buffer 4000
```

In this example:

- Any warning or error messages are placed in the `staff.msgs` file.
- A copy of the changes made is stored in Tivoli® Storage Manager (TSM).
- 4000 pages of buffer space are to be used during the load operation.

The following is another example of a **LOAD** command issued through the CLP:

```
db2 load from stafftab.ixf of ixf messages staff.msgs
   tempfiles path /u/myuser replace into staff
```

In this example:

- The table data is being replaced.
- The **TEMPFILES PATH** parameter is used to specify `/u/myuser` as the server path into which temporary files are written.

Note: These examples use relative path names for the load input file. Relative path names are only allowed on calls from a client on the same database partition as the database. The use of fully qualified path names is recommended.

What to do next

After you invoke the load utility, you can use the **LIST UTILITIES** command to monitor the progress of the load operation. If a load operation is performed in either **INSERT** mode, **REPLACE** mode, or **RESTART** mode, detailed progress monitoring support is available. Issue the **LIST UTILITIES** command with the **SHOW DETAILS** parameter to view detailed information about the current load phase. Details are not available for a load operation performed in **TERMINATE** mode. The **LIST UTILITIES** command simply shows that a load terminate utility is currently running.

A load operation maintains unique constraints, range constraints for partitioned tables, generated columns, and LBAC security rules. For all other constraints, the table is placed in the Set Integrity Pending state at the beginning of a load operation. After the load operation is complete, the SET INTEGRITY statement must be used to take the table out of Set Integrity Pending state.

Loading XML data

The load utility can be used for the efficient movement of large volumes of XML data into tables.

When loading data into an XML table column, you can use the XML FROM option to specify the paths of the input XML data file or files. For example, to load data from an XML file `/home/user/xmlpath/xmlfile1.xml` you could use the following command:

```
LOAD FROM data1.del OF DEL XML FROM /home/user/xmlpath INSERT INTO USER.T1
```

The delimited ASCII input file `data1.del` contains an XML data specifier (XDS) that describes the location of the XML data to load. For example, the following XDS describes an XML document at offset 123 bytes in file `xmldata.ext` that is 456 bytes in length:

```
<XDS FIL='xmldata.ext' OFF='123' LEN='456' />
```

Loading XML data using a declared cursor is supported. The following example declares a cursor and uses the cursor and the **LOAD** command to add data from the table CUSTOMERS into the table LEVEL1_CUSTOMERS:

```
DECLARE cursor_income_level1 CURSOR FOR
   SELECT * FROM customers
   WHERE XMLEXISTS('$DOC/customer[income_level=1]');

LOAD FROM cursor_income_level1 OF CURSOR INSERT INTO level1_customers;
```

The `ANYORDER` file type modifier of the **LOAD** command is supported for loading XML data into an XML column.

During load, distribution statistics are not collected for columns of type XML.

Loading XML data in a partitioned database environment

For tables that are distributed among database partitions, you can load XML data from XML data files into the tables in parallel. When loading XML data from files into tables, the XML data files must be read-accessible to all the database partitions where loading is taking place

Validating inserted documents against schemas

The XMLVALIDATE option allows XML documents to be validated against XML schemas as they are loaded. In the following example, incoming XML documents are validated against the schema identified by the XDS in the delimited ASCII input file `data2.del`:

```
LOAD FROM data2.del OF DEL XML FROM /home/user/xmlpath XMLVALIDATE
     USING XDS INSERT INTO USER.T2
```

In this case, the XDS contains an SCH attribute with the fully qualified SQL identifier of the XML schema to use for validation, "S1.SCHEMA_A":

```
<XDS FIL='xmldata.ext' OFF='123' LEN='456' SCH='S1.SCHEMA_A' />
```

Specifying parse options

You can use the XMLPARSE option to specify whether whitespace in the loaded XML documents is preserved or stripped. In the following example, all loaded XML documents are validated against the schema with SQL identifier "S2.SCHEMA_A" and these documents are parsed with whitespace preserved:

```
LOAD FROM data2.del OF DEL XML FROM /home/user/xmlpath XMLPARSE PRESERVE
WHITESPACE XMLVALIDATE USING SCHEMA S2.SCHEMA_A INSERT INTO USER.T1
```

Load sessions - CLP examples

Example 1

TABLE1 has 5 columns:

- COL1 VARCHAR 20 NOT NULL WITH DEFAULT
- COL2 SMALLINT
- COL3 CHAR 4
- COL4 CHAR 2 NOT NULL WITH DEFAULT
- COL5 CHAR 2 NOT NULL

ASCFILE1 has 6 elements:

- ELE1 positions 01 to 20
- ELE2 positions 21 to 22
- ELE3 positions 23 to 23
- ELE4 positions 24 to 27
- ELE5 positions 28 to 31
- ELE6 positions 32 to 32
- ELE7 positions 33 to 40

Data Records:

```
1...5...10...15...20...25...30...35...40
Test data 1         XXN 123abcdN
Test data 2 and 3   QQY    XXN
Test data 4,5 and 6 WWN6789    Y
```

The following command loads the table from the file:

```
db2 load from ascfile1 of asc modified by striptblanks reclen=40
   method L (1 20, 21 22, 24 27, 28 31)
   null indicators (0,0,23,32)
   insert into table1 (col1, col5, col2, col3)
```

Note:

1. The specification of `striptblanks` in the MODIFIED BY parameter forces the truncation of blanks in VARCHAR columns (COL1, for example, which is 11, 17 and 19 bytes long, in rows 1, 2 and 3, respectively).
2. The specification of `reclen=40` in the MODIFIED BY parameter indicates that there is no newline character at the end of each input record, and that each record is 40 bytes long. The last 8 bytes are not use to load the table.
3. Since COL4 is not provided in the input file, it will be inserted into TABLE1 with its default value (it is defined NOT NULL WITH DEFAULT).
4. Positions 23 and 32 are used to indicate whether COL2 and COL3 of TABLE1 will be loaded NULL for a given row. If there is a `Y` in the column's null indicator position for a given record, the column will be NULL. If there is an `N`, the data values in the column's data positions of the input record (as defined in L(.....)) are used as the source of column data for the row. In this example, neither column in row 1 is NULL; COL2 in row 2 is NULL; and COL3 in row 3 is NULL.
5. In this example, the NULL INDICATORS for COL1 and COL5 are specified as 0 (zero), indicating that the data is not nullable.

6. The NULL INDICATOR for a given column can be anywhere in the input record, but the position must be specified, and the `Y` or `N` values must be supplied.

Example 2 (using dump files)

Table FRIENDS is defined as:

```
table friends "( c1 INT NOT NULL, c2 INT, c3 CHAR(8) )"
```

If an attempt is made to load the following data records into this table,

```
23, 24, bobby
, 45, john
4,, mary
```

the second row is rejected because the first INT is NULL, and the column definition specifies NOT NULL. Columns which contain initial characters that are not consistent with the DEL format will generate an error, and the record will be rejected. Such records can be written to a dump file.

DEL data appearing in a column outside of character delimiters is ignored, but does generate a warning. For example:

```
22,34,"bob"
24,55,"sam" sdf
```

The utility will load "sam" in the third column of the table, and the characters "sdf" will be flagged in a warning. The record is not rejected. Another example:

```
22 3, 34,"bob"
```

The utility will load `22,34,"bob"`, and generate a warning that some data in column one following the `22` was ignored. The record is not rejected.

Example 3 (Loading a table with an identity column)

TABLE1 has 4 columns:

- C1 VARCHAR(30)
- C2 INT GENERATED BY DEFAULT AS IDENTITY
- C3 DECIMAL(7,2)
- C4 CHAR(1)

TABLE2 is the same as TABLE1, except that C2 is a GENERATED ALWAYS identity column.

Data records in DATAFILE1 (DEL format):

```
"Liszt"
"Hummel",,187.43, H
"Grieg",100, 66.34, G
"Satie",101, 818.23, I
```

Data records in DATAFILE2 (DEL format):

```
"Liszt", 74.49, A
"Hummel", 0.01, H
"Grieg", 66.34, G
"Satie", 818.23, I
```

Note:

1. The following command generates identity values for rows 1 and 2, since no identity values are supplied in DATAFILE1 for those rows. Rows 3 and 4, however, are assigned the user-supplied identity values of 100 and 101, respectively.

   ```
   db2 load from datafile1.del of del replace into table1
   ```

2. To load DATAFILE1 into TABLE1 so that identity values are generated for all rows, issue one of the following commands:

   ```
   db2 load from datafile1.del of del method P(1, 3, 4)
      replace into table1 (c1, c3, c4)
   db2load from datafile1.del of del modified by identityignore
      replace into table1
   ```

3. To load DATAFILE2 into TABLE1 so that identity values are generated for each row, issue one of the following commands:

   ```
   db2 load from datafile2.del of del replace into table1 (c1, c3, c4)
   db2 load from datafile2.del of del modified by identitymissing
      replace into table1
   ```

4. To load DATAFILE1 into TABLE2 so that the identity values of 100 and 101 are assigned to rows 3 and 4, issue the following command:

   ```
   db2 load from datafile1.del of del modified by identityoverride
      replace into table2
   ```

   In this case, rows 1 and 2 will be rejected, because the utility has been instructed to override system-generated identity values in favor of user-supplied values. If user-supplied values are not present, however, the row must be rejected, because identity columns are implicitly not NULL.

5. If DATAFILE1 is loaded into TABLE2 without using any of the identity-related file type modifiers, rows 1 and 2 will be loaded, but rows 3 and 4 will be rejected, because they supply their own non-NULL values, and the identity column is GENERATED ALWAYS.

Example 3 (loading from CURSOR)

MY.TABLE1 has 3 columns:

- ONE INT
- TWO CHAR(10)
- THREE DATE

MY.TABLE2 has 3 columns:

- ONE INT
- TWO CHAR(10)
- THREE DATE

Cursor MYCURSOR is defined as follows:

```
declare mycursor cursor for select * from my.table1
```

The following command loads all the data from MY.TABLE1 into MY.TABLE2:

```
load from mycursor of cursor method P(1,2,3) insert into
  my.table2(one,two,three)
```

Note:

1. Only one cursor name can be specified in a single LOAD command. That is, `load from mycurs1, mycurs2 of cursor...` is not allowed.
2. `P` and `N` are the only valid METHOD values for loading from a cursor.

3. In this example, METHOD P and the insert column list (`one,two,three`) could have been omitted since they represent default values.
4. MY.TABLE1 can be a table, view, alias, or nickname.

Load considerations for partitioned tables

All of the existing load features are supported when the target table is partitioned with the exception of the following general restrictions:

- Consistency points are not supported when the number of partitioning agents is greater than one.
- Loading data into a subset of data partitions while the remaining data partitions remain fully online is not supported.
- The exception table used by a load operation cannot be partitioned.
- An exception table cannot be specified if the target table contains an XML column.
- A unique index cannot be rebuilt when the load utility is running in insert mode or restart mode, and the load target table has any detached dependents.
- Similar to loading MDC tables, exact ordering of input data records is not preserved when loading partitioned tables. Ordering is only maintained within the cell or data partition.
- Load operations utilizing multiple formatters on each database partition only preserve approximate ordering of input records. Running a single formatter on each database partition, groups the input records by cell or table partitioning key. To run a single formatter on each database partition, explicitly request `CPU_PARALLELISM` of 1.

General load behavior

The load utility inserts data records into the correct data partition. There is no requirement to use an external utility, such as a splitter, to partition the input data before loading.

The load utility does not access any detached or attached data partitions. Data is inserted into visible data partitions only. Visible data partitions are neither attached nor detached. In addition, a load replace operation does not truncate detached or attached data partitions. Since the load utility acquires locks on the catalog system tables, the load utility waits for any uncommitted ALTER TABLE transactions. Such transactions acquire an exclusive lock on the relevant rows in the catalog tables, and the exclusive lock must terminate before the load operation can proceed. This means that there can be no uncommitted ALTER TABLE ...ATTACH, DETACH, or ADD PARTITION transactions while load operation is running. Any input source records destined for an attached or detached data partition are rejected, and can be retrieved from the exception table if one is specified. An informational message is written to the message file to indicate some of the target table data partitions were in an attached or detached state. Locks on the relevant catalog table rows corresponding to the target table prevent users from changing the partitioning of the target table by issuing any ALTER TABLE ...ATTACH, DETACH, or ADD PARTITION operations while the load utility is running.

Handling of invalid rows

When the load utility encounters a record that does not belong to any of the visible data partitions the record is rejected and the load utility continues processing. The number of records rejected because of the range constraint violation is not explicitly displayed, but is included in the overall number of rejected records. Rejecting a record because of the range

violation does not increase the number of row warnings. A single message (SQL0327N) is written to the load utility message file indicating that range violations are found, but no per-record messages are logged. In addition to all columns of the target table, the exception table includes columns describing the type of violation that had occurred for a particular row. Rows containing invalid data, including data that cannot be partitioned, are written to the dump file.

Because exception table inserts are expensive, you can control which constraint violations are inserted into the exception table. For instance, the default behavior of the load utility is to insert rows that were rejected because of a range constraint or unique constraint violation, but were otherwise valid, into the exception table. You can turn off this behavior by specifying, respectively, NORANGEEXC or NOUNIQUEEXC with the FOR EXCEPTION clause. If you specify that these constraint violations should not be inserted into the exception table, or you do not specify an exception table, information about rows violating the range constraint or unique constraint is lost.

History file

If the target table is partitioned, the corresponding history file entry does not include a list of the table spaces spanned by the target table. A different operation granularity identifier ('R' instead of 'T') indicates that a load operation ran against a partitioned table.

Terminating a load operation

Terminating a load replace completely truncates all visible data partitions, terminating a load insert truncates all visible data partitions to their lengths before the load. Indexes are invalidated during a termination of an ALLOW READ ACCESS load operation that failed in the load copy phase. Indexes are also invalidated when terminating an ALLOW NO ACCESS load operation that touched the index (It is invalidated because the indexing mode is rebuild, or a key was inserted during incremental maintenance leaving the index in an inconsistent state). Loading data into multiple targets does not have any effect on load recovery operations except for the inability to restart the load operation from a consistency point taken during the load phase In this case, the `SAVECOUNT` load option is ignored if the target table is partitioned. This behavior is consistent with loading data into a MDC target table.

Generated columns

If a generated column is in any of the partitioning, dimension, or distribution keys, the `generatedoverride` file type modifier is ignored and the load utility generates values as if the `generatedignore` file type modifier is specified. Loading an incorrect generated column value in this case can place the record in the wrong physical location, such as the wrong data partition, MDC block or database partition. For example, once a record is on a wrong data partition, set integrity has to move it to a different physical location, which cannot be accomplished during online set integrity operations.

Data availability

The current ALLOW READ ACCESS load algorithm extends to partitioned tables. An ALLOW READ ACCESS load operation allows concurrent readers to access the whole table, including both loading and non-loading data partitions.

Important: Starting with Version 10.1 Fix Pack 1, the ALLOW READ ACCESS parameter is deprecated and might be removed in a future release. For more details, see "ALLOW READ ACCESS parameter in the LOAD command is deprecated" at .

The ingest utility also supports partitioned tables and is better suited to allow data concurrency and availability than the LOAD command with the ALLOW READ ACCESS parameter. It can move large amounts of data from files and pipes without locking the target table. In addition, data becomes accessible as soon as it is committed based on elapsed time or number of rows.

Data partition states

After a successful load, visible data partitions might change to either or both Set Integrity Pending or Read Access Only table state, under certain conditions. Data partitions might be placed in these states if there are constraints on the table which the load operation cannot maintain. Such constraints might include check constraints and detached materialized query tables. A failed load operation leaves all visible data partitions in the Load Pending table state.

Error isolation

Error isolation at the data partition level is not supported. Isolating the errors means continuing a load on data partitions that did not run into an error and stopping on data partitions that did run into an error. Errors can be isolated between different database partitions, but the load utility cannot commit transactions on a subset of visible data partitions and roll back the remaining visible data partitions.

Other considerations

- Incremental indexing is not supported if any of the indexes are marked invalid. An index is considered invalid if it requires a rebuild or if detached dependents require validation with the SET INTEGRITY statement.
- Loading into tables partitioned using any combination of partitioned by range, distributed by hash, or organized by dimension algorithms is also supported.
- For log records which include the list of object and table space IDs affected by the load, the size of these log records (LOAD START and COMMIT (PENDING LIST)) could grow considerably and hence reduce the amount of active log space available to other applications.
- When a table is both partitioned and distributed, a partitioned database load might not affect all database partitions. Only the objects on the output database partitions are changed.
- During a load operation, memory consumption for partitioned tables increases with the number of tables. Note, that the total increase is not linear as only a small percentage of the overall memory requirement is proportional to the number of data partitions.

LBAC-protected data load considerations

For a successful load operation into a table with protected rows, you must have LBAC (label-based access control) credentials. You must also provide a valid security label, or a security label that can be converted to a valid label, for the security policy currently associated with the target table.

If you do not have valid LBAC credentials, the load fails and an error (SQLSTATE 42512) is returned. In cases where the input data does not contain a security label or that security label is not in its internal binary format, you can use several file type modifiers to allow your load to proceed.

When you load data into a table with protected rows, the target table has one column with a data type of DB2SECURITYLABEL. If the input row of data does not contain a value for that column, that row is rejected unless the `usedefaults` file type modifier is specified in the load command, in which case the security label you hold for write access from the security policy protecting the table is used. If you do not hold a security label for write access, the row is rejected and processing continues on to the next row.

When you load data into a table that has protected rows and the input data does include a value for the column with a data type of DB2SECURITYLABEL, the same rules are followed as when you insert data into that table. If the security label protecting the row being loaded (the one in that row of the data file) is one that you are able to write to, then that security label is used to protect the row. (In other words, it is written to the column that has a data type of DB2SECURITYLABEL.) If you are not able to write to a row protected by that security label, what happens depends on how the security policy protecting the source table was created:

- If the CREATE SECURITY POLICY statement that created the policy included the option `RESTRICT NOT AUTHORIZED WRITE SECURITY LABEL`, the row is rejected.
- If the CREATE SECURITY POLICY statement did not include the option or if it instead included the `OVERRIDE NOT AUTHORIZED WRITE SECURITY LABEL` option, the security label in the data file for that row is ignored and the security label you hold for write access is used to protect that row. No error or warning is issued in this case. If you do not hold a security label for write access, the row is rejected and processing continues on to the next row.

Delimiter considerations

When loading data into a column with a data type of DB2SECURITYLABEL, the value in the data file is assumed by default to be the actual bytes that make up the internal representation of that security label. However, some raw data might contain newline characters which could be misinterpreted by the **LOAD** command as delimiting the row. If you have this problem, use the `delprioritychar` file type modifier to ensure that the character delimiter takes precedence over the row delimiter. When you use `delprioritychar`, any record or column delimiters that are contained within character delimiters are not recognized as being delimiters. Using the `delprioritychar` file type modifier is safe to do even if none of the values contain a newline character, but it does slow the load down slightly.

If the data being loaded is in ASC format, you might have to take an extra step in order to prevent any trailing white space from being included in the loaded security labels and security label names. ASCII format uses column positions as delimiters, so this might occur when loading into variable-length fields. Use the `striptblanks` file type modifier to truncate any trailing blank spaces.

Nonstandard security label values

You can also load data files in which the values for the security labels are strings containing the values of the components in the security label, for example, *S:(ALPHA,BETA)*. To do so you must use the file type modifier

seclabelchar. When you use seclabelchar, a value for a column with a data type of DB2SECURITYLABEL is assumed to be a string constant containing the security label in the string format for security labels. If a string is not in the proper format, the row is not inserted and a warning (SQLSTATE 01H53) is returned. If the string does not represent a valid security label that is part of the security policy protecting the table, the row is not inserted and a warning (SQLSTATE 01H53) is returned.

You can also load a data file in which the values of the security label column are security label names. To load this sort of file you must use the file type modifier seclabelname. When you use seclabelname, all values for columns with a data type of DB2SECURITYLABEL are assumed to be string constants containing the names of existing security labels. If no security label exists with the indicated name for the security policy protecting the table, the row is not loaded and a warning (SQLSTATE 01H53) is returned.

Rejected rows

Rows that are rejected during the load are sent to either a dumpfile or an exception table (if they are specified in the **LOAD** command), depending on the reason why the rows were rejected. Rows that are rejected due to parsing errors are sent to the dumpfile. Rows that violate security policies are sent to the exception table.

Note: You cannot specify an exception table if the target table contains an XML column.

Examples

For all examples, the input data file myfile.del is in DEL format. All are loading data into a table named REPS, which was created with this statement:

```
create table reps (row_label db2securitylabel,
id integer,
name char(30))
security policy data_access_policy
```

For this example, the input file is assumed to contain security labels in the default format:

```
db2 load from myfile.del of del modified by delprioritychar insert into reps
```

For this example, the input file is assumed to contain security labels in the security label string format:

```
db2 load from myfile.del of del modified by seclabelchar insert into reps
```

For this example, the input file is assumed to contain security labels names for the security label column:

```
db2 load from myfile.del of del modified by seclabelname insert into reps
```

Identity column load considerations

The load utility can be used to load data into a table containing an identity column whether or not the input data has identity column values.

If no identity-related file type modifiers are used, the utility works according to the following rules:

- If the identity column is GENERATED ALWAYS, an identity value is generated for a table row whenever the corresponding row in the input file is missing a

value for the identity column, or a NULL value is explicitly given. If a non-NULL value is specified for the identity column, the row is rejected (SQL3550W).
- If the identity column is GENERATED BY DEFAULT, the load utility makes use of user-supplied values, if they are provided; if the data is missing or explicitly NULL, a value is generated.

The load utility does not perform any extra validation of user-supplied identity values beyond what is normally done for values of the identity column's data type (that is, SMALLINT, INT, BIGINT, or DECIMAL). Duplicate values are not reported.

In most cases the load utility cannot guarantee that identity column values are assigned to rows in the same order that these rows appear in the data file. Because the assignment of identity column values is managed in parallel by the load utility, those values are assigned in arbitrary order. The exceptions to this are as follows:
- In single-partition databases, rows are not processed in parallel when `CPU_PARALLELISM` is set to 1. In this case, identity column values are implicitly assigned in the same order that rows appear in the data file parameter.
- In multi-partition databases, identity column values are assigned in the same order that the rows appear in the data file if the identity column is in the distribution key and if there is a single partitioning agent (that is, if you do not specify multiple partitioning agents or the `anyorder` file type modifier).

When loading a table in a partitioned database where the table has an identity column in the partitioning key and the `identityoverride` modifier is not specified, the `SAVECOUNT` option cannot be specified. When there is an identity column in the partitioning key and identity values are being generated, restarting a load from the load phase on at least one database partition requires restarting the whole load from the beginning of the load phase, which means that there can't be any consistency points.

Note: A load `RESTART` operation is not permitted if all of the following criteria are met:
- The table being loaded is in a partitioned database environment, and it contains at least one identity column that is either in the distribution key or is referenced by a generated column that is part of the distribution key.
- The `identityoverride` modifier is not specified.
- The previous load operation that failed included loading database partitions that failed after the load phase.

A load `TERMINATE` or `REPLACE` operation should be issued instead.

There are three mutually exclusive ways you can simplify the loading of data into tables that contain an identity column: the `identitymissing`, the `identityignore`, and the `identityoverride` file type modifiers.

Loading data without identity columns

The `identitymissing` modifier makes loading a table with an identity column more convenient if the input data file does not contain any values (not even NULLS) for the identity column. For example, consider a table defined with the following SQL statement:

```
create table table1 (c1 varchar(30),
                     c2 int generated by default as identity,
                     c3 decimal(7,2),
                     c4 char(1))
```

If you want to load TABLE1 with data from a file (`load.del`) that has been exported from a table that does not have an identity column, see the following example:

```
Robert, 45.2, J
Mike, 76.9, K
Leo, 23.4, I
```

One way to load this file would be to explicitly list the columns to be loaded through the **LOAD** command as follows:

```
db2 load from load.del of del replace into table1 (c1, c3, c4)
```

For a table with many columns, however, this syntax might be cumbersome and prone to error. An alternate method of loading the file is to use the `identitymissing` file type modifier as follows:

```
db2 load from load.del of del modified by identitymissing
   replace into table1
```

This command would result in the three columns in the data file being loaded into c1, c3, and c4 of TABLE1. A value will be generated for each row in c2.

Loading data with identity columns

The `identityignore` modifier indicates to the load utility that even though the input data file contains data for the identity column, the data should be ignored, and an identity value should be generated for each row. For example, a user might want to load TABLE1, as defined previously, from a data file (`load.del`) containing the following data:

```
Robert, 1, 45.2, J
Mike, 2, 76.9, K
Leo, 3, 23.4, I
```

If the user-supplied values of 1, 2, and 3 are not used for the identity column, you can issue the following **LOAD** command:

```
db2 load from load.del of del method P(1, 3, 4)
   replace into table1 (c1, c3, c4)
```

Again, this approach might be cumbersome and prone to error if the table has many columns. The `identityignore` modifier simplifies the syntax as follows:

```
db2 load from load.del of del modified by identityignore
   replace into table1
```

Loading data with user-supplied values

The `identityoverride` modifier is used for loading user-supplied values into a table with a GENERATED ALWAYS identity column. This can be quite useful when migrating data from another database system, and the table must be defined as GENERATED ALWAYS, or when loading a table from data that was recovered using the `DROPPED TABLE RECOVERY` option on the **ROLLFORWARD DATABASE** command. When this modifier is used, any rows with no data (or NULL data) for the identity column are rejected (SQL3116W). You should also note that when using this modifier, it is possible to violate the uniqueness property of GENERATED ALWAYS columns.In this situation, perform a load `TERMINATE` operation, followed

by a subsequent load INSERT or REPLACE operation.

Generated column load considerations

You can load data into a table containing (nonidentity) generated columns whether or not the input data has generated column values. The load utility generates the column values.

If no generated column-related file type modifiers are used, the load utility works according to the following rules:

- Values are created for generated columns when the corresponding row of the data file is missing a value for the column or a NULL value is supplied. If a non-NULL value is supplied for a generated column, the row is rejected (SQL3550W).
- If a NULL value is created for a generated column that is not nullable, the entire row of data is rejected (SQL0407N). This could occur if, for example, a non-nullable generated column is defined as the sum of two table columns that include NULL values in the data file.

There are three mutually exclusive ways you can simplify the loading of data into tables that contain a generated column: the generatedmissing, the generatedignore, and the generatedoverride file type modifiers:

Loading data without generated columns

The generatedmissing modifier makes loading a table with generated columns more convenient if the input data file does not contain any values (not even NULLS) for all generated columns present in the table. For example, consider a table defined with the following SQL statement:

```
CREATE TABLE table1 (c1 INT,
                     c2 INT,
                     g1 INT GENERATED ALWAYS AS (c1 + c2),
                     g2 INT GENERATED ALWAYS AS (2 * c1),
                     c3 CHAR(1))
```

If you want to load TABLE1 with data from a file (load.del) that has been exported from a table that does not have any generated columns, see the following example:

```
1, 5, J
2, 6, K
3, 7, I
```

One way to load this file would be to explicitly list the columns to be loaded through the LOAD command as follows:

```
DB2 LOAD FROM load.del of del REPLACE INTO table1 (c1, c2, c3)
```

For a table with many columns, however, this syntax might be cumbersome and prone to error. An alternate method of loading the file is to use the generatedmissing file type modifier as follows:

```
DB2 LOAD FROM load.del of del MODIFIED BY generatedmissing
   REPLACE INTO table1
```

This command will result in the three columns of data file being loaded into c1, c2, and c3 of TABLE1. Due to the generatedmissing modifier, values for columns g1 and g2 of TABLE1 will be generated automatically and will not map to any of the data file columns.

Loading data with generated columns

The generatedignore modifier indicates to the load utility that even though the input data file contains data for all generated columns present in the target table, the data should be ignored, and the computed values should be loaded into each generated column. For example, if you want to load TABLE1, as defined previously, from a data file (load.del) containing the following data:

```
1, 5, 10, 15, J
2, 6, 11, 16, K
3, 7, 12, 17, I
```

The user-supplied, non-NULL values of 10, 11, and 12 (for g1), and 15, 16, and 17 (for g2) result in the row being rejected (SQL3550W) if no generated-column related file type modifiers are used. To avoid this, the user could issue the following LOAD command:

```
DB2 LOAD FROM load.del of del method P(1, 2, 5)
   REPLACE INTO table1 (c1, c2, c3)
```

Again, this approach might be cumbersome and prone to error if the table has many columns. The generatedignore modifier simplifies the syntax as follows:

```
DB2 LOAD FROM load.del of del MODIFIED BY generatedignore
   REPLACE INTO table1
```

This command will result in the columns of data file being loaded into c1 (with the data 1, 2, 3), c2 (with the data 5,6,7), and c3 (with the data J, K, I) of TABLE1. Due to the generatedignore modifier, values for columns g1 and g2 of TABLE1 will be generated automatically and the data file columns (10, 11, 12 and 15, 16, 17) will be ignored.

Loading data with user-supplied values

The generatedoverride modifier is used for loading user-supplied values into a table with generated columns. This can be useful when migrating data from another database system, or when loading a table from data that was recovered using the RECOVER DROPPED TABLE option of the **ROLLFORWARD DATABASE** command. When this modifier is used, any rows with no data (or NULL data) for non-nullable generated columns are rejected (SQL3116W).

When this modifier is used, the table is placed in the Set Integrity Pending state after the load operation. To take the table out of Set Integrity Pending state without verifying the user-supplied values, issue the following command:

```
SET INTEGRITY FOR table-name GENERATED COLUMN IMMEDIATE
  UNCHECKED
```

To take the table out of the Set Integrity Pending state and force verification of the user-supplied values, issue the following command:

```
SET INTEGRITY FOR table-name IMMEDIATE CHECKED
```

If a generated column is in any of the partitioning, dimension, or distribution keys, the generatedoverride modifier is ignored and the load utility generates values as if the generatedignore modifier is specified. This is done to avoid a scenario where a user-supplied generated column value conflicts with its generated column definition, which would place the resulting record in the wrong physical location, such as the wrong data partition, MDC block, or database partition.

Note: The **LOAD** utility does not support generating column values when one of the generated column expressions contains one of the following:

- a user-defined function that is a compiled compound SQL
- a user-defined function that is FENCED

If you attempt to load into such tables the load operation fails. However, you can provide your own values for these types of generated columns by using the `generatedoverride` file type modifier.

Moving data using the CURSOR file type

By specifying the CURSOR file type when using the **LOAD** command, you can load the results of an SQL query directly into a target table without creating an intermediate exported file.

Additionally, you can load data from another database by referencing a nickname within the SQL query, by using the `DATABASE` option within the DECLARE CURSOR statement, or by using the sqlu_remotefetch_entry media entry when using the API interface.

There are three approaches for moving data using the CURSOR file type. The first approach uses the Command Line Processor (CLP), the second the API, and the third uses the ADMIN_CMD procedure. The key differences between the CLP and the ADMIN_CMD procedure are outlined in the following table.

Table 21. Differences between the CLP and ADMIN_CMD procedure.

Differences	CLP	ADMIN_CMD_procedure
Syntax	The query statement as well as the source database used by the cursor are defined outside of the LOAD command using a DECLARE CURSOR statement.	The query statement as well as the source database used by the cursor is defined within the LOAD command using the LOAD from (DATABASE database-alias query-statement)
User authorization for accessing a different database	If the data is in a different database than the one you currently connect to, the DATABASE keyword must be used in the DECLARE CURSOR statement. You can specify the user id and password in the same statement as well. If the user id and password are not specified in the DECLARE CURSOR statement, the user id and password explicitly specified for the source database connection are used to access the target database.	If the data is in a different database than the one you are currently connected to, the DATABASE keyword must be used in the LOAD command before the query statement. The user id and password explicitly specified for the source database connection are required to access the target database. You cannot specify a userid or password for the source database. Therefore, if no userid and password were specified when the connection to the target database was made, or the userid and password specified cannot be used to authenticate against the source database, the ADMIN_CMD procedure cannot be used to perform the load.

To execute a LOAD FROM CURSOR operation from the CLP, a cursor must first be declared against an SQL query. Once this is declared, you can issue the **LOAD** command using the declared cursor's name as the *cursorname* and CURSOR as the file type.

For example:

1. Suppose a source and target table both reside in the same database with the following definitions:

   Table ABC.TABLE1 has 3 columns:

   - ONE INT
   - TWO CHAR(10)
   - THREE DATE

   Table ABC.TABLE2 has 3 columns:

   - ONE VARCHAR
   - TWO INT
   - THREE DATE

   Executing the following CLP commands will load all the data from ABC.TABLE1 into ABC.TABLE2:

   ```
   DECLARE mycurs CURSOR FOR SELECT TWO, ONE, THREE FROM abc.table1
      LOAD FROM mycurs OF cursor INSERT INTO abc.table2
   ```

   Note: The preceding example shows how to load from an SQL query through the CLP. However, loading from an SQL query can also be accomplished through the db2Load API. Define the *piSourceList* of the sqlu_media_list structure to use the *sqlu_statement_entry* structure and SQLU_SQL_STMT media type and define the *piFileType* value as SQL_CURSOR.

2. Suppose the source and target tables reside in different databases with the following definitions:

Table ABC.TABLE1 in database 'dbsource' has 3 columns:

- ONE INT
- TWO CHAR(10)
- THREE DATE

Table ABC.TABLE2 in database 'dbtarget' has 3 columns:

- ONE VARCHAR
- TWO INT
- THREE DATE

Provided that you have enabled federation and cataloged the data source ('dsdbsource'), you can declare a nickname against the source database, then declare a cursor against this nickname, and invoke the LOAD command with the FROM CURSOR option, as demonstrated in the following example:

```
CREATE NICKNAME myschema1.table1 FOR dsdbsource.abc.table1
DECLARE mycurs CURSOR FOR SELECT TWO,ONE,THREE FROM myschema1.table1
LOAD FROM mycurs OF cursor INSERT INTO abc.table2
```

Or, you can use the DATABASE option of the DECLARE CURSOR statement, as demonstrated in the following example:

```
DECLARE mycurs CURSOR DATABASE dbsource USER dsciaraf USING mypasswd
FOR SELECT TWO,ONE,THREE FROM abc.table1
LOAD FROM mycurs OF cursor INSERT INTO abc.table2
```

Using the DATABASE option of the DECLARE CURSOR statement (also known as the remotefetch media type when using the Load API) has some benefits over the nickname approach:

Performance

Fetching of data using the remotefetch media type is tightly integrated within a load operation. There are fewer layers of transition to fetch a record compared to the nickname approach. Additionally, when source and target tables are distributed identically in a multi-partition database, the load utility can parallelize the fetching of data, which can further improve performance.

Ease of use

There is no need to enable federation, define a remote datasource, or declare a nickname. Specifying the `DATABASE` option (and the `USER` and `USING` options if necessary) is all that is required.

While this method can be used with cataloged databases, the use of nicknames provides a robust facility for fetching from various data sources which cannot simply be cataloged.

To support this remotefetch functionality, the load utility makes use of infrastructure which supports the SOURCEUSEREXIT facility. The load utility spawns a process which executes as an application to manage the connection to the source database and perform the fetch. This application is associated with its own transaction and is not associated with the transaction under which the load utility is running.

Note:

1. The previous example shows how to load from an SQL query against a cataloged database through the CLP using the `DATABASE` option of the DECLARE CURSOR statement. However, loading from an SQL query against a cataloged database can also be done through the db2Load API, by defining the *piSourceList* and *piFileTypevalues* of the *db2LoadStruct* structure to use the sqlu_remotefetch_entry media entry and SQLU_REMOTEFETCH media type respectively.
2. As demonstrated in the previous example, the source column types of the SQL query do not need to be identical to their target column types, although they do have to be compatible.

Restrictions

When loading from a cursor defined using the `DATABASE` option (or equivalently when using the sqlu_remotefetch_entry media entry with the db2Load API), the following restrictions apply:

1. The `SOURCEUSEREXIT` option cannot be specified concurrently.
2. The `METHOD N` option is not supported.
3. The `usedefaults` file type modifier is not supported.

Propagating dependent immediate staging tables

If the table being loaded is an underlying table of a staging table with the immediate propagate attribute, and if the load operation is done in insert mode, the subsequent propagation into the dependent immediate staging tables is incremental.

During incremental propagation, the rows corresponding to the appended rows in the underlying tables are appended into the staging tables. Incremental propagation is faster in the case of large underlying tables with small amounts of appended data. Performance is also improved if the staging table is used to refresh its dependent deferred materialized query table. There are cases in which incremental propagation is not allowed, and the staging table is marked incomplete. That is, the staging byte of the CONST_CHECKED column has a value of F. In this state, the staging table can not be used to refresh its dependent deferred materialized query table, and a full refresh is required in the materialized query table maintenance process.

If a table is in incomplete state and the `INCREMENTAL` option has been specified, but incremental propagation of the table is not possible, an error is returned. If any of the following have taken place, the system turns off immediate data propagation and sets the table state to incomplete:

- A load replace operation has taken place on an underlying table of the staging table, or the NOT LOGGED INITIALLY WITH EMPTY TABLE option has been activated after the last integrity check on the underlying table.
- The dependent materialized query table of the staging table, or the staging table has been loaded in `REPLACE` or `INSERT` mode.
- An underlying table has been taken out of Set Integrity Pending state before the staging table has been propagated by using the `FULL ACCESS` option during integrity checking.
- An underlying table of the staging table has been checked for integrity non-incrementally.
- The table space containing the staging table or its underlying table has been rolled forward to a point in time, and the staging table and its underlying table reside in different table spaces.

If the staging table has a `W` value in the CONST_CHECKED column of the SYSCAT.TABLES catalog, and the `NOT INCREMENTAL` option is not specified, incremental propagation to the staging table takes place and the CONST_CHECKED column of SYSCAT.TABLES is marked as `U` to indicate that not all data has been verified by the system.

The following example illustrates a load insert operation into the underlying table UT1 of staging table G1 and its dependent deferred materialized query table AST1. In this scenario, both the integrity checking for UT1 and the refreshing of AST1 are processed incrementally:

```
LOAD FROM IMTFILE1.IXF of IXF INSERT INTO UT1;
LOAD FROM IMTFILE2.IXF of IXF INSERT INTO UT1;
SET INTEGRITY FOR UT1,G1 IMMEDIATE CHECKED;

REFRESH TABLE AST1 INCREMENTAL;
```

Refreshing dependent immediate materialized query tables

If the underlying table of an immediate refresh materialized query table is loaded using the `INSERT` option, executing the SET INTEGRITY statement on the

dependent materialized query tables defined with REFRESH IMMEDIATE results in an incremental refresh of the materialized query table.

During an incremental refresh, the rows corresponding to the appended rows in the underlying tables are updated and inserted into the materialized query tables. Incremental refresh is faster in the case of large underlying tables with small amounts of appended data. There are cases in which incremental refresh is not allowed, and full refresh (that is, recomputation of the materialized query table definition query) is used.

When the INCREMENTAL option is specified, but incremental processing of the materialized query table is not possible, an error is returned if:

- A load replace operation has taken place into an underlying table of the materialized query table or the NOT LOGGED INITIALLY WITH EMPTY TABLE option has been activated since the last integrity check on the underlying table.
- The materialized query table has been loaded (in either REPLACE or INSERT mode).
- An underlying table has been taken out of Set Integrity Pending state before the materialized query table is refreshed by using the FULL ACCESS option during integrity checking.
- An underlying table of the materialized query table has been checked for integrity non-incrementally.
- The materialized query table was in Set Integrity Pending state before an upgrade.
- The table space containing the materialized query table or its underlying table has been rolled forward to a point in time, and the materialized query table and its underlying table reside in different table spaces.

If the materialized query table has one or more W values in the CONST_CHECKED column of the SYSCAT.TABLES catalog, and if the NOT INCREMENTAL option is not specified in the SET INTEGRITY statement, the table is incrementally refreshed and the CONST_CHECKED column of SYSCAT.TABLES is marked U to indicate that not all data has been verified by the system.

The following example illustrates a load insert operation into the underlying table UTI of the materialized query table AST1. UT1 is checked for data integrity and is placed in the no data movement mode. UT1 is put back into full access state once the incremental refresh of AST1 is complete. In this scenario, both the integrity checking for UT1 and the refreshing of AST1 are processed incrementally.

```
LOAD FROM IMTFILE1.IXF of IXF INSERT INTO UT1;
LOAD FROM IMTFILE2.IXF of IXF INSERT INTO UT1;
SET INTEGRITY FOR UT1 IMMEDIATE CHECKED;
REFRESH TABLE AST1;
```

MDC and ITC considerations

The following restrictions apply when loading data into multidimensional clustering (MDC) and insert time clustering (ITC) tables:

- The SAVECOUNT option of the **LOAD** command is not supported.
- The totalfreespace file type modifier is not supported since these tables manage their own free space.
- The anyorder file type modifier is required for MDC or ITC tables. If a load is executed into an MDC or ITC table without the anyorder modifier, it will be explicitly enabled by the utility.

When using the **LOAD** command with an MDC or ITC table, violations of unique constraints are be handled as follows:

- If the table included a unique key before the load operation and duplicate records are loaded into the table, the original record remains and the new records are deleted during the delete phase.
- If the table did not include a unique key before the load operation and both a unique key and duplicate records are loaded into the table, only one of the records with the unique key is loaded and the others are deleted during the delete phase.

  Note: There is no explicit technique for determining which record is loaded and which is deleted.

Performance Considerations

To improve the performance of the load utility when loading MDC tables with more than one dimension, the *util_heap_sz* database configuration parameter value should be increased. The mdc-load algorithm performs significantly better when more memory is available to the utility. This reduces disk I/O during the clustering of data that is performed during the load phase. Beginning in version 9.5, the value of the DATA BUFFER option of the **LOAD** command can temporarily exceed *util_heap_sz* if more memory is available in the system. .

MDC or ITC load operations always have a build phase since all MDC and ITC tables have block indexes.

During the load phase, extra logging for the maintenance of the block map is performed. There are approximately two extra log records per extent allocated. To ensure good performance, the *logbufsz* database configuration parameter should be set to a value that takes this into account.

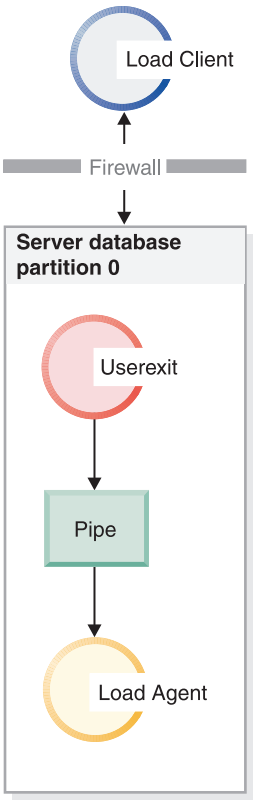
A system temporary table with an index is used to load data into MDC and ITC tables. The size of the table is proportional to the number of distinct cells loaded. The size of each row in the table is proportional to the size of the MDC dimension key. ITC tables only have one cell and use a 2-byte dimension key. To minimize disk I/O caused by the manipulation of this table during a load operation, ensure that the buffer pool for the temporary table space is large enough.

Moving data using a customized application (user exit)

The load `SOURCEUSEREXIT` option provides a facility through which the load utility can execute a customized script or executable, referred to herein as a *user exit*.

The purpose of the user exit is to populate one or more named pipes with data that is simultaneously read from by the load utility. In a multi-partition database, multiple instances of the user exit can be invoked concurrently to achieve parallelism of the input data.

As Figure 5 on page 118 shows, the load utility creates a one or more named pipes and spawns a process to execute your customized executable. Your user exit feeds data into the named pipe(s) while the load utility simultaneously reads.



(artname: 00023506.gif)
Figure 5. The load utility reads from the pipe and processes the incoming data.

The data fed into the pipe must reflect the load options specified, including the file type and any file type modifiers. The load utility does not directly read the data files specified. Instead, the data files specified are passed as arguments to your user exit when it is executed.

Invoking your user exit

The user exit must reside in the bin subdirectory of the Db2 installation directory (often known as sqllib). The load utility invokes the user exit executable with the following command line arguments:

```
<base pipename> <number of source media>
<source media 1> <source media 2> ... <user exit ID>
<number of user exits> <database partition number>
```

Where:

<base pipename >
Is the base name for named-pipes that the load utility creates and reads data from. The utility creates one pipe for every source file provided to the LOAD command, and each of these pipes is appended with `.xxx`, where `xxx` is the index of the source file provided. For example, if there are 2 source files provided to the LOAD command, and the <base pipename> argument passed to the user exit is `pipe123`, then the two named pipes that your user exit should feed with data are `pipe123.000` and `pipe123.001`. In a partitioned database environment, the load utility appends the database partition (DBPARTITION) number `.yyy` to the base pipe name, resulting in the pipe name `pipe123.yyy.xxx`..

<number of source media>
Is the number of media arguments which follow.

<source media 1> <source media 2> ...
Is the list of one or more source files specified in the LOAD command. Each source file is placed inside double quotation marks.

<user exit ID>
Is a special value useful when the PARALLELIZE option is enabled. This integer value (from 1 to N, where N is the total number of user exits being spawned) identifies a particular instance of a running user exit. When the PARALLELIZE option is not enabled, this value defaults to 1.

<number of user exits>
Is a special value useful when the PARALLELIZE option is enabled. This value represents the total number of concurrently running user exits. When the PARALLELIZE option is not enabled, this value defaults to 1.

<database partition number>
Is a special value useful when the PARALLELIZE option is enabled. This is the database partition (DBPARTITION) number on which the user exit is executing. When the PARALLELIZE option is not enabled, this value defaults to 0.

Additional options and features

The following section describes additional SOURCEUSEREXIT facility options:

REDIRECT
This option allows you to pass data into the STDIN handle or capture data from the STDOUT and STDERR handles of the user exit process.

INPUT FROM BUFFER <buffer>
Allows you to pass information directly into the STDIN input stream of your user exit. After spawning the process which executes the user exit, the load utility acquires the file-descriptor to the STDIN of this new process and passes in the buffer provided. The user exit reads from STDIN to acquire the information. The load utility simply sends the contents of <buffer> to the user exit using STDIN and does not interpret or modify its contents. For example, if your user exit is designed to read two values from STDIN, an eight-byte userid and an eight-byte password, your user exit executable written in C might contain the following lines:

```
rc = read (stdin, pUserID, 8);
rc = read (stdin, pPasswd, 8);
```

A user could pass this information using the INPUT FROM BUFFER option as shown in the following LOAD command:

```
LOAD FROM myfile1 OF DEL INSERT INTO table1
SOURCEUSEREXIT myuserexit1 REDIRECT INPUT FROM BUFFER myuseridmypasswd
```

Note: The load utility limits the size of <buffer> to the maximum size of a LOB value. However, from within the command line processor (CLP), the size of <buffer> is restricted to the maximum size of a CLP statement. From within CLP, it is also recommended that <buffer> contain only traditional ASCII characters. These issues can be avoided if the load utility is invoked using the db2Load API, or if the INPUT FROM FILE option is used instead.

INPUT FROM FILE <filename>

Allows you to pass the contents of a client side file directly into the STDIN input stream of your user exit. This option is almost identical to the `INPUT FROM BUFFER` option, however this option avoids the potential CLP limitation. The filename must be a fully qualified client side file and must not be larger than the maximum size of a LOB value.

OUTPUT TO FILE <filename>

Allows you to capture the STDOUT and STDERR streams from your user exit process into a server side file. After spawning the process which executes the user exit executable, the load utility redirects the STDOUT and STDERR handles from this new process into the filename specified. This option is useful for debugging and logging errors and activity within your user exit. The filename must be a fully qualified server side file. When the `PARALLELIZE` option is enabled, one file exists per user exit and each file appends a three-digit numeric identifier, such as *filename.000*.

PARALLELIZE

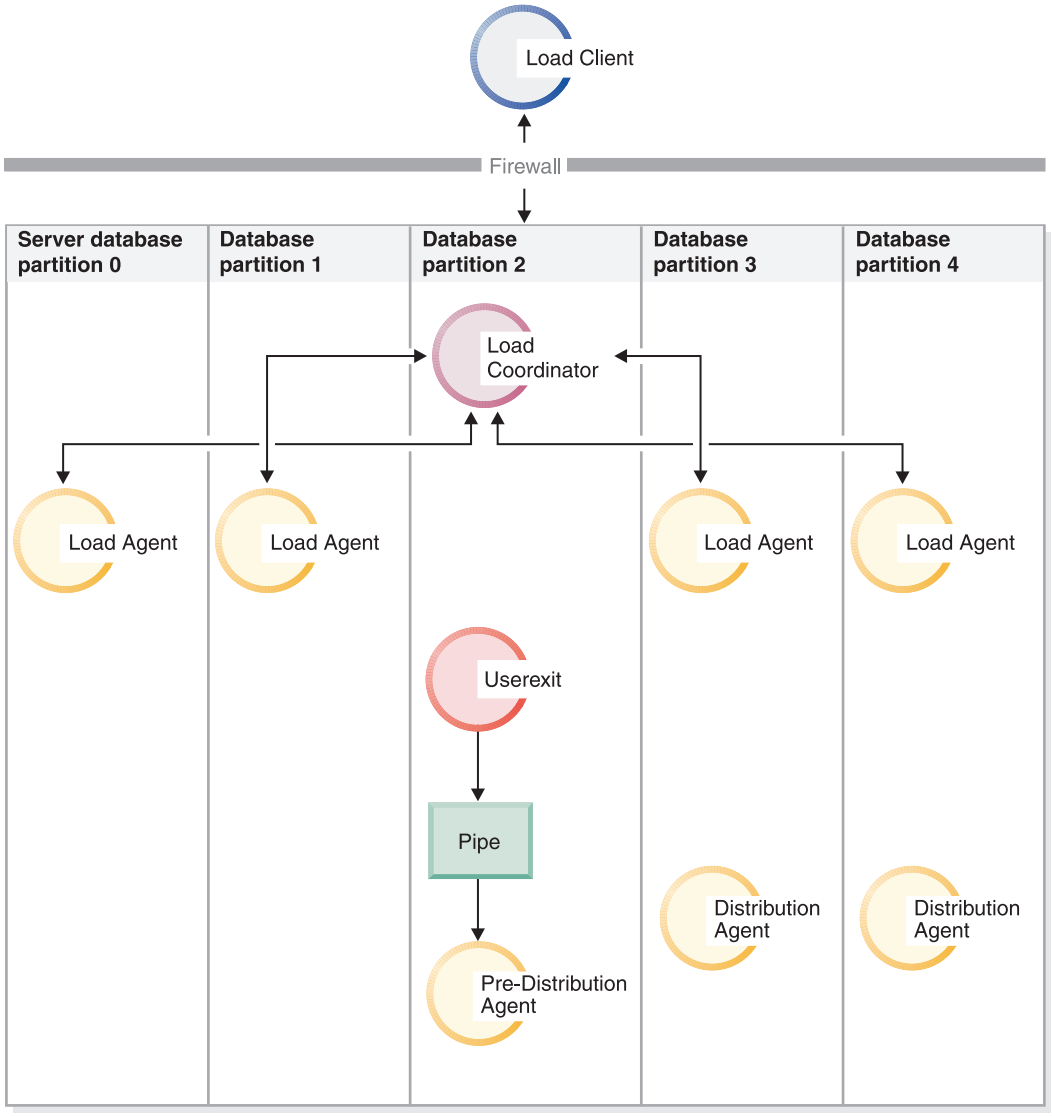
This option can increase the throughput of data coming into the load utility by invoking multiple user exit processes simultaneously. This option is only applicable to a multi-partition database. The number of user exit instances invoked is equal to the number of partitioning agents if data is to be distributed across multiple database partitions during the load operation, otherwise it is equal to the number of loading agents.

The <user exit ID>, <number of user exits>, and <database partition number> arguments passed into each user exit reflect the unique identifier (1 to N), the total number of user exits (N), and the database partition (DBPARTITION) number on which the user exit instance is running, respectively. You should ensure that any data written to the named pipe by each user exit process is not duplicated by the other concurrent processes. While there are many ways your user exit application might accomplish this, these values could be helpful to ensure data is not duplicated. For example, if each record of data contains a unique integer column value, your user exit application could use the <user exit ID> and <number of user exits> values to ensure that each user exit instance returns a unique result set into its named pipe. Your user exit application might use the **MODULUS** property in the following way:

```
i = <user exit ID>
N = <number of user exits>

foreach record
{
   if ((unique-integer MOD N) == i)
   {
     write this record to my named-pipe
   }
}
```

The number of user exit processes spawned depends on the distribution mode specified for database partitioning:

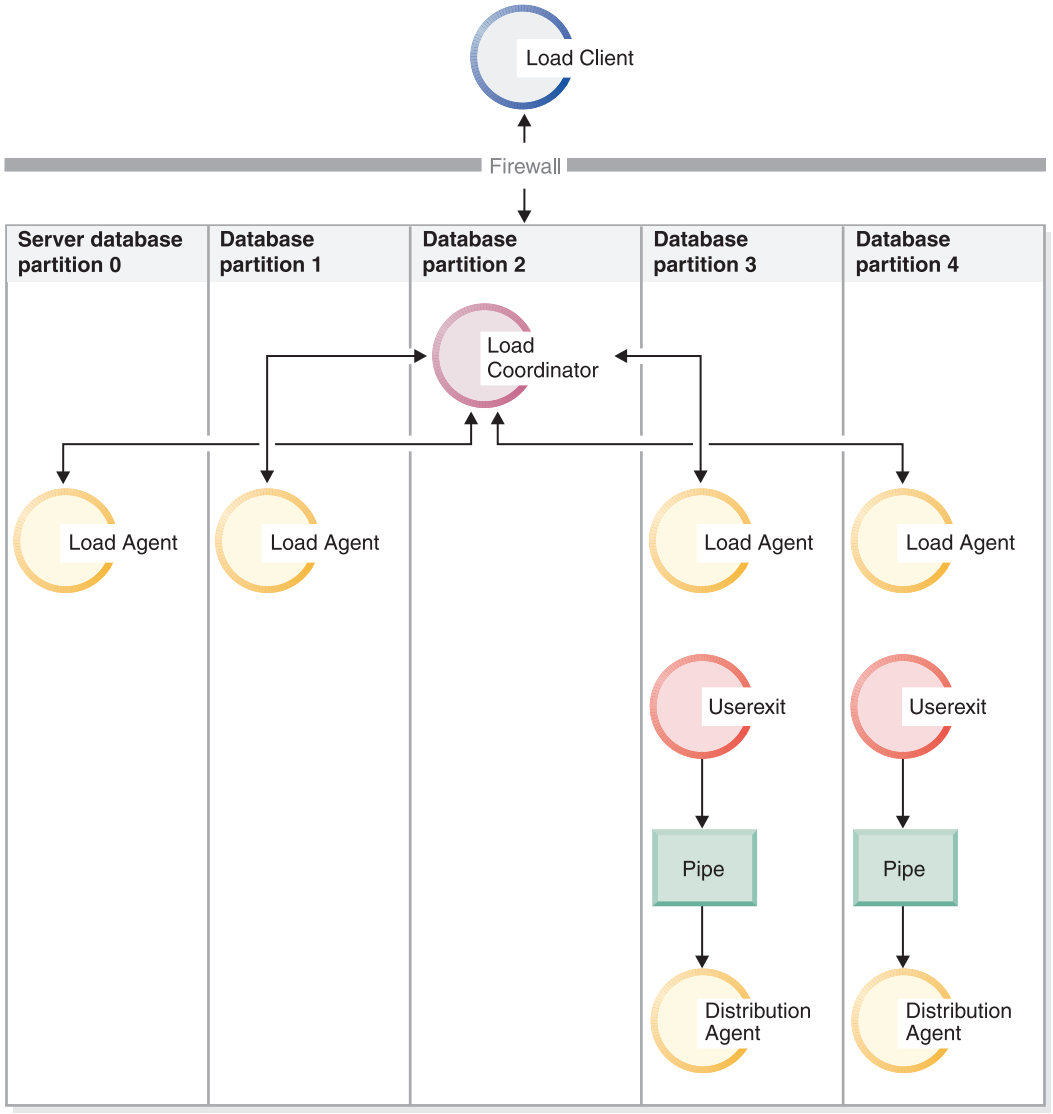
1. As Figure 6 on page 121 shows, one user exit process is spawned for every pre-partitioning agent when `PARTITION_AND_LOAD` (default) or `PARTITION_ONLY` without PARALLEL is specified. .



(artname: 00023507.gif)

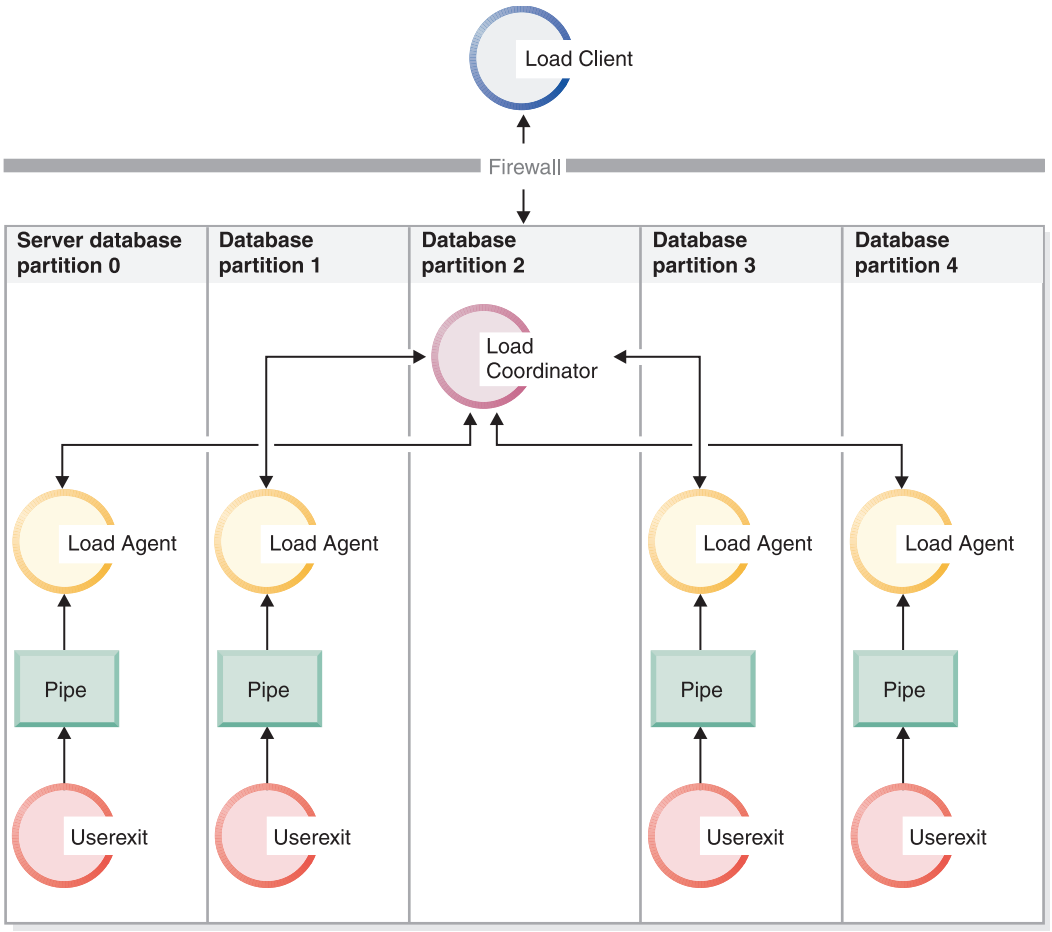
Figure 6. The various tasks performed when `PARTITION_AND_LOAD` (default) or `PARTITION_ONLY` without PARALLEL is specified.

2. As Figure 7 on page 122 shows, one user exit process is spawned for every partitioning agent when `PARTITION_AND_LOAD` (default) or `PARTITION_ONLY` with `PARALLEL` is specified.



(artname: 00024493.gif)
Figure 7. The various tasks performed when `PARTITION_AND_LOAD` (default) or `PARTITION_ONLY` with PARALLEL is specified.

3. As Figure 8 on page 123 shows, one user exit process is spawned for every load agent when `LOAD_ONLY` or `LOAD_ONLY_VERIFY_PART` is specified.



(artname: 00023508.gif)
Figure 8. The various tasks performed when LOAD_ONLY or LOAD_ONLY_VERIFY_PART is specified.

Restrictions

- The LOAD_ONLY and LOAD_ONLY_VERIFY_PART partitioned-db-cfg mode options are not supported when the SOURCEUSEREXIT PARALLELIZE option is not specified.

Examples

Example 1: A Load userexit script that replaces all tab characters '\t' with comma characters ',' from every record of the source media file. To invoke the Load utility using this userexit script, use a command similar to the following:

```
DB2 LOAD FROM /path/file1 OF DEL INSERT INTO schema1.table1
SOURCEUSEREXIT example1.pl REDIRECT OUTPUT TO FILE /path/ue_msgs.txt
```

Note that the userexit must be placed into the sqllib/bin/ folder, and requires execute permissions.

example1.pl:

```
#!/bin/perl

# Filename: example1.pl
#
# This script is a simple example of a userexit for the Load utility
# SOURCEUSEREXIT feature. This script will replace all tab characters '\t'
# with comma characters ',' from every record of the source media file.
```

```
#
# To invoke the Load utility using this userexit, use a command similar to:
#
#  db2 LOAD FROM /path/file1 OF DEL INSERT INTO schema1.table1
#   SOURCEUSEREXIT example1.pl REDIRECT OUTPUT TO FILE /path/ue_msgs.txt
#
# The userexit must be placed into the sqllib/bin/ folder, and requires
# execute permissions.
#------------------------------------------------------------------
if ($#ARGV < 5)
{
  print "Invalid number of arguments:\n@ARGV\n";
  print "Load utility should invoke userexit with 5 arguments (or more):\n";
  print "<base pipename> <number of source media> ";
  print "<source media 1> <source media 2> ... <user exit ID> ";
  print "<number of user exits> <database partition number> ";
  print "<optional: redirected input> \n";
  die;
}

# Open the output fifo file (the Load utility is reading from this pipe)
#------------------------------------------------------------------------
$basePipeName = $ARGV[0];
$outputPipeName = sprintf("%s.000", $basePipeName);
open(PIPETOLOAD, '>', $outputPipeName) || die "Could not open $outputPipeName";

# Get number of Media Files
#--------------------------
$NumMediaFiles = $ARGV[1];

# Open each media file, read the contents, replace '\t' with ',', send to Load
#------------------------------------------------------------------------------
for ($i=0; $i<$NumMediaFiles; $i++)
{
  # Open the media file
  #--------------------
  $mediaFileName = $ARGV[2+$i];
  open(MEDIAFILETOREAD, '<', $mediaFileName) || die "Could not open $mediaFileName";

  # Read each record of data
  #-------------------------
  while ( $line = <MEDIAFILETOREAD> )
  {
    # Replace '\t' characters with ','
    #---------------------------------
    $line =~ s/\t/,/g;

    # send this record to Load for processing
    #----------------------------------------
    print PIPETOLOAD $line;
  }
  # Close the media file
  #---------------------
  close MEDIAFILETOREAD;
}

# Close the fifo
#---------------
close PIPETOLOAD;

exit 0;
```

Monitoring a load operation using the LIST UTILITIES command

You can use the **LIST UTILITIES** command to monitor the progress of load operations on a database.

Procedure

To use the **LIST UTILITIES** command:
Issue the **LIST UTILITIES** command and specify the **SHOW DETAIL** parameter:

```
list utilities show detail
```

Example

The following is an example of the output for monitoring the performance of a load operation using the **LIST UTILITIES** command:

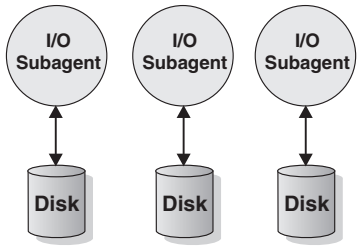
```
ID                                   = 10
Type                                 = LOAD
Database Name                        = TEST
Member Number                        = 1
Description                          = OFFLINE LOAD DEL AUTOMATIC INDEXING REPLACE
COPY NO BEER   .TABLE1
Start Time                           = 08/16/2011 08:52:53.861841
State                                = Executing
Invocation Type                      = User
Progress Monitoring:
   Phase Number                      = 1
      Description                    = SETUP
      Total Work                     = 0 bytes
      Completed Work                 = 0 bytes
      Start Time                     = 08/16/2011 08:52:53.861865

   Phase Number [Current]            = 2
      Description                    = LOAD
      Total Work                     = 49900 rows
      Completed Work                 = 25313 rows
      Start Time                     = 08/16/2011 08:52:54.277687

   Phase Number                      = 3
      Description                    = BUILD
      Total Work                     = 2 indexes
      Completed Work                 = 0 indexes
      Start Time                     = Not Started
```

Additional considerations for load

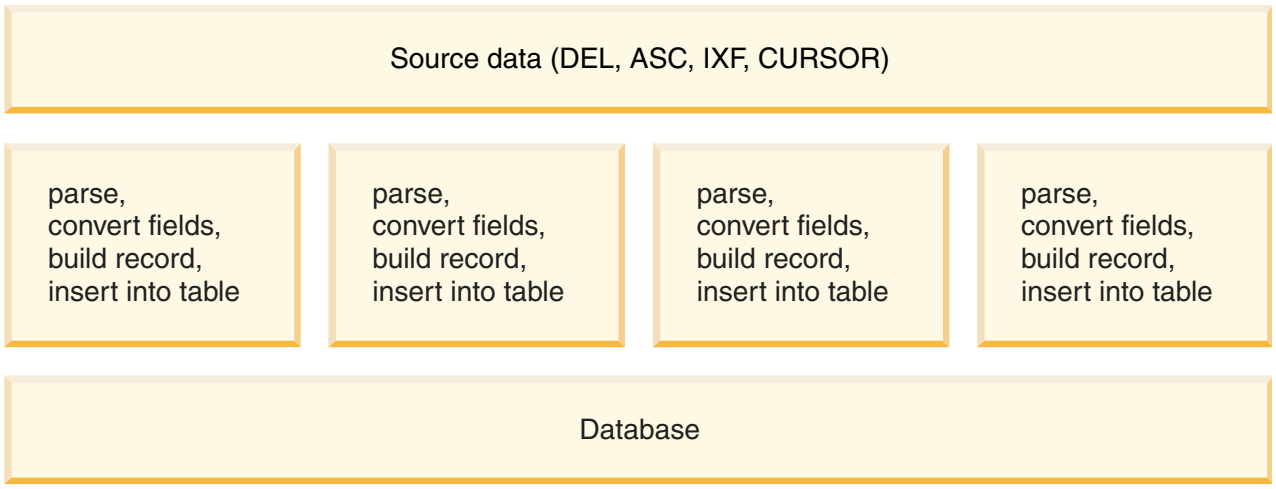
Parallelism and loading

The load utility takes advantage of a hardware configuration in which multiple processors or multiple storage devices are used, such as in a symmetric multiprocessor (SMP) environment.

There are several ways in which parallel processing of large amounts of data can take place using the load utility. One way is through the use of multiple storage devices, which allows for I/O parallelism during the load operation (see Figure 9 on page 126). Another way involves the use of multiple processors in an SMP environment, which allows for intra-partition parallelism (see Figure 10 on page 126). Both can be used together to provide even faster loading of data.



(artname: 00002434.gif)
Figure 9. Taking Advantage of I/O Parallelism When Loading Data



(artname: 00002432.gif)
Figure 10. Taking Advantage of Intra-partition Parallelism When Loading Data

Index creation during load operations

Indexes are built during the build phase of a load operation. There are four indexing modes that can be specified in the **LOAD** command.

1. `REBUILD`. All indexes are rebuilt.
2. `INCREMENTAL`. Indexes are extended with new data.
3. `AUTOSELECT`. The load utility automatically decides between `REBUILD` or `INCREMENTAL` mode. `AUTOSELECT` is the default. If a **LOAD REPLACE** operation is taking place, the `REBUILD` indexing mode is used. Otherwise, the indexing mode chosen is based on the ratio of the amount of existing data in the table to the amount of newly loaded data. If the ratio is sufficiently large, the `INCREMENTAL` indexing mode is chosen. Otherwise, the `REBUILD` indexing mode is chosen.
4. `DEFERRED`. The load utility does not attempt index creation if this mode is specified. Indexes are marked as needing a refresh, and a rebuild might be forced the first time they are accessed. The `DEFERRED` option is not allowed in any of the following situations:
   - If the **ALLOW READ ACCESS** option is specified (it does not maintain the indexes and index scanners require a valid index)
   - If any unique indexes are defined against the table
   - If any expression-based indexes are defined against the table
   - If XML data is being loaded (the XML Paths index is unique and is created by default whenever an XML column is added to a table)

Load operations that specify the **ALLOW READ ACCESS** option require special consideration in terms of space usage and logging depending on the type of indexing mode chosen. When the **ALLOW READ ACCESS** option is specified, the load utility keeps indexes available for queries even while they are being rebuilt.

When a load operation in **ALLOW READ ACCESS** mode specifies the **INDEXING MODE INCREMENTAL** option, the load utility writes some log records that protect the integrity of the index tree. The number of log records written is a fraction of the number of inserted keys and is a number considerably less than would be needed by a similar SQL insert operation. A load operation in **ALLOW NO ACCESS** mode with the **INDEXING MODE INCREMENTAL** option specified writes only a small log record beyond the normal space allocation logs.

Note: This is only true if you did not specify **COPY YES** and have the **logindexbuild** configuration parameter set to ON.

When a load operation in **ALLOW READ ACCESS** mode specifies the **INDEXING MODE REBUILD** option, new indexes are built as a *shadow* either in the same table space as the original index or in a system temporary table space. The original indexes remain intact and are available during the load operation and are only replaced by the new indexes at the end of the load operation while the table is exclusively locked. If the load operation fails and the transaction is rolled back, the original indexes remain intact.

By default, the shadow index is built in the same table space as the original index. Since both the original index and the new index are maintained simultaneously, there must be sufficient table space to hold both indexes at the same time. If the load operation is aborted, the extra space used to build the new index is released. If the load operation commits, the space used for the original index is released and the new index becomes the current index. When the new indexes are built in the same table space as the original indexes, replacing the original indexes takes place almost instantaneously.

If the indexes are built within an SMS table space, you can see index files in the table space directory with the .IN1 suffix and the .INX suffix. These suffixes do not indicate which is the original index and which is the shadow index. However, if the indexes are built in a DMS table space, you cannot see the new shadow index.

Improving index creation performance

Building new indexes in a system temporary table space

The new index can be built in a system temporary table space to avoid running out of space in the original table space. The **USE** *tablespace-name* option allows the indexes to be rebuilt in a system temporary table space when using **INDEXING MODE REBUILD** and **ALLOW READ ACCESS** options. The system temporary table can be an SMS or a DMS table space, but the page size of the system temporary table space must match the page size of the original index table space.

The **USE** *tablespace-name* option is ignored if the load operation is not in **ALLOW READ ACCESS** mode, or if the indexing mode is incompatible. The **USE** *tablespace-name* option is only supported for the **INDEXING MODE REBUILD** or **INDEXING MODE AUTOSELECT** options. If the **INDEXING MODE AUTOSELECT** option is specified and the load utility selects incremental maintenance of the indexes, the **USE** *tablespace-name* is ignored.

A load restart operation can use an alternate table space for building an index, even if the original load operation did not use an alternate table space. A load restart operation cannot be issued in **ALLOW READ ACCESS** mode if the original load operation was not issued in **ALLOW READ ACCESS** mode. Load terminate operations do not rebuild indexes, so the **USE** *tablespace-name* is ignored.

During the build phase of the load operation, the indexes are built in the system temporary table space. Then, during the index copy phase, the index is copied from the system temporary table space to the original index table space. To make sure that there is sufficient space in the original index table space for the new index, space is allocated in the original table space during the build phase. So, if the load operation runs out of index space, it will do so during the build phase. If this happens, the original index is not lost.

The index copy phase occurs after the build and delete phases. Before the index copy phase begins, the table is locked exclusively. That is, it is unavailable for read access throughout the index copy phase. Since the index copy phase is a physical copy, the table might be unavailable for a significant amount of time.

Note: If either the system temporary table space or the index table space are DMS table spaces, the read from the system temporary table space can cause random I/O on the system temporary table space and can cause a delay. The write to the index table space is still optimized and the **DISK_PARALLELISM** values are used.

Considerations for large indexes

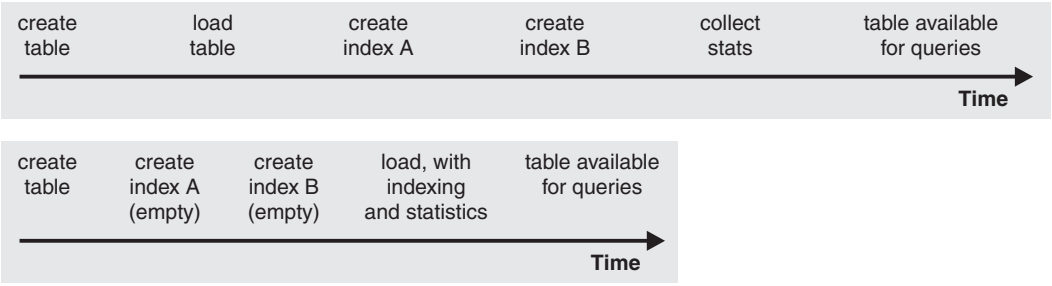
In order to improve performance when building large indexes during a load, it can be useful to tune the **sortheap** database configuration parameter. **sortheap** allocates the amount of memory dedicated to the sorting of index keys during a load operation. For example, to direct the load utility to use 4000 pages of main memory per index for key sorting, set **sortheap** to 4000 pages, disconnect all applications from the database, and then issue the **LOAD** command.

If an index is so large that it cannot be sorted in memory, a *sort spill*, or an overflow, occurs. That is, the data is divided among several "sort runs" and stored in a temporary table space that is merged later. Use the **sort_overflows** monitor element to determine whether a sort spill occurred. If there is no way to avoid a sort spill by increasing the size of the **sortheap** parameter, ensure that the buffer pool for temporary table spaces be large enough to minimize the amount of disk I/O that spilling causes. Furthermore, to achieve I/O parallelism during the merging of sort runs, it is recommended that temporary table spaces be declared with multiple containers, each residing on a different disk device. If there is more than one index defined on a table, memory consumption increases proportionally because the load operation keeps all keys in memory.

Deferring index creation

Generally speaking, it is more efficient to allow indexes to be created during the load operation by specifying either REBUILD or INCREMENTAL mode than it is to have index creation deferred. As Figure 11 on page 129 indicates, tables are normally built in three steps: data loading, index building, and statistics collection. This causes multiple data I/O during the load operation, index creation (there can be several indexes for each table),

and statistics collection (which causes I/O on the table data and on all of the indexes). A much faster alternative is to let the load utility complete all of these tasks in one pass through the data. It should be noted, however, that unique indexes reduce load performance if duplicates are encountered.



(artname: 00002439.gif)
Figure 11. Increasing load performance through concurrent indexing and statistics collection. Tables are normally built in three steps: data loading, index building, and statistics collection. This causes multiple data I/O during the load operation, during index creation (there can be several indexes for each table), and during statistics collection (which causes I/O on the table data and on all of the indexes). A much faster alternative is to let the load utility complete all of these tasks in one pass through the data.

At certain times, deferring index creation and invoking the CREATE INDEX statement can improve performance. Sorting during index rebuild uses up to **sortheap** pages. If more space is required, TEMP buffer pool is used and (eventually) spilled to disk. If load spills, and thus decreases performance, it might be advisable to run **LOAD** with **INDEXING MODE DEFERRED** and re-create the index later. The CREATE INDEX statement creates one index at a time, reducing memory usage while scanning the table multiple times to collect keys.

Another advantage of building indexes with a CREATE INDEX statement instead of concurrently with the load operation is that the CREATE INDEX statement can use multiple processes, or threads, to sort keys. The actual building of the index is not executed in parallel.

Compression dictionary creation during load operations

LOAD INSERT and **LOAD REPLACE** operations on tables for which compression is enabled can trigger the creation of compression dictionaries. Depending on what type of row compression a table uses, dictionary creation happens in different ways.

Classic row compression uses a single *table-level compression dictionary* to compress data. *Adaptive compression* uses multiple *page-level compression dictionaries* to compress individual pages of data, along with the table-level compression dictionaries used in classic row compression.

Page-level compression dictionaries

Page-level dictionaries are created and updated automatically during either **LOAD INSERT** or **LOAD REPLACE** operations; the **KEEPDICTIONARY** and **RESETDICTIONARY** options of the **LOAD** command have no effect on page-level dictionaries.

Table-level compression dictionaries

Table-level dictionaries are created automatically for both LOAD INSERT and LOAD REPLACE operations if no dictionary exists; however, if a table-level dictionary does exist, by default, the dictionary is not updated. More specifically,

LOAD REPLACE operations assume the **KEEPDICTIONARY** option by default. You can specify the **RESETDICTIONARY** option to remove the existing table-level dictionary and create a new one.

LOAD INSERT always follows the behavior implied by the **KEEPDICTIONARY** option.

When building table-level dictionaries for non-XML data, the load utility uses the data that exists in the target table to build the dictionaries, under the assumption that this preexisting data is representative of the type of data that will be stored in that table. In cases where there is insufficient preexisting data in the target table, the load utility builds the dictionaries once it has sampled enough input data. In this situation, the load utility uses only the input data to build the dictionary.

For XML data, the load utility samples incoming data only.

When dictionaries are created for range-partitioned tables, each partition is treated like an individual table. There will not be any cross-partition dictionaries and dictionary creation does not occur on partitions already containing dictionaries. For table data, the dictionary generated for each partition is based on the preexisting table data (and, if necessary, the loaded data) in that partition only. In Version 9.7 Fix Pack 1 and later, if the preexisting data in a partition is less than the minimum threshold, the dictionary is generated based only on the loaded data. For XML data, the dictionary generated for each partition is based the data being loaded into that partition.

LOAD REPLACE using the KEEPDICTIONARY option

A **LOAD REPLACE** that uses the **KEEPDICTIONARY** option keeps the existing dictionaries and uses them to compress the loaded data, as long as the target table has the COMPRESS attribute enabled. If dictionaries do not exist, the load utility generates new ones (provided the data that is being loaded into the table surpasses a predetermined threshold for table rows or XML documents stored in the default XML storage object) for tables with the COMPRESS attribute enabled. Since the data in the target table is replaced, the load utility uses only the input data to build the dictionaries. After a dictionary has been created, it is inserted into the table and the load operation continues.

LOAD REPLACE using the RESETDICTIONARY option

There are two key implications of using the **RESETDICTIONARY** option when loading into a table with the COMPRESS attribute on. First, dictionary creation occurs as long as any amount of data will exist in the target table once the **LOAD REPLACE** has completed. In other words, the new compression dictionaries can be based on a single row of data or a single XML document. The other implication is that the existing dictionaries are deleted but are not replaced (the target table will no longer have compression dictionaries) if any of the following situations are true:

- The operation is performed on a table with the COMPRESS attribute off
- Nothing was loaded (zero rows), in which case ADM5591W is printed to the notification log

Note: If you issue a **LOAD TERMINATE** operation after a **LOAD REPLACE** with the **RESETDICTIONARY** option, any existing compression dictionaries will be deleted and not replaced.

Performance impact

Dictionary creation affects the performance of a load operation in two ways:

- For **LOAD INSERT** operations, all of the preexisting table data, not just the minimum threshold for dictionary creation, is scanned before building the table-level compression dictionary. Therefore, the time used for this scan increases with table size.
- There is additional processing to build the compression dictionaries, although the time used for building the dictionaries is minimal.

While some operations related to the building of dictionaries can affect the CPU utilization by the **LOAD** command, load operations are generally I/O bound. That is, much of the time spent waiting for the load to complete is taken up waiting for data to be written to disk. The increased load on the CPU caused by dictionary creation generally does not increase the elapsed time required to perform the load; indeed, because data is written in compressed format, I/O times can actually decrease as compared to loading data into uncompressed tables.

Options for improving load performance

There are various command parameters that you can use to optimize load performance. There are also a number of file type modifiers unique to load which can, in some cases, significantly improve that utility's performance.

Command parameters

The load utility attempts to deliver the best performance possible by determining optimal values for DISK_PARALLELISM, CPU_PARALLELISM, and DATA BUFFER, if these parameters have not be specified by the user. Optimization is done based on the size and the free space available in the utility heap. Consider using the autonomic DISK_PARALLELISM and CPU_PARALLELISM settings before attempting to tune these parameters for your particular needs.

Following is information about the performance implications of various options available through the load utility:

ALLOW READ ACCESS

This option allows you to query a table while a load operation is in progress. You can only view data that existed in the table prior to the load operation. If the INDEXING MODE INCREMENTAL option is also specified, and the load operation fails, the subsequent load terminate operation might have to correct inconsistencies in the index. This requires an index scan which involves considerable I/O. If the ALLOW READ ACCESS option is also specified for the load terminate operation, the buffer pool is used for I/O.

Important: Starting with Version 10.1 Fix Pack 1, the ALLOW READ ACCESS parameter is deprecated and might be removed in a future release. For more details, see "ALLOW READ ACCESS parameter in the LOAD command is deprecated" at .

COPY YES or NO

Use this parameter to specify whether a copy of the input data is to be made during a load operation. COPY YES, which is only applicable when forward recovery is enabled, reduces load performance because all of the loading data is copied during the load operation. The increased I/O activity might increase the load time on an I/O-bound system. Specifying multiple devices or directories (on different disks) can offset some of the performance penalty resulting from this operation. COPY NO, which is only

applicable when forward recovery is enabled, does not affect load performance. However, all table spaces related to the loaded table will be placed in a Backup Pending state, and those table spaces must be backed up before the table can be accessed.

CPU_PARALLELISM

Use this parameter to exploit the number of processes running per database partition (if this is part of your machine's capability), and significantly improve load performance. The parameter specifies the number of processes or threads used by the load utility to parse, convert, and format data records. The maximum number allowed is 30. If there is insufficient memory to support the specified value, the utility adjusts the value. If this parameter is not specified, the load utility selects a default value that is based on the number of CPUs on the system.

Record order in the source data is preserved (see Figure 12) regardless of the value of this parameter, provided that:

- the `anyorder` file type modifier is not specified
- the PARTITIONING_DBPARTNUMS option (and more than one partition is to be used for partitioning) is not specified

If tables include either LOB or LONG VARCHAR data, `CPU_PARALLELISM` is set to 1. Parallelism is not supported in this case.

Although use of this parameter is not restricted to symmetric multiprocessor (SMP) hardware, you might not obtain any discernible performance benefit from using it in non-SMP environments.

(artname: 00002433.gif)

Figure 12. Record Order in the Source Data is Preserved When the Number of Processes Running Per Database Partition is Exploited During a Load Operation

DATA BUFFER

The `DATA BUFFER` parameter specifies the total amount of memory, in 4 KB units, allocated to the load utility as a buffer. It is recommended that this buffer be several *extents* in size. The data buffer is allocated from the utility heap; however, the data buffer can exceed the setting for the *util_heap_sz* database configuration parameter as long as there is available memory in the system.

DISK_PARALLELISM

The `DISK_PARALLELISM` parameter specifies the number of processes or threads used by the load utility to write data records to disk. Use this parameter to exploit available containers when loading data, and significantly improve load performance. The maximum number allowed is the greater of four times the `CPU_PARALLELISM` value (actually used by the load utility), or 50. By default, `DISK_PARALLELISM` is equal to the sum of the table space containers on all table spaces containing objects for the table being loaded, except where this value exceeds the maximum number allowed.

NONRECOVERABLE

If forward recovery is enabled, use this parameter if you do not need to be able to recover load transactions against a table upon rollforward. A

NONRECOVERABLE load and a COPY NO load have identical performance. However, there is a significant difference in terms of potential data loss. A NONRECOVERABLE load marks a table as not rollforward recoverable while leaving the table fully accessible. This can create a problematic situation in which if you need to rollforward through the load operation, then the loaded data as well as all subsequent updates to the table will be lost. A COPY NO load places all dependent table spaces in the Backup Pending state which renders the table inaccessible until a backup is performed. Because you are forced to take a backup after that type of load, you will not risk losing the loaded data or subsequent updates to the table. That is to say, a COPY NO load is totally recoverable.

Note: When these load transactions are encountered during subsequent restore and rollforward recovery operations, the table is not updated, and is marked invalid. Further actions against this table are ignored. After the rollforward operation is complete, the table can only be dropped.

SAVECOUNT

Use this parameter to set an interval for the establishment of consistency points **during the load phase** of a load operation. The synchronization of activities performed to establish a consistency point takes time. If done too frequently, there is a noticeable reduction in load performance. If a very large number of rows is to be loaded, it is recommended that a large SAVECOUNT value be specified (for example, a value of 10 million in the case of a load operation involving 100 million records).

A load restart operation automatically continues from the last consistency point, provided that the load restart operation resumes from the load phase.

STATISTICS USE PROFILE

Collect statistics specified in table statistics profile. Use this parameter to collect data distribution and index statistics more efficiently than through invocation of the RUNSTATS utility following completion of the load operation, even though performance of the load operation itself decreases (particularly when DETAILED INDEXES ALL is specified).

For optimal performance, applications require the best data distribution and index statistics possible. Once the statistics are updated, applications can use new access paths to the table data based on the latest statistics. New access paths to a table can be created by rebinding the application packages using the **BIND** command. The table statistics profile is created by running the **RUNSTATS** command with the SET PROFILE options.

When loading data into large tables, it is recommended that a larger value for the *stat_heap_sz* (statistics heap size) database configuration parameter be specified.

USE <tablespace-name>

When an ALLOW READ ACCESS load is taking place and the indexing mode is REBUILD, this parameter allows an index to be rebuilt in a system temporary table space and copied back to the index table space during the index copy phase of a load operation.

By default, the fully rebuilt index (also known as the *shadow index*) is built in the same table space as the original index. This might cause resource problems as both the original and the shadow index reside in the same table space simultaneously. If the shadow index is built in the same table space as the original index, the original index is instantaneously replaced

by the shadow. However, if the shadow index is built in a system temporary table space, the load operation requires an index copy phase which copies the index from a system temporary table space to the index table space. There is considerable I/O involved in the copy. If either of the table spaces is a DMS table space, the I/O on the system temporary table space might not be sequential. The values specified by the DISK_PARALLELISM option are respected during the index copy phase.

WARNINGCOUNT

Use this parameter to specify the number of warnings that can be returned by the utility before a load operation is forced to terminate. Set the WARNINGCOUNT parameter to a relatively low number if you are expecting only a few or no warnings. The load operation stops after the WARNINGCOUNT number is reached. This gives you the opportunity to correct problems before attempting to complete the load operation.

File type modifiers

ANYORDER

By default, the load utility preserves record order of source data. When load is operating under an SMP environment, synchronization between parallel processing is required to ensure that order is preserved.

In an SMP environment, specifying the anyorder file type modifier instructs the load utility to not preserve the order, which improves efficiency by avoiding the synchronization necessary to preserve that order. However, if the data to be loaded is presorted, anyorder might corrupt the presorted order, and the benefits of presorting are lost for subsequent queries.

Note: The anyorder file type modifier has no effect if CPU_PARALLELISM is 1, and it is not compatible with the SAVECOUNT option.

BINARYNUMERICS, ZONEDDECIMAL, and PACKEDDECIMAL

For fixed length non-delimited ASCII (ASC) source data, representing numeric data in binary can result in improved performance when loading. If the packeddecimal file type modifier is specified, decimal data is interpreted by the load utility to be in packed decimal format (two digits per byte). If the zoneddecimal file type modifier is specified, decimal data is interpreted by the load utility to be in zoned decimal format (one digit per byte). For all other numeric types, if the binarynumerics file type modifier is specified, data is interpreted by the load utility to be in binary format.

Note:

- When the binarynumerics, packeddecimal, or zoneddecimal file type modifiers are specified, numeric data is interpreted in big-endian (high byte first) format, regardless of platform.
- The packeddecimal and zoneddecimal file type modifiers are mutually exclusive.
- The packeddecimal and zoneddecimal file type modifiers only apply to the decimal target columns, and the binary data must match the target column definitions.
- The reclen file type modifier must be specified when the binarynumerics, packeddecimal, or zoneddecimal file type modifiers are specified.

FASTPARSE

Use with caution. In situations where the data being loaded is known to be valid, it can be unnecessary to have load perform the same amount of syntax checking as with more suspect data. In fact, decreasing the scope of this step can improve load's performance by about 10 or 20 percent. This can be done by using the `fastparse` file type modifier, which reduces the data checking that is performed on user-supplied column values from ASC and DEL files.

NOROWWARNINGS

During a load operation, warning messages about rejected rows are written to a specified file. However, if the load utility has to process a large volume of rejected, invalid or truncated records, it can adversely affect load's performance. In cases where many warnings are anticipated, it is useful to use the `norowwarnings` file type modifier to suppress the recording of these warnings.

PAGEFREESPACE, INDEXFREESPACE, and TOTALFREESPACE

As data is inserted and updated in tables over time, the need for table and index reorganization grows. One solution is to increase the amount of free space for tables and indexes using `pagefreespace`, `indexfreespace`, and `totalfreespace`. The first two modifiers, which take precedence over the PCTFREE value, specify the percentage of data and index pages that is to be left as free space, while `totalfreespace` specifies the percentage of the total number of pages that is to be appended to the table as free space.

Load features for maintaining referential integrity

Although the load utility is typically more efficient than the import utility, it requires a number of features to ensure the referential integrity of the information being loaded:

- **Table locks**, which provide concurrency control and prevent uncontrolled data access during a load operation
- **Table states** and **table space states**, which can either control access to data or elicit specific user actions
- **Load exception tables**, which ensure that rows of invalid data are not simply deleted without your knowledge

Checking for integrity violations following a load operation

Following a load operation, the loaded table might be in set integrity pending state in either READ or NO ACCESS mode if any of the following conditions exist:

- The table has table check constraints or referential integrity constraints defined on it.
- The table has generated columns and a V7 or earlier client was used to initiate the load operation.
- The table has descendent immediate materialized query tables or descendent immediate staging tables referencing it.
- The table is a staging table or a materialized query table.

The `STATUS` flag of the `SYSCAT.TABLES` entry corresponding to the loaded table indicates the set integrity pending state of the table. For the loaded table to be fully usable, the `STATUS` must have a value of `N` and the `ACCESS MODE` must have a value of `F`, indicating that the table is fully accessible and in normal state.

If the loaded table has descendent tables, the SET INTEGRITY PENDING CASCADE parameter can be specified to indicate whether or not the set integrity pending state of the loaded table should be immediately cascaded to the descendent tables.

If the loaded table has constraints as well as descendent foreign key tables, dependent materialized query tables and dependent staging tables, and if all of the tables are in normal state before the load operation, the following will result based on the load parameters specified:

INSERT, ALLOW READ ACCESS, and SET INTEGRITY PENDING CASCADE IMMEDIATE
The loaded table, its dependent materialized query tables and dependent staging tables are placed in set integrity pending state with read access.

INSERT, ALLOW READ ACCESS, and SET INTEGRITY PENDING CASCADE DEFERRED
Only the loaded table is placed in set integrity pending with read access. Descendent foreign key tables, descendent materialized query tables and descendent staging tables remain in their original states.

INSERT, ALLOW NO ACCESS, and SET INTEGRITY PENDING CASCADE IMMEDIATE
The loaded table, its dependent materialized query tables and dependent staging tables are placed in set integrity pending state with no access.

INSERT or REPLACE, ALLOW NO ACCESS, and SET INTEGRITY PENDING CASCADE DEFERRED
Only the loaded table is placed in set integrity pending state with no access. Descendent foreign key tables, descendent immediate materialized query tables and descendent immediate staging tables remain in their original states.

REPLACE, ALLOW NO ACCESS, and SET INTEGRITY PENDING CASCADE IMMEDIATE
The table and all its descendent foreign key tables, descendent immediate materialized query tables, and descendent immediate staging tables are placed in set integrity pending state with no access.

Note: Specifying the ALLOW READ ACCESS option in a load replace operation results in an error.

To remove the set integrity pending state, use the SET INTEGRITY statement. The SET INTEGRITY statement checks a table for constraints violations, and takes the table out of set integrity pending state. If all the load operations are performed in INSERT mode, the SET INTEGRITY statement can be used to incrementally process the constraints (that is, it checks only the appended portion of the table for constraints violations). For example:

```
db2 load from infile1.ixf of ixf insert into table1
db2 set integrity for table1 immediate checked
```

Only the appended portion of TABLE1 is checked for constraint violations. Checking only the appended portion for constraints violations is faster than checking the entire table, especially in the case of a large table with small amounts of appended data.

In IBM Data Studio Version 3.1 or later, you can use the task assistant for setting integrity. Task assistants can guide you through the process of setting options,

reviewing the automatically generated commands to perform the task, and running these commands. For more details, see Administering databases with task assistants.

If a table is loaded with the SET INTEGRITY PENDING CASCADE DEFERRED option specified, and the SET INTEGRITY statement is used to check for integrity violations, the descendent tables are placed in set integrity pending state with no access. To take the tables out of this state, you must issue an explicit request.

If a table with dependent materialized query tables or dependent staging tables is loaded using the INSERT option, and the SET INTEGRITY statement is used to check for integrity violations, the table is taken out of set integrity pending state and placed in No Data Movement state. This is done to facilitate the subsequent incremental refreshes of the dependent materialized query tables and the incremental propagation of the dependent staging tables. In the No Data Movement state, operations that might cause the movement of rows within the table are not allowed.

You can override the No Data Movement state by specifying the FULL ACCESS option when you issue the SET INTEGRITY statement. The table is fully accessible, however a full re-computation of the dependent materialized query tables takes place in subsequent REFRESH TABLE statements and the dependent staging tables are forced into an incomplete state.

If the ALLOW READ ACCESS option is specified for a load operation, the table remains in read access state until the SET INTEGRITY statement is used to check for constraints violations. Applications can query the table for data that existed before the load operation once it has been committed, but will not be able to view the newly loaded data until the SET INTEGRITY statement is issued.

Several load operations can take place on a table before checking for constraints violations. If all of the load operations are completed in ALLOW READ ACCESS mode, only the data that existed in the table before the first load operation is available for queries.

One or more tables can be checked in a single invocation of this statement. If a dependent table is to be checked on its own, the parent table can not be in set integrity pending state. Otherwise, both the parent table and the dependent table must be checked at the same time. In the case of a referential integrity cycle, all the tables involved in the cycle must be included in a single invocation of the SET INTEGRITY statement. It might be convenient to check the parent table for constraints violations while a dependent table is being loaded. This can only occur if the two tables are not in the same table space.

When issuing the SET INTEGRITY statement, you can specify the INCREMENTAL option to explicitly request incremental processing. In most cases, this option is not needed, because the Db2 database selects incremental processing. If incremental processing is not possible, full processing is used automatically. When the INCREMENTAL option is specified, but incremental processing is not possible, an error is returned if:

- New constraints are added to the table while it is in set integrity pending state.
- A load replace operation takes place, or the NOT LOGGED INITIALLY WITH EMPTY TABLE option is activated, after the last integrity check on the table.
- A parent table is load replaced or checked for integrity non-incrementally.

- The table is in set integrity pending state before an upgrade. Full processing is required the first time the table is checked for integrity after an upgrade.
- The table space containing the table or its parent is rolled forward to a point in time and the table and its parent reside in different table spaces.

If a table has one or more W values in the CONST_CHECKED column of the SYSCAT.TABLES catalog, and if the NOT INCREMENTAL option is not specified in the SET INTEGRITY statement, the table is incrementally processed and the CONST_CHECKED column of SYSCAT.TABLES is marked as U to indicate that not all data has been verified by the system.

The SET INTEGRITY statement does not activate any DELETE triggers as a result of deleting rows that violate constraints, but once the table is removed from set integrity pending state, triggers are active. Thus, if you correct data and insert rows from the exception table into the loaded table, any INSERT triggers defined on the table are activated. The implications of this should be considered. One option is to drop the INSERT trigger, insert rows from the exception table, and then re-create the INSERT trigger.

Table locking during load operations

In most cases, the load utility uses table level locking to restrict access to tables. The level of locking depends on the stage of the load operation and whether it was specified to allow read access.

A load operation in ALLOW NO ACCESS mode uses a super exclusive lock (Z-lock) on the table for the duration of the load.

Before a load operation in ALLOW READ ACCESS mode begins, the load utility waits for all applications that began before the load operation to release their locks on the target table. At the beginning of the load operation, the load utility acquires an update lock (U-lock) on the table. It holds this lock until the data is being committed. When the load utility acquires the U-lock on the table, it waits for all applications that hold locks on the table before the start of the load operation to release them, even if they have compatible locks. This is achieved by temporarily upgrading the U-lock to a Z-lock which does not conflict with new table lock requests on the target table as long as the requested locks are compatible with the load operation's U-lock. When data is being committed, the load utility upgrades the lock to a Z-lock, so there can be some delay in commit time while the load utility waits for applications with conflicting locks to finish.

Note: The load operation can time out while it waits for the applications to release their locks on the table before loading. However, the load operation does not time out while waiting for the Z-lock needed to commit the data.

Applications with conflicting locks

Use the LOCK WITH FORCE option of the **LOAD** command to force off applications holding conflicting locks on a target table so that the load operation can proceed. Before a load operation in ALLOW READ ACCESS mode can proceed, applications holding the following locks are forced off:

- Table locks that conflict with a table update lock (for example, import or insert).
- All table locks that exist at the commit phase of the load operation.

Applications holding conflicting locks on the system catalog tables are not forced off by the load utility. If an application is forced off the system by the load utility, the application loses its database connection, and an error is returned (SQL1224N).

When you specify the COPY NO option for a load operation on a recoverable database, all objects in the target table space are locked in share mode before the table space is placed in the Backup Pending state. This occurs regardless of the access mode. If you specify the LOCK WITH FORCE option, all applications holding locks on objects in the table space that conflict with a share lock are forced off.

Read access load operations

The load utility provides two options that control the amount of access other applications have to a table being loaded. The ALLOW NO ACCESS option locks the table exclusively and allows no access to the table data while the table is being loaded.

The ALLOW NO ACCESS option is the default behavior. The ALLOW READ ACCESS option prevents all write access to the table by other applications, but allows read access to preexisting data. This section deals with the ALLOW READ ACCESS option.

Important: Starting with Version 10.1 Fix Pack 1, the ALLOW READ ACCESS parameter is deprecated and might be removed in a future release. For more details, see "ALLOW READ ACCESS parameter in the LOAD command is deprecated" at .

Table data and index data that exist before the start of a load operation are visible to queries while the load operation is in progress. Consider the following example:

1. Create a table with one integer column:

   ```
   create table ED (ed int)
   ```

2. Load three rows:

   ```
   load from File1 of del insert into ED
   ...
   Number of rows read         = 3
   Number of rows skipped      = 0
   Number of rows loaded       = 3
   Number of rows rejected     = 0
   Number of rows deleted      = 0
   Number of rows committed    = 3
   ```

3. Query the table:

   ```
   select * from ED

   ED
   -----------
             1
             2
             3

     3 record(s) selected.
   ```

4. Perform a load operation with the ALLOW READ ACCESS option specified and load two more rows of data:

   ```
   load from File2 of del insert into ED allow read access
   ```

5. At the same time, on another connection query the table while the load operation is in progress:

   ```
   select * from ED

   ED
   -----------
             1
             2
             3

     3 record(s) selected.
   ```

6. Wait for the load operation to finish and then query the table:

```
select * from ED

ED
-----------
          1
          2
          3
          4
          5

  5 record(s) selected.
```

The ALLOW READ ACCESS option is very useful when loading large amounts of data because it gives users access to table data at all times, even when the load operation is in progress or after a load operation has failed. The behavior of a load operation in ALLOW READ ACCESS mode is independent of the isolation level of the application. That is, readers with any isolation level can always read the preexisting data, but they are not be able to read the newly loaded data until the load operation has finished.

Read access is provided throughout the load operation except for two instances: at the beginning and at the end of the operation.

Firstly, the load operation acquires a special Z-lock for a short duration of time near the end of its setup phase. If an application holds an incompatible lock on the table before the load operation requesting this special Z-lock, then the load operation waits a finite amount of time for this incompatible lock to be released before timing out and failing. The amount of time is determined by the *locktimeout* database configuration parameter. If the LOCK WITH FORCE option is specified then the load operation forces other applications off to avoid timing out. The load operation acquires the special Z-lock, commits the phase, releases the lock, and then continues onto the load phase. Any application that requests a lock on the table for reading after the start of the load operation in ALLOW READ ACCESS mode is granted the lock, and it does not conflict with this special Z-lock. New applications attempting to read existing data from the target table are able to do so.

Secondly, before data is committed at the end of the load operation, the load utility acquires an exclusive lock (Z-lock) on the table. The load utility waits until all applications that hold locks on the table release them. This can cause a delay before the data is committed. The LOCK WITH FORCE option is used to force off conflicting applications, and allow the load operation to proceed without having to wait. Usually, a load operation in ALLOW READ ACCESS mode acquires an exclusive lock for a short amount of time; however, if the USE <tablespace-name> option is specified, the exclusive lock lasts for the entire period of the index copy phase.

When the load utility is running against a table defined on multiple database partitions, the load process model executes on each individual database partition, meaning that locks are acquired and released independently of other db-partitions. Thus, if a query or other operation is executed concurrently and is competing for the same locks, there is a chance for deadlocks. For example, suppose that operation A is granted a table lock on db-partition 0 and the load operation is granted a table lock on db-partition 1. A deadlock can occur because operation A is waiting to be granted a table lock on db-partition 1, while the load operation is waiting for a table lock on db-partition 0. In this case, the deadlock detector will arbitrarily roll back one of the operations.

Note:

1. If a load operation is interrupted or fails, it remains at the same access level that was specified when the load operation was issued. That is, if a load operation in ALLOW NO ACCESS mode fails, the table data is inaccessible until a load terminate or a load restart is issued. If a load operation in ALLOW READ ACCESS mode aborts, the preexisting table data is still accessible for read access.
2. If the ALLOW READ ACCESS option was specified for an interrupted or failed load operation, it can also be specified for the load restart or load terminate operation. However, if the interrupted or failed load operation specified the ALLOW NO ACCESS option, the ALLOW READ ACCESS option cannot be specified for the load restart or load terminate operation.

The ALLOW READ ACCESS option is not supported if:

- The REPLACE option is specified. Since a load replace operation truncates the existing table data before loading the new data, there is no preexisting data to query until after the load operation is complete.
- The indexes have been marked invalid and are waiting to be rebuilt. Indexes can be marked invalid in some rollforward scenarios or through the use of the **db2dart** command.
- The INDEXING MODE DEFERRED option is specified. This mode marks the indexes as requiring a rebuild.
- An ALLOW NO ACCESS load operation is being restarted or terminated. Until it is brought fully online, a load operation in ALLOW READ ACCESS mode cannot take place on the table.
- A load operation is taking place to a table that is in Set Integrity Pending No Access state. This is also the case for multiple load operations on tables with constraints. A table is not brought online until the SET INTEGRITY statement is issued.

Generally, if table data is taken offline, read access is not available during a load operation until the table data is back online.

Table space states during and after load operations

The load utility uses table space states to preserve database consistency during a load operation. These states work by controlling access to data or eliciting user actions.

The load utility does not quiesce (put persistent locks on) the table spaces involved in the load operation and uses table space states only for load operations for which you specify the **COPY NO** parameter.

You can check table space states by using the **LIST TABLESPACES** command. Table spaces can be in multiple states simultaneously. The states returned by **LIST TABLESPACES** are as follows:

Normal

The Normal state is the initial state of a table space after it is created, indicating that no (abnormal) states currently affect it.

Load in Progress

The Load in Progress state indicates that there is a load in progress on the table space. This state prevents the backup of dependent tables during the load. The table space state is distinct from the Load in Progress table state (which is used in all load operations) because the load utility places table

spaces in the Load in Progress state only when you specify the **COPY NO** parameter for a recoverable database. The table spaces remain in this state for the duration of the load operation.

Backup Pending

If you perform a load operation for a recoverable database and specify the **COPY NO** parameter, table spaces are placed in the Backup Pending table space state after the first commit. You cannot update a table space in the Backup Pending state. You can remove the table space from the Backup Pending state only by backing up the table space. Even if you cancel the load operation, the table space remains in the Backup Pending state because the table space state is changed at the beginning of the load operation and cannot be rolled back.

Restore Pending

If you perform a successful load operation with the **COPY NO** option, restore the database, and then rollforward through that operation, the associated table spaces are placed in the Restore Pending state. To remove the table spaces from the Restore Pending state, you must perform a restore operation.

Note: Db2 LOAD does not set the table space state to **Load Pending** or **Delete Pending**.

Example of a table space state

If you load an input file (`staffdata.del`) into a table NEWSTAFF, as follows:

```
update db cfg for sample using logarchmeth1 logretain;
backup db sample;
connect to sample;
create table newstaff like staff;
load from staffdata.del of del insert into newstaff copy no;
connect reset;
```

and you open another session and issue the following commands,

```
connect to sample;
list tablespaces;
connect reset;
```

USERSPACE1 (the default table space for the sample database) is in the Load in Progress state and, after the first commit, the Backup Pending state as well. After the load operation finishes, the **LIST TABLESPACES** command reveals that USERSPACE1 is now in the Backup Pending state:

```
Tablespace ID                        = 2
Name                                 = USERSPACE1
Type                                 = Database managed space
Contents                             = All permanent data. Large table space.
State                                = 0x0020
  Detailed explanation:
  Backup pending
```

Table states during and after load operations

The load utility uses table states to preserve database consistency during a load operation. These states work by controlling access to data or eliciting user actions.

To determine the state of a table, issue the **LOAD QUERY** command, which also checks the status of a load operation. Tables can be in a number of states simultaneously. The states returned by **LOAD QUERY** are as follows:

Normal State
The Normal state is the initial state of a table after it is created, indicating that no (abnormal) states currently affect the table.

Read Access Only
If you specify the **ALLOW READ ACCESS** option, the table is in the Read Access Only state. The data in the table that existed before the invocation of the load command is available in read-only mode during the load operation. If you specify the **ALLOW READ ACCESS** option and the load operation fails, the data that existed in the table before the load operation continues to be available in read-only mode after the failure.

Load in Progress
The Load in Progress table state indicates that there is a load in progress on the table. The load utility removes this transient state after the load is successfully completed. However, if the load operation fails or is interrupted, the table state will change to Load Pending.

Redistribute in Progress
The Redistribute in Progress table state indicates that there is a redistribute in progress on the table. The redistribute utility removes this transient state after it has successfully completed processing the table. However, if the redistribute operation fails or is interrupted, the table state will change to Redistribute Pending.

Load Pending
The Load Pending table state indicates that a load operation failed or was interrupted. You can take one of the following steps to remove the Load Pending state:

- Address the cause of the failure. For example, if the load utility ran out of disk space, add containers to the table space. Then, restart the load operation.
- Terminate the load operation.
- Run a load **REPLACE** operation against the same table on which the load operation failed.
- Recover table spaces for the loading table by using the **RESTORE DATABASE** command with the most recent table space or database backup, then carry out further recovery actions.

Redistribute Pending
The Redistribute Pending table state indicates that a redistribute operation failed or was interrupted. You can perform a **REDISTRIBUTE CONTINUE** or **REDISTRIBUTE ABORT** operation to remove the Redistribute Pending state.

Not Load Restartable
In the Not Load Restartable state, a table is partially loaded and does not allow a load restart operation. There are two situations in which a table is placed in the Not Load Restartable state:

- If you perform a rollforward operation after a failed load operation that you could not successfully restart or terminate
- If you perform a restore operation from an online backup that you took while the table was in the Load in Progress or Load Pending state

The table is also in the Load Pending state. To remove the table from the Not Load Restartable state, issue the **LOAD TERMINATE** or the **LOAD REPLACE** command.

Set Integrity Pending
The Set Integrity Pending state indicates that the loaded table has constraints which have not yet been verified. The load utility places a table in this state when it begins a load operation on a table with constraints. Use the SET INTEGRITY statement to take the table out of Set Integrity Pending state.

Type-1 indexes
The Type-1 Indexes state indicates that the table currently uses type-1 indexes. Type-1 indexes are no longer supported since Version 9.7. You should convert them to type-2 indexes before upgrading to Version 10. Otherwise, the type-1 indexes are automatically rebuilt as type-2 indexes the first time a table is accessed.

For details on how to convert type-1 indexes before upgrading databases, see the "Converting type-1 indexes to type-2 indexes" topic.

Unavailable
Rolling forward through an unrecoverable load operation places a table in the Unavailable state. In this state, the table is unavailable; you must drop it or restore it from a backup.

Example of a table in multiple states

If you load an input file (`staffdata.del`) with a substantial amount of data into a table NEWSTAFF, as follows:

```
connect to sample;
create table newstaff like staff;
load from staffdata.del of del insert into newstaff allow read access;
connect reset;
```

and you open another session and issue the following commands,

```
connect to sample;
load query table newstaff;
connect reset;
```

the **LOAD QUERY** command reveals that the NEWSTAFF table is in the Read Access Only and Load in Progress table states:

```
Tablestate:
 Load in Progress
 Read Access Only
```

Load exception tables

A load exception table is a consolidated report of all of the rows that violated unique index rules, range constraints, and security policies during a load operation. You specify a load exception table by using the FOR EXCEPTION clause of the **LOAD** command.

Restriction: An exception table cannot contain an identity column or any other type of generated column. If an identity column is present in the primary table, the corresponding column in the exception table should only contain the column's type, length, and nullability attributes. In addition, the exception table cannot be a range-partitioned table or a column-organized table or have a unique index. Moreover, you cannot specify an exception table if either of the following conditions is true:

- The target table uses LBAC security and has at least one XML column
- The target table is range partitioned and has at least one XML column

The exception table used with the load utility is identical to the exception tables used by the SET INTEGRITY statement. It is a user-created table that reflects the definition of the table being loaded and includes some additional columns.

You can assign a load exception table to the table space where the table being loaded resides or to another table space. In either case, assign the load exception table and the table being loaded to the same database partition group, and ensure that both tables use the same distribution key. Additionally, ensure that the exception table and table being loaded have the same partition map id (SYSIBM.SYSTABLES.PMAP_ID), which can potentially be different during the redistribute operation (add/drop database partition operation).

When to use an exception table

Use an exception table when loading data that has a unique index and could have duplicate records. If you do not specify an exception table and duplicate records are found, the load operation continues, and only a warning message is issued about the deleted duplicate records. The duplicate records are not logged.

After the load operation is completed, you can use information in the exception table to correct data that is in error. You can then insert the corrected data into the table.

Rows are appended to existing information in the exception table. Because there is no checking done to ensure that the table is empty, new information is simply added to the invalid rows from previous load operations. If you want only the invalid rows from the current load operation, you can remove the existing rows before invoking the utility. Alternatively, when you define a load operation, you can specify that the exception table record the time when a violation is discovered and the name of the constraint violated.

Because each deletion event is logged, the log could fill up during the delete phase of the load if there are a large number of records that violate a uniqueness condition.

Any rows rejected because of invalid data before the building of an index are not inserted into the exception table.

Failed or incomplete loads

Restarting an interrupted load operation

If a failure or interruption occurs during a load operation, you can use the load utility to terminate the operation, reload the table, or restart the load operation.

If the load utility does not even start because of a user error such as a nonexistent data file or invalid column names, the operation terminates and leaves the target table in a normal state.

When the load operation begins, the target table is placed in the Load in Progress table state. In the event of a failure, the table state will change to Load Pending. To remove the table from this state, you can issue a **LOAD TERMINATE** to roll back the operation, issue a **LOAD REPLACE** to reload the entire table, or issue a **LOAD RESTART**.

Typically, restarting the load operation is the best choice in this situation. It saves time because the load utility restarts the load operation from the last successfully reached point in its progress, rather than from the beginning of the operation.

Where exactly the operation restarts from depends upon the parameters specified in the original command. If the SAVECOUNT option was specified, and the previous load operation failed in the load phase, the load operation restarts at the last consistency point it reached. Otherwise, the load operation restarts at the beginning of the last phase successfully reached (the load, build, or delete phase).

If you are loading XML documents, the behavior is slightly different. Because the SAVECOUNT option is not supported with loading XML data, load operations that fail during the load phase restart from the beginning of the operation. Just as with other data types, if the load fails during the build phase, indexes are built in REBUILD mode, so the table is scanned to pick up all index keys from each row; however, each XML document must also be scanned to pick up the index keys. This process of scanning XML documents for keys requires them to be reparsed, which is an expensive operation. Furthermore, the internal XML indexes, such as the regions and paths indexes, need to be rebuilt first, which also requires a scan of the XDA object.

Once you have fixed the situation that caused the load operation to fail, reissue the load command. Ensure that you specify exactly the same parameters as in the original command, so that the load utility can find the necessary temporary files. An exception to this is if you want to disallow read access. A load operation that specified the ALLOW READ ACCESS option can also be restarted as an ALLOW NO ACCESS option.

Note: Do not delete or modify any temporary files created by the load utility.

If the load operation resulting from the following command fails,

```
LOAD FROM file_name OF file_type
SAVECOUNT n
MESSAGES message_file
load_method
INTO target_tablename
```

you would restart it by replacing the specified load method (load_method) with the RESTART method:

```
LOAD FROM file_name OF file_type
SAVECOUNT n
MESSAGES message_file
RESTART
INTO target_tablename
```

Failed loads that cannot be restarted

You cannot restart failed or interrupted load operations if the table involved in the operation is in the Not Load Restartable table state. Tables are put in that state for the following reasons:

- A rollforward operation is performed after a failed load operation that has not been successfully restarted or terminated
- A restore operation is performed from an online backup that was taken while the table was in the Load in Progress or Load Pending table state

You should issue either a **LOAD TERMINATE** or a **LOAD REPLACE** command.

Failed load limitations

The **BACKUP DATABASE** command might return an I/O error if the **LOAD** command fails on a table in SMS tablespace and the table is left in Load Pending state.

Table data might not appear consistent when a table is in Load Pending state. Inconsistent table data will cause the **BACKUP DATABASE** command to fail. The table will remain inconsistent until a subsequent **LOAD TERMINATE**, **LOAD RESTART**, or **LOAD REPLACE** command is completed.

You must remove the table from the Load Pending state before backing up your database.

Restarting or terminating an ALLOW READ ACCESS load operation

An interrupted or canceled load operation that specifies the **ALLOW READ ACCESS** parameter can also be restarted or terminated using the **ALLOW READ ACCESS** parameter. Using the **ALLOW READ ACCESS** parameter allows other applications to query the table data while the terminate or restart operation is in progress. As with a load operation in **ALLOW READ ACCESS** mode, the table is locked exclusively before the data being committed.

About this task

If the index object is unavailable or marked invalid, a load restart or terminate operation in **ALLOW READ ACCESS** mode is not permitted.

If the original load operation is interrupted or canceled in the index copy phase, a restart operation in the **ALLOW READ ACCESS** mode is not permitted because the index might be corrupted.

If a load operation in **ALLOW READ ACCESS** mode is interrupted or canceled in the load phase, it restarts in the load phase. If it is interrupted or canceled in any phase other than the load phase, it restarts in the build phase. If the original load operation is in **ALLOW NO ACCESS** mode, a restart operation occurs in the delete phase if the original load operation reaches that point and the index is valid. If the index is marked invalid, the load utility restarts the load operation from the build phase.

Note: All load restart operations choose the REBUILD indexing mode even if the **INDEXING MODE INCREMENTAL** parameter is specified.

Issuing a **LOAD TERMINATE** command generally causes the interrupted or canceled load operation to be rolled back with minimal delay. However, when issuing a **LOAD TERMINATE** command for a load operation where **ALLOW READ ACCESS** and **INDEXING MODE INCREMENTAL** are specified, there might be a delay while the load utility scans the indexes and corrects any inconsistencies. The length of this delay depends on the size of the indexes and occurs whether the **ALLOW READ ACCESS** parameter is specified for the load terminate operation. The delay does not occur if the original load operation failed before the build phase.

Note: The delay resulting from corrections to inconsistencies in the index is considerably less than the delay caused by marking the indexes as invalid and rebuilding them.

A load restart operation cannot be undertaken on a table that is in the Not Load Restartable table state. A table can be placed in the Not Load Restartable table state during a rollforward operation. This can occur if you roll forward to a point in time that is before the end of a load operation, or if you roll forward through an interrupted or canceled load operation but do not roll forward to the end of the load terminate or load restart operation.

Important: Starting with Version 10.1 Fix Pack 1, the ALLOW READ ACCESS parameter is deprecated and might be removed in a future release. For more details, see "ALLOW READ ACCESS parameter in the LOAD command is deprecated" at .

Recovering data with the load copy location file

The **DB2LOADREC** registry variable is used to identify the file with the load copy location information. This file is used during rollforward recovery to locate the load copy.

DB2LOADREC has information about:

- Media type
- Number of media devices to be used
- Location of the load copy generated during a table load operation
- File name of the load copy, if applicable

If the location file does not exist, or no matching entry is found in the file, the information from the log record is used.

The information in the file might be overwritten before rollforward recovery takes place.

Note:

1. In a multi-partition database, the **DB2LOADREC** registry variable must be set for all the database partition servers using the **db2set** command.
2. In a multi-partition database, the load copy file must exist at each database partition server, and the file name (including the path) must be the same.
3. If an entry in the file identified by the **DB2LOADREC** registry variable is not valid, the old load copy location file is used to provide information to replace the invalid entry.

The following information is provided in the location file. The first five parameters must have valid values, and are used to identify the load copy. The entire structure is repeated for each load copy recorded. For example:

```
TIMestamp      19950725182542         * Time stamp generated at load time
DBPartition    0                      * DB Partition number (OPTIONAL)
SCHema         PAYROLL                * Schema of table loaded
TABlename      EMPLOYEES              * Table name
DATabasename   DBT                    * Database name
DB2instance    toronto                * DB2INSTANCE
BUFfernumber   NULL                   * Number of buffers to be used for
                                        recovery
SESsionnumber  NULL                   * Number of sessions to be used for
                                        recovery
TYPeofmedia    L                      * Type of media - L for local device
                                                       A for TSM
                                                       O for other vendors
LOCationnumber 3                      * Number of locations
   ENTry       /u/toronto/dbt.payroll.employes.001
   ENT         /u/toronto/dbt.payroll.employes.002
   ENT         /dev/rmt0
TIM            19950725192054
DBP            18
SCH            PAYROLL
TAB            DEPT
DAT            DBT
DB2            toronto
BUF            NULL
```

```
SES             NULL
TYP             A
TIM             19940325192054
SCH             PAYROLL
TAB             DEPT
DAT             DBT
DB2             toronto
BUF             NULL
SES             NULL
TYP             O
SHRlib          /@sys/lib/backup_vendor.a
```

Note:

1. The first three characters in each keyword are significant. All keywords are required in the specified order. Blank lines are not accepted.
2. The time stamp is in the form *yyyymmddhhmmss*.
3. All fields are mandatory, except for BUF and SES (which can be NULL), and DBP (which can be missing from the list). If SES is NULL, the value specified by the *dft_loadrec_ses* configuration parameter is used. If BUF is NULL, the default value is SES+2.
4. If even one of the entries in the location file is invalid, the previous load copy location file is used to provide those values.
5. The media type can be local device (`L` for tape, disk or diskettes), TSM (`A`), or other vendor (`O`). If the type is `L`, the number of locations, followed by the location entries, is required. If the type is `A`, no further input is required. If the type is `O`, the shared library name is required.
6. The `SHRlib` parameter points to a library that has a function to store the load copy data.
7. If you invoke a load operation, specifying the `COPY NO` or the `NONRECOVERABLE` option, and do not take a backup copy of the database or affected table spaces after the operation completes, you cannot restore the database or table spaces to a point in time that follows the load operation. That is, you cannot use rollforward recovery to re-create the database or table spaces to the state they were in following the load operation. You can only restore the database or table spaces to a point in time that precedes the load operation.

If you want to use a particular load copy, you can use the recovery history file for the database to determine the time stamp for that specific load operation. In a multi-partition database, the recovery history file is local to each database partition.

Load dump file

Specifying the `dumpfile` file type modifier tells the load utility the name and the location of the exception file to which rejected rows are written.

When running in a partitioned database environment, rows can be rejected either by the partitioning subagents or by the loading subagents. Because of this, the dump file name is given an extension that identifies the subagent type, as well as the database partition number where the exceptions were generated. For example, if you specified the following dump file value:

```
dumpfile = "/u/usrname/dumpit"
```

Then rows that are rejected by the load subagent on database partition five will be stored in a file named `/u/usrname/dumpit.load.005`, rows that are rejected by the load Subagent on database partition two will be stored in a file named

/u/usrname/dumpit.load.002, and rows that are rejected by the partitioning subagent on database partition two will be stored in a file named /u/usrname/dumpit.part.002, and so on.

For rows rejected by the load subagent, if the row is less than 32 768 bytes in length, the record is copied to the dump file in its entirety; if it is longer, a row fragment (including the final bytes of the record) is written to the file.

For rows rejected by the partitioning subagent, the entire row is copied to the dump file regardless of the record size.

Load temporary files

The Db2 database system creates temporary binary files during load processing. These files are used for load crash recovery, load terminate operations, warning and error messages, and runtime control data.

Load temporary files are removed when the load operation completes without error. The temporary files are written to a path that can be specified through the *temp-pathname* parameter of the LOAD command, or in the *piTempFilesPath* parameter of the **db2Load** API. The default path is a subdirectory of the partition-global directory.

Load operations against different databases must not specify the same temporary files path.

The temporary files path resides on the server machine and is accessed by the Db2 instance exclusively. Therefore, it is imperative that any path name qualification given to the *temp-pathname* parameter reflects the directory structure of the server, not the client, and that the Db2 instance owner has read and write permission on the path.

Note: In a Db2 pureScale® environment, the load temporary files should reside on a path that is accessible by all members (for example, on a shared disk). The temporary files need to be on a shared disk, otherwise member crash recovery and **LOAD TERMINATE** operations executed from a different member might have issues.

This is different from a partitioned database environment, where the load temporary files path should reside on a local disk. You should avoid choosing a Network File System (NFS) based path, otherwise there is significant performance degradation during the load operation.

Attention: The temporary files written to this path must not be tampered with under any circumstances. Doing so causes the load operation to malfunction and places your database in jeopardy.

Load utility log records

The utility manager produces log records associated with a number of Db2 utilities, including the load utility.

The following log records mark the beginning or end of a specific activity during a load operation:

- **Setup phase**
  - Load Start. This log record signifies the beginning of a load operation's setup phase.

  - Commit log record. This log record signifies the successful completion of the setup phase.
  - Abort log record. This log record signifies the failure of the setup phase. (Alternately, in a single partition database, if the Load setup phase fails prior to physically modifying the table, it will generate a Local Pending commit log record).
- **Load phase**
  - Load Start. This log record signifies the beginning of a load operation's load phase.
  - Local Pending commit log record. This log record signifies the successful completion of the load phase.
  - Abort log record. This log record signifies the failure of the load phase.
- **Delete phase**
  - Load Delete Start. This log record is associated with the beginning of the delete phase in a load operation. The delete phase is started only if there are duplicate primary key values. During the delete phase, each delete operation on a table record, or an index key, is logged.
  - Load Delete End. This log record is associated with the end of the delete phase in a load operation. This delete phase is repeated during the rollforward recovery of a successful load operation.

The following list outlines the log records that the load utility creates depending on the size of the input data:

- Two log records are created for every table space extent allocated or deleted by the utility in a DMS table space.
- One log record is created for every chunk of identity values consumed.
- Log records are created for every data row or index key deleted during the delete phase of a load operation.
- Log records are created that maintain the integrity of the index tree when performing a load operation with the `ALLOW READ ACCESS` and `INDEXING MODE INCREMENTAL` options specified. The number of records logged is considerably less than a fully logged insertion into the index.

Load overview-partitioned database environments

In a multi-partition database, large amounts of data are located across many database partitions. Distribution keys are used to determine on which database partition each portion of the data resides. The data must be *distributed* before it can be loaded at the correct database partition.

When loading tables in a multi-partition database, the load utility can:

- Distribute input data in parallel
- Load data simultaneously on corresponding database partitions
- Transfer data from one system to another system

Loading data into a multi-partition database takes place in two phases: the *setup phase*, during which database partition resources such as table locks are acquired, and the *load phase*, during which the data is loaded into the database partitions. You can use the `ISOLATE_PART_ERRS` option of the **LOAD** command to select how errors are handled during either of these phases, and how errors on one or more of the database partitions affect the load operation on the database partitions that are not experiencing errors.

When loading data into a multi-partition database you can use one of the following modes:

PARTITION_AND_LOAD
Data is distributed (perhaps in parallel) and loaded simultaneously on the corresponding database partitions. When loading into a random distribution table that uses the random by generation method, this is the only supported mode.

PARTITION_ONLY
Data is distributed (perhaps in parallel) and the output is written to files in a specified location on each loading database partition. Each file includes a partition header that specifies how the data was distributed across the database partitions, and that the file can be loaded into the database using the LOAD_ONLY mode.

LOAD_ONLY
Data is assumed to be already distributed across the database partitions; the distribution process is skipped, and the data is loaded simultaneously on the corresponding database partitions.

LOAD_ONLY_VERIFY_PART
Data is assumed to be already distributed across the database partitions, but the data file does not contain a partition header. The distribution process is skipped, and the data is loaded simultaneously on the corresponding database partitions. During the load operation, each row is checked to verify that it is on the correct database partition. Rows containing database partition violations are placed in a dump file if the `dumpfile` file type modifier is specified. Otherwise, the rows are discarded. If database partition violations exist on a particular loading database partition, a single warning is written to the load message file for that database partition.

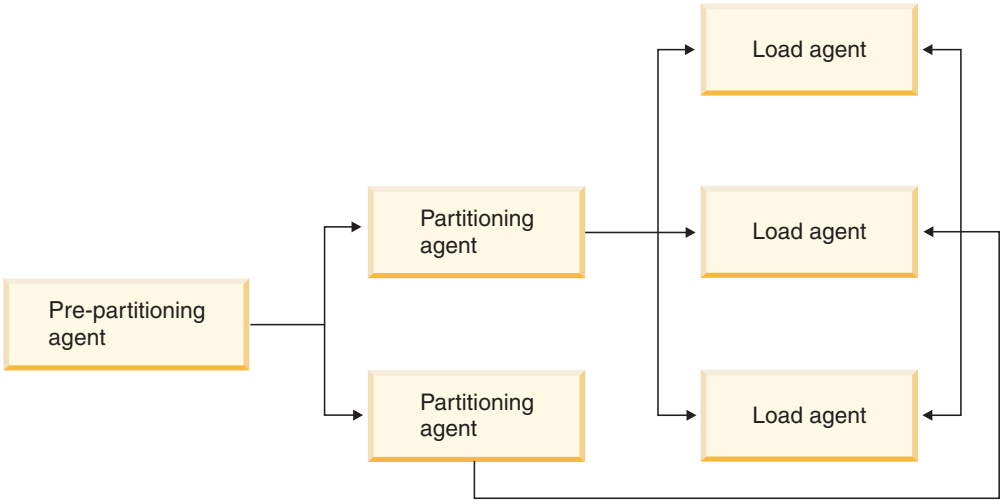
ANALYZE
An optimal distribution map with even distribution across all database partitions is generated.

Concepts and terminology

The following terminology is used when discussing the behavior and operation of the load utility in a partitioned database environment with multiple database partitions:

- The *coordinator partition* is the database partition to which the user connects in order to perform the load operation. In the PARTITION_AND_LOAD, PARTITION_ONLY, and ANALYZE modes, it is assumed that the data file resides on this database partition unless the `CLIENT` option of the **LOAD** command is specified. Specifying `CLIENT` indicates that the data to be loaded resides on a remotely connected client.
- In the PARTITION_AND_LOAD, PARTITION_ONLY, and ANALYZE modes, the *pre-partitioning agent* reads the user data and distributes it to the next agent in the pipeline. The actual agent depends on the distribution method.
  - For random distribution tables using random by generation method, the data is distributed in a round-robin fashion directly to the loading agents.
  - Otherwise, data is distributed in a round-robin fashion to the partitioning agents which then distribute the data. This process is always performed on the coordinator partition. A maximum of one partitioning agent is allowed per database partition for any load operation.

- In the PARTITION_AND_LOAD, LOAD_ONLY, and LOAD_ONLY_VERIFY_PART modes, *load agents* run on each output database partition and coordinate the loading of data to that database partition.
- *Load to file agents* run on each output database partition during a PARTITION_ONLY load operation. They receive data from partitioning agents and write it to a file on their database partition.
- The `SOURCEUSEREXIT` option provides a facility through which the load utility can execute a customized script or executable, referred to herein as the *user exit*.



(artname: 00002435.gif)
Figure 13. Partitioned Database Load Overview. The source data is read by the pre-partitioning agent, and approximately half of the data is sent to each of two partitioning agents which distribute the data and send it to one of three database partitions. The load agent at each database partition loads the data.

Loading data in a partitioned database environment

Using the load utility to load data into a partitioned database environment.

Before you begin

Before loading a table in a multi-partition database:

- Ensure that the **svcename** database manager configuration parameter and the **DB2COMM** profile registry variable are set correctly. This step is important because the load utility uses TCP/IP to transfer data from the pre-partitioning agent to the partitioning agents, and from the partitioning agents to the loading database partitions.
- Before invoking the load utility, you must be connected to (or be able to implicitly connect to) the database into which you want to load the data.
- Since the load utility issues a COMMIT statement, complete all transactions and release any locks by issuing either a COMMIT or a ROLLBACK statement before beginning the load operation. If the `PARTITION_AND_LOAD`, `PARTITION_ONLY`, or `ANALYZE` mode is being used, the data file that is being loaded must reside on this database partition unless:
  1. The **CLIENT** parameter has been specified, in which case the data must reside on the client machine;
  2. The input source type is `CURSOR`, in which case there is no input file.

- Run the Design Advisor to determine the best database partition for each table. For more information, see "The Design Advisor" in *Troubleshooting and Tuning Database Performance*.

Restrictions

The following restrictions apply when using the load utility to load data in a multi-partition database:

- The location of the input files to the load operation cannot be a tape device.
- The **ROWCOUNT** parameter is not supported unless the ANALYZE mode is being used.
- If the target table has an identity column that is needed for distributing and the **identityoverride** file type modifier is not specified, or if you are using multiple database partitions to distribute and then load the data, the use of a **SAVECOUNT** greater than 0 on the **LOAD** command is not supported.
- If an identity column forms part of the distribution key or it is a random distribution table using the random by generation method, only the PARTITION_AND_LOAD mode is supported.
- The LOAD_ONLY and LOAD_ONLY_VERIFY_PART modes cannot be used with the **CLIENT** parameter of the **LOAD** command.
- The LOAD_ONLY_VERIFY_PART mode cannot be used with the CURSOR input source type.
- The distribution error isolation modes LOAD_ERRS_ONLY and SETUP_AND_LOAD_ERRS cannot be used with the **ALLOW READ ACCESS** and **COPY YES** parameters of the **LOAD** command.
- Multiple load operations can load data into the same table concurrently if the database partitions specified by the**OUTPUT_DBPARTNUMS** and **PARTITIONING_DBPARTNUMS** options do not overlap. For example, if a table is defined on database partitions 0 through 3, one load operation can load data into database partitions 0 and 1 while a second load operation can load data into database partitions 2 and 3. If the database partitions specified by the **PARTITIONING_DBPARTNUMS** options do overlap, then load will automatically choose a **PARTITIONING_DBPARTNUMS** parameter where no load partitioning subagent is already executing on the table, or fail if none are available.

  Starting with Version 9.7 Fix Pack 6, if the database partitions specified by the **PARTITIONING_DBPARTNUMS** options do overlap, the load utility automatically tries to pick up a **PARTITIONING_DBPARTNUMS** parameter from the database partitions indicated by **OUTPUT_DBPARTNUMS** where no load partitioning subagent is already executing on the table, or fail if none are available.

  It is strongly recommended that if you are going to explicitly specify partitions with the **PARTITIONING_DBPARTNUMS** option, you should use that option with all concurrent **LOAD** commands, with each command specifying different partitions. If you only specify **PARTITIONING_DBPARTNUMS** on some of the concurrent load commands or if you specify overlapping partitions, the **LOAD** command will need to pick alternate partitioning nodes for at least some of the concurrent loads, and in rare cases the command might fail (SQL2038N).
- Only non-delimited ASCII (ASC) and Delimited ASCII (DEL) files can be distributed across tables spanning multiple database partitions. PC/IXF files cannot be distributed, however, you can load a PC/IXF file into a table that is distributed over multiple database partitions by using the load operation in the LOAD_ONLY_VERIFY_PART mode.

Example

The following examples illustrate how to use the **LOAD** command to initiate various types of load operations. The database used in the following examples has five database partitions: 0, 1, 2, 3 and 4. Each database partition has a local directory `/db2/data/`. Two tables, TABLE1 and TABLE2, are defined on database partitions 0, 1, 3 and 4. When loading from a client, the user has access to a remote client that is not one of the database partitions.

Distribute and load example

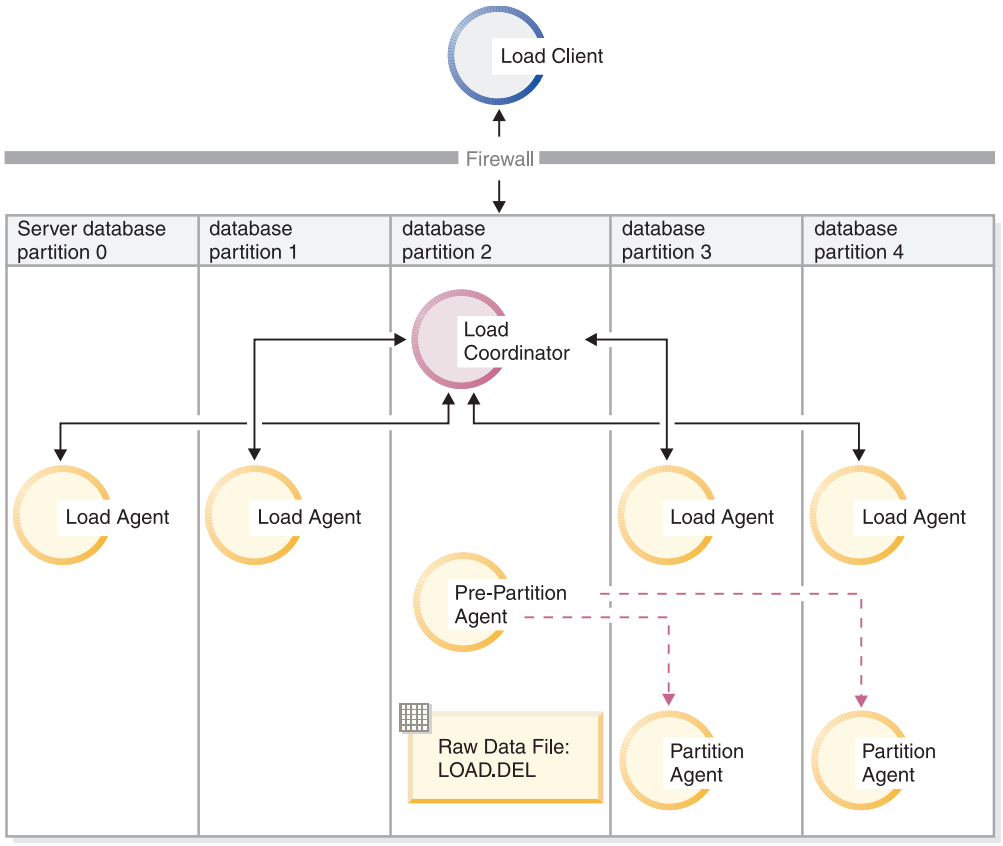
In this scenario, you are connected to a database partition that might or might not be a database partition where TABLE1 is defined. The data file `load.del` resides in the current working directory of this database partition. To load the data from `load.del` into all of the database partitions where TABLE1 is defined, issue the following command:

```
LOAD FROM LOAD.DEL of DEL REPLACE INTO TABLE1
```

Note: In this example, default values are used for all of the configuration parameters for partitioned database environments: The **MODE** parameter defaults to `PARTITION_AND_LOAD`. The **OUTPUT_DBPARTNUMS** parameter defaults to all database partitions on which TABLE1 is defined. The **PARTITIONING_DBPARTNUMS** defaults to the set of database partitions selected according to the **LOAD** command rules for choosing database partitions when none are specified.

To perform a load operation where data is distributed over database partitions 3 and 4, issue the following command:

```
LOAD FROM LOAD.DEL of DEL REPLACE INTO TABLE1
 PARTITIONED DB CONFIG PARTITIONING_DBPARTNUMS (3,4)
```



(artname: 00007956.gif)
Figure 14. Loading data into database partitions 3 and 4.. This diagram illustrates the behavior resulting when the previous command is issued. Data is loaded into database partitions 3 and 4.

Distribute only example

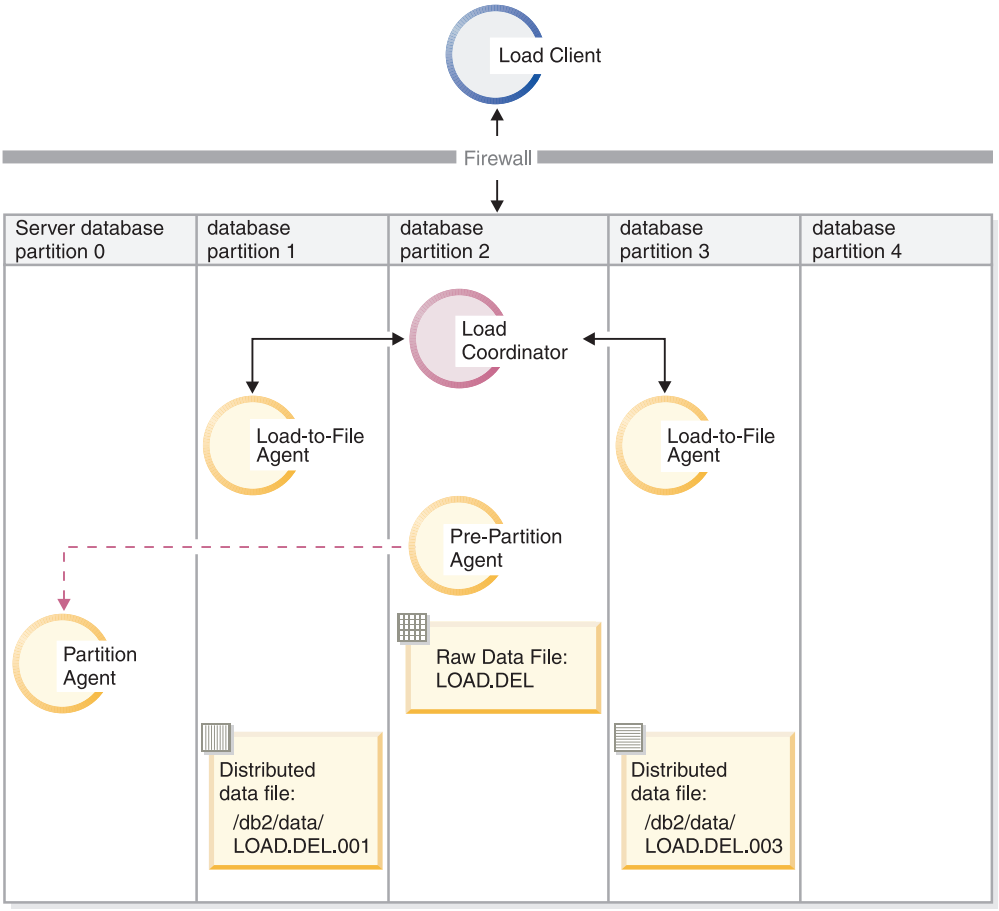
In this scenario, you are connected to a database partition that might or might not be a database partition where TABLE1 is defined. The data file `load.del` resides in the current working directory of this database partition. To distribute (but not load) `load.del` to all the database partitions on which TABLE1 is defined, using database partitions 3 and 4 issue the following command:

```
LOAD FROM LOAD.DEL of DEL REPLACE INTO TABLE1
 PARTITIONED DB CONFIG MODE PARTITION_ONLY
 PART_FILE_LOCATION /db2/data
 PARTITIONING_DBPARTNUMS (3,4)
```

This results in a file `load.del.xxx` being stored in the `/db2/data` directory on each database partition, where *xxx* is a three-digit representation of the database partition number.

To distribute the `load.del` file to database partitions 1 and 3, using only one partitioning agent running on database partition 0 (which is the default for **PARTITIONING_DBPARTNUMS**), issue the following command:

```
LOAD FROM LOAD.DEL OF DEL REPLACE INTO TABLE1
 PARTITIONED DB CONFIG MODE PARTITION_ONLY
 PART_FILE_LOCATION /db2/data
 OUTPUT_DBPARTNUMS (1,3)
```

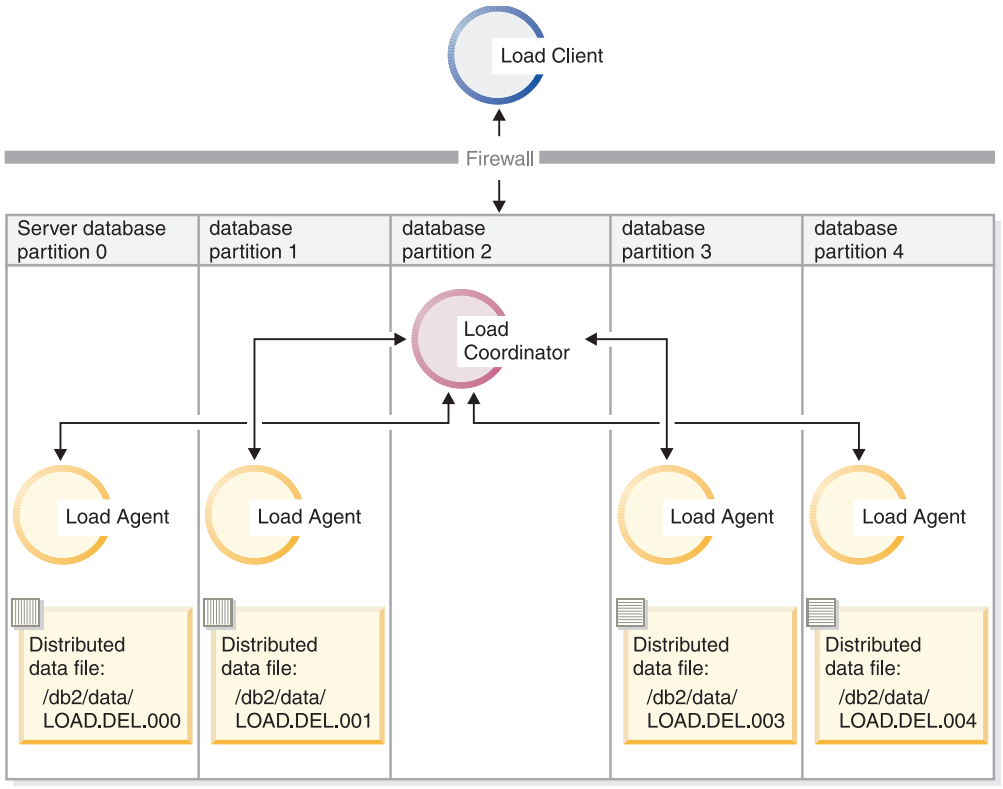


(artname: 00007957.gif)
Figure 15. Loading data into database partitions 1 and 3 using one partitioning agent.. This diagram illustrates the behavior that results when the previous command is issued. Data is loaded into database partitions 1 and 3, using one partitioning agent running on database partition 0.

Load only example

If you have already performed a load operation in the `PARTITION_ONLY` mode and want to load the partitioned files in the `/db2/data` directory of each loading database partition to all the database partitions on which TABLE1 is defined, issue the following command:

```
LOAD FROM LOAD.DEL OF DEL REPLACE INTO TABLE1
 PARTITIONED DB CONFIG MODE LOAD_ONLY
 PART_FILE_LOCATION /db2/data
```



(artname: 00007958.gif)
Figure 16. Loading data into all database partitions where a specific table is defined.. This diagram illustrates the behavior resulting when the previous command is issued. Distributed data is loaded to all database partitions where TABLE1 is defined.

To load into database partition 4 only, issue the following command:

```
LOAD FROM LOAD.DEL OF DEL REPLACE INTO TABLE1
 PARTITIONED DB CONFIG MODE LOAD_ONLY
 PART_FILE_LOCATION /db2/data
 OUTPUT_DBPARTNUMS (4)
```

Loading pre-distributed files without distribution map headers

The **LOAD** command can be used to load data files without distribution headers directly into several database partitions. If the data files exist in the `/db2/data` directory on each database partition where TABLE1 is defined and have the name `load.del.xxx`, where *xxx* is the database partition number, the files can be loaded by issuing the following command:

```
LOAD FROM LOAD.DEL OF DEL modified by dumpfile=rejected.rows
 REPLACE INTO TABLE1
 PARTITIONED DB CONFIG MODE LOAD_ONLY_VERIFY_PART
 PART_FILE_LOCATION /db2/data
```

To load the data into database partition 1 only, issue the following command:

```
LOAD FROM LOAD.DEL OF DEL modified by dumpfile=rejected.rows
 REPLACE INTO TABLE1
 PARTITIONED DB CONFIG MODE LOAD_ONLY_VERIFY_PART
 PART_FILE_LOCATION /db2/data
 OUTPUT_DBPARTNUMS (1)
```

Note: Rows that do not belong on the database partition from which they were loaded are rejected and put into the dump file, if one has been specified.

Loading from a remote client to a multi-partition database

To load data into a multi-partition database from a file that is on a remote client, you must specify the **CLIENT** parameter of the **LOAD** command. This parameter indicates that the data file is not on a server partition. For example:

```
LOAD CLIENT FROM LOAD.DEL OF DEL REPLACE INTO TABLE1
```

Note: You cannot use the LOAD_ONLY or LOAD_ONLY_VERIFY_PART modes with the **CLIENT** parameter.

Loading from a cursor

As in a single-partition database, you can load from a cursor into a multi-partition database. In this example, for the PARTITION_ONLY and LOAD_ONLY modes, the **PART_FILE_LOCATION** parameter must specify a fully qualified file name. This name is the fully qualified base file name of the distributed files that are created or loaded on each output database partition. Multiple files can be created with the specified base name if there are LOB columns in the target table.

To distribute all the rows in the answer set of the statement SELECT * FROM TABLE1 to a file on each database partition named /db2/data/select.out.*xxx* (where *xxx* is the database partition number), for future loading into TABLE2, issue the following commands:

```
DECLARE C1 CURSOR FOR SELECT * FROM TABLE1

LOAD FROM C1 OF CURSOR REPLACE INTO TABLE2
 PARTITIONED DB CONFIG MODE PARTITION_ONLY
 PART_FILE_LOCATION /db2/data/select.out
```

The data files produced by the previous operation can then be loaded by issuing the following **LOAD** command:

```
LOAD FROM C1 OF CURSOR REPLACE INTO TABLE2
 PARTITIONED CB CONFIG MODE LOAD_ONLY
 PART_FILE_LOCATION /db2/data/select.out
```

Loading data in a partitioned database environment-hints and tips: The following is some information to consider before loading a table in a multi-partition database:

- Familiarize yourself with the load configuration options by using the utility with small amounts of data.
- If the input data is already sorted, or in some chosen order, and you want to maintain that order during the loading process, only one database partition should be used for distributing. Parallel distribution cannot guarantee that the data is loaded in the same order it was received. The load utility chooses a single partitioning agent by default if the anyorder modifier is not specified on the **LOAD** command.
- If large objects (LOBs) are being loaded from separate files (that is, if you are using the lobsinfile modifier through the load utility), all directories containing the LOB files must be read-accessible to all the database partitions where loading is taking place. The LOAD *lob-path* parameter must be fully qualified when working with LOBs.
- You can force a job running in a multi-partition database to continue even if the load operation detects (at startup time) that some loading database partitions or

associated table spaces or tables are offline, by setting the ISOLATE_PART_ERRS option to SETUP_ERRS_ONLY or SETUP_AND_LOAD_ERRS.

- Use the STATUS_INTERVAL load configuration option to monitor the progress of a job running in a multi-partition database. The load operation produces messages at specified intervals indicating how many megabytes of data have been read by the pre-partitioning agent. These messages are dumped to the pre-partitioning agent message file. To view the contents of this file during the load operation, connect to the coordinator partition and issue a **LOAD QUERY** command against the target table.
- Better performance can be expected if the database partitions participating in the distribution process (as defined by the PARTITIONING_DBPARTNUMS option) are different from the loading database partitions (as defined by the OUTPUT_DBPARTNUMS option), since there is less contention for CPU cycles. When loading data into a multi-partition database, invoke the load utility on a database partition that is not participating in either the distributing or the loading operation.
- Specifying the MESSAGES parameter in the **LOAD** command saves the messages files from the pre-partitioning, partitioning, and load agents for reference at the end of the load operation. To view the contents of these files during a load operation, connect to the appropriate database partition and issue a **LOAD QUERY** command against the target table.
- The load utility chooses only one output database partition on which to collect statistics. The RUN_STAT_DBPARTNUM database configuration option can be used to specify the database partition.
- Before loading data in a multi-partition database, run the Design Advisor to determine the best partition for each table. For more information, see "The Design Advisor" in *Troubleshooting and Tuning Database Performance*.

Troubleshooting

If the load utility is hanging, you can:

- Use the STATUS_INTERVAL parameter to monitor the progress of a multi-partition database load operation. The status interval information is dumped to the pre-partitioning agent message file on the coordinator partition.
- Check the partitioning agent messages file to see the status of the partitioning agent processes on each database partition. If the load is proceeding with no errors, and the TRACE option has been set, there should be trace messages for a number of records in these message files.
- Check the load messages file to see if there are any load error messages.

  Note: You must specify the MESSAGES option of the **LOAD** command in order for these files to exist.
- Interrupt the current load operation if you find errors suggesting that one of the load processes encountered errors.

Monitoring a load operation in a partitioned database environment using the LOAD QUERY command

During a load operation in a partitioned database environment, message files are created by some of the load processes on the database partitions where they are being executed.

The message files store all information, warning, and error messages produced during the execution of the load operation. The load processes that produce

message files that can be viewed by the user are the load agent, pre-partitioning agent, and partitioning agent. The content of the message file is only available after the load operation is finished.

You can connect to individual database partitions during a load operation and issue the **LOAD QUERY** command against the target table. When issued from the CLP, this command displays the contents of the load message files that currently reside on that database partition for the table that is specified in the **LOAD QUERY** command.

For example, table TABLE1 is defined on database partitions 0 through 3 in database WSDB. You are connected to database partition 0 and issue the following **LOAD** command:

```
load from load.del of del replace into table1 partitioned db config
    partitioning_dbpartnums (1)
```

This command initiates a load operation that includes load agents running on database partitions 0, 1, 2, and 3; a partitioning agent running on database partition 1; and a pre-partitioning agent running on database partition 0.

Database partition 0 contains one message file for the pre-partitioning agent and one for the load agent on that database partition. To view the contents of these files at the same time, start a new session and issue the following commands from the CLP:

```
set client connect_node 0
connect to wsdb
load query table table1
```

Database partition 1 contains one file for the load agent and one for the partitioning agent. To view the contents of these files, start a new session and issue the following commands from the CLP:

```
set client connect_node 1
connect to wsdb
load query table table1
```

Note: The messages generated by the STATUS_INTERVAL load configuration option appear in the pre-partitioning agent message file. To view these message during a load operation, you must connect to the coordinator partition and issue the **LOAD QUERY** command.

Saving the contents of message files

If a load operation is initiated through the **db2Load** API, the messages option (piLocalMsgFileName) must be specified and the message files are brought from the server to the client and stored for you to view.

For multi-partition database load operations initiated from the CLP, the message files are not displayed to the console or retained. To save or view the contents of these files after a multi-partition database load is complete, the MESSAGES option of the **LOAD** command must be specified. If this option is used, once the load operation is complete the message files on each database partition are transferred to the client machine and stored in files using the base name indicated by the MESSAGES option. For multi-partition database load operations, the name of the file corresponding to the load process that produced it is listed in the following table:

Process type	File name
Load Agent	<*message-file-name*>.LOAD.<*dbpartition-number*>
Partitioning Agent	<*message-file-name*>.PART.<*dbpartition-number*>
Pre-partitioning Agent	<*message-file-name*>.PREP.<*dbpartition-number*>

For example, if the MESSAGES option specifies /wsdb/messages/load, the load agent message file for database partition 2 is /wsdb/messages/load.LOAD.002.

Note: It is strongly recommended that the MESSAGES option be used for multi-partition database load operations initiated from the CLP.

Resuming, restarting, or terminating load operations in a partitioned database environment

The steps you need to take following failed load operations in a partitioned database environment depend on when the failure occurred.

The load process in a multi-partition database consists of two stages:

1. The *setup stage*, during which database partition-level resources such as table locks on output database partitions are acquired

   In general, if a failure occurs during the setup stage, restart and terminate operations are not necessary. What you need to do depends on the error isolation mode that was specified for the failed load operation.

   If the load operation specified that setup stage errors were not to be isolated, the entire load operation is canceled and the state of the table on each database partition is rolled back to the state it was in before the load operation.

   If the load operation specified that setup stage errors were to be isolated, the load operation continues on the database partitions where the setup stage was successful, but the table on each of the failing database partitions is rolled back to the state it was in before the load operation. This means that a single load operation can fail at different stages if some partitions fail during the setup stage and others fail during the load stage
2. The *load stage*, during which data is formatted and loaded into tables on the database partitions

   If a load operation fails on at least one database partition during the load stage of a multi-partition database load operation, a **LOAD RESTART** or **LOAD TERMINATE** command must be issued. This is necessary because loading data in a multi-partition database is done through a single transaction.

   If you can fix the problems that caused the failed load to occur, choose a **LOAD RESTART**. This saves time because if a load restart operation is initiated, the load operation continues from where it left off on all database partitions.

   If you want the table returned to the state it was in before the initial load operation, choose a **LOAD TERMINATE**.

Determining when a load failed

The first thing you need to do if your load operation in a partitioned environment fails is to determine on which partitions it failed and at what stage each of them failed. This is done by looking at the partition summary. If the **LOAD** command was issued from the CLP, the partition summary is displayed at the end of the load (see

following example). If the **LOAD** command was issued from the db2Load API, the partition summary is contained in the **poAgentInfoList** field of the db2PartLoadOut structure.

If there is an entry of "LOAD" for "Agent Type", for a given partition, then that partition reached the load stage, otherwise a failure occurred during the setup stage. A negative SQL Code indicates that it failed. In the following example, the load failed on partition 1 during the load stage.

```
Agent Type     Node     SQL Code     Result
_____________________________________________________________
LOAD           000      +00000000    Success.
_____________________________________________________________
LOAD           001      -00000289    Error. May require RESTART.
_____________________________________________________________
LOAD           002      +00000000    Success.
_____________________________________________________________
LOAD           003      +00000000    Success.
.
.
.
```

Resuming, restarting, or terminating a failed load

Only loads with the ISOLATE_PART_ERRS option specifying SETUP_ERRS_ONLY or SETUP_AND_LOAD_ERRS should fail during the setup stage. For loads that fail on at least one output database partition fail during this stage, you can issue a **LOAD REPLACE** or **LOAD INSERT** command. Use the OUTPUT_DBPARTNUMS option to specify only those database partitions on which it failed.

For loads that fail on at least one output database partition during the load stage, issue a **LOAD RESTART** or **LOAD TERMINATE** command.

For loads that fail on at least one output database partition during the setup stage and at least one output database partition during the load stage, you need to perform two load operations to resume the failed load-one for the setup stage failures and one for the load stage failures, as previously described. To effectively undo this type of failed load operation, issue a **LOAD TERMINATE** command. However, after issuing the command, you must account for all partitions because no changes were made to the table on the partitions that failed during the setup stage, and all the changes are undone for the partitions that failed during the load stage.

For example, TABLE1 is defined on database partitions 0 through 3 in database WSDB. The following command is issued:

```
load from load.del of del insert into table1 partitioned db config
isolate_part_errs setup_and_load_errs
```

There is a failure on output database partition 1 during the setup stage. Since setup stage errors are isolated, the load operation continues, but there is a failure on partition 3 during the load stage. To resume the load operation, you would issue the following commands:

```
load from load.del of del replace into table1 partitioned db config
output_dbpartnums (1)

load from load.del of del restart into table1 partitioned db config
isolate_part_errs setup_and_load_errs
```

Note: For load restart operations, the options specified in the **LOAD RESTART** command are honored, so it is important that they are identical to the ones specified in the original **LOAD** command.

Migration and version compatibility

The **DB2_PARTITIONEDLOAD_DEFAULT** registry variable can be used to revert to pre-Db2 Universal Database Version 8 load behavior in a multi-partition database.

Note: The **DB2_PARTITIONEDLOAD_DEFAULT** registry variable is deprecated and might be removed in a later release.

Reverting to the pre-Db2 Version 8 behavior of the **LOAD** command in a multi-partition database, allows you to load a file with a valid distribution header into a single database partition without specifying any extra partitioned database configuration options. You can do this by setting the value of **DB2_PARTITIONEDLOAD_DEFAULT** to NO. You might choose to use this option if you want to avoid modifying existing scripts that issue the **LOAD** command against single database partitions. For example, to load a distribution file into database partition 3 of a table that resides in a database partition group with four database partitions, issue the following command:

```
db2set DB2_PARTITIONEDLOAD_DEFAULT=NO
```

Then issue the following commands from the Db2 Command Line Processor:

```
CONNECT RESET

SET CLIENT CONNECT_NODE 3

CONNECT TO DB MYDB

LOAD FROM LOAD.DEL OF DEL REPLACE INTO TABLE1
```

In a multi-partition database, when no multi-partition database load configuration options are specified, the load operation takes place on all the database partitions on which the table is defined. The input file does not require a distribution header, and the MODE option defaults to PARTITION_AND_LOAD. To load a single database partition, the OUTPUT_DBPARTNUMS option must be specified.

Reference - Load in a partitioned environment

Load sessions in a partitioned database environment - CLP examples:

The following examples demonstrate loading data in a multi-partition database.

The database has four database partitions numbered 0 through 3. Database WSDB is defined on all of the database partitions, and table TABLE1 resides in the default database partition group which is also defined on all of the database partitions.

Example 1

To load data into TABLE1 from the user data file load.del which resides on database partition 0, connect to database partition 0 and then issue the following command:

```
load from load.del of del replace into table1
```

If the load operation is successful, the output will be as follows:

```
Agent Type     Node     SQL Code     Result
_____________________________________________________________
LOAD           000      +00000000    Success.
_____________________________________________________________
LOAD           001      +00000000    Success.
_____________________________________________________________
LOAD           002      +00000000    Success.
_____________________________________________________________
LOAD           003      +00000000    Success.
_____________________________________________________________
PARTITION      001      +00000000    Success.
_____________________________________________________________
PRE_PARTITION  000      +00000000    Success.
_____________________________________________________________
RESULTS:       4 of 4 LOADs completed successfully.
_____________________________________________________________

Summary of Partitioning Agents:
Rows Read                   = 100000
Rows Rejected               = 0
Rows Partitioned            = 100000

Summary of LOAD Agents:
Number of rows read         = 100000
Number of rows skipped      = 0
Number of rows loaded       = 100000
Number of rows rejected     = 0
Number of rows deleted      = 0
Number of rows committed    = 100000
```

The output indicates that there was one load agent on each database partition and each ran successfully. It also shows that there was one pre-partitioning agent running on the coordinator partition and one partitioning agent running on database partition 1. These processes completed successfully with a normal SQL return code of 0. The statistical summary shows that the pre-partitioning agent read 100,000 rows, the partitioning agent distributed 100,000 rows, and the sum of all rows loaded by the load agents is 100,000.

Example 2

In the following example, data is loaded into TABLE1 in the `PARTITION_ONLY` mode. The distributed output files is stored on each of the output database partitions in the directory `/db/data`:

```
load from load.del of del replace into table1 partitioned db config mode
   partition_only part_file_location /db/data
```

The output from the load command is as follows:

```
Agent Type     Node     SQL Code     Result
_____________________________________________________________
LOAD_TO_FILE   000      +00000000    Success.
_____________________________________________________________
LOAD_TO_FILE   001      +00000000    Success.
_____________________________________________________________
LOAD_TO_FILE   002      +00000000    Success.
_____________________________________________________________
LOAD_TO_FILE   003      +00000000    Success.
_____________________________________________________________
PARTITION      001      +00000000    Success.
_____________________________________________________________
PRE_PARTITION  000      +00000000    Success.
_____________________________________________________________
```

```
Summary of Partitioning Agents:
Rows Read                  = 100000
Rows Rejected              = 0
Rows Partitioned           = 100000
```

The output indicates that there was a load-to-file agent running on each output database partition, and these agents ran successfully. There was a pre-partitioning agent on the coordinator partition, and a partitioning agent running on database partition 1. The statistical summary indicates that 100,000 rows were successfully read by the pre-partitioning agent and 100,000 rows were successfully distributed by the partitioning agent. Since no rows were loaded into the table, no summary of the number of rows loaded appears.

Example 3

To load the files that were generated during the PARTITION_ONLY load operation shown previously, issue the following command:

```
load from load.del of del replace into table1 partitioned db config mode
    load_only part_file_location /db/data
```

The output from the load command will be as follows:

```
Agent Type     Node     SQL Code     Result
_____________________________________________________
LOAD           000      +00000000    Success.
_____________________________________________________
LOAD           001      +00000000    Success.
_____________________________________________________
LOAD           002      +00000000    Success.
_____________________________________________________
LOAD           003      +00000000    Success.
_____________________________________________________
RESULTS:       4 of 4 LOADs completed successfully.
_____________________________________________________

Summary of LOAD Agents:
Number of rows read         = 100000
Number of rows skipped      = 0
Number of rows loaded       = 100000
Number of rows rejected     = 0
Number of rows deleted      = 0
Number of rows committed    = 100000
```

The output indicates that the load agents on each output database partition ran successfully and that the sum of the number of rows loaded by all load agents is 100,000. No summary of rows distributed is indicated since distribution was not performed.

Example 4

If the following LOAD command is issued:

```
load from load.del of del replace into table1
```

and one of the loading database partitions runs out of space in the table space during the load operation, the following output might be returned:

```
SQL0289N  Unable to allocate new pages in table space "DMS4KT".
SQLSTATE=57011

Agent Type     Node     SQL Code     Result
_____________________________________________________
LOAD           000      +00000000    Success.
```

```
_____________________________________________________________________
LOAD           001      -00000289    Error. May require RESTART.
_____________________________________________________________________
LOAD           002      +00000000    Success.
_____________________________________________________________________
LOAD           003      +00000000    Success.
_____________________________________________________________________
PARTITION      001      +00000000    Success.
_____________________________________________________________________
PRE_PARTITION  000      +00000000    Success.
_____________________________________________________________________
RESULTS:       3 of 4 LOADs completed successfully.
_____________________________________________________________________

Summary of Partitioning Agents:
Rows Read                    = 0
Rows Rejected                = 0
Rows Partitioned             = 0

Summary of LOAD Agents:
Number of rows read          = 0
Number of rows skipped       = 0
Number of rows loaded        = 0
Number of rows rejected      = 0
Number of rows deleted       = 0
Number of rows committed     = 0
```

The output indicates that the load operation returned error SQL0289. The database partition summary indicates that database partition 1 ran out of space. If additional space is added to the containers of the table space on database partition 1, the load operation can be restarted as follows:

```
load from load.del of del restart into table1
```

Load configuration options for partitioned database environments:

There are a number of configuration options that you can use to modify a load operation in a partitioned database environment.

MODE *X*

Specifies the mode in which the load operation occurs when loading a multi-partition database. PARTITION_AND_LOAD is the default. Valid values are:

- PARTITION_AND_LOAD. Data is distributed (perhaps in parallel) and loaded simultaneously on the corresponding database partitions.
- PARTITION_ONLY. Data is distributed (perhaps in parallel) and the output is written to files in a specified location on each loading database partition. For file types other than CURSOR, the format of the output file name on each database partition is *filename.xxx*, where *filename* is the input file name specified in the **LOAD** command and *xxx* is the 3-digit database partition number. For the CURSOR file type, the name of the output file on each database partition is determined by the PART_FILE_LOCATION option. See the PART_FILE_LOCATION option for details on how to specify the location of the distribution file for each database partition.

  Note:

  1. This mode cannot be used for a CLI load operation.
  2. If the table contains an identity column that is needed for distribution, then this mode is not supported, unless the **identityoverride** file type modifier is specified.

3. This mode cannot be used for random distribution tables that use the random by generation method.
4. Distribution files generated for file type CURSOR are not compatible between Db2 releases. This means that distribution files of file type CURSOR that were generated in a previous release cannot be loaded using the LOAD_ONLY mode. Similarly, distribution files of file type CURSOR that were generated in the current release cannot be loaded in a future release using the LOAD_ONLY mode.

- LOAD_ONLY. Data is assumed to be already distributed; the distribution process is skipped, and the data is loaded simultaneously on the corresponding database partitions. For file types other than CURSOR, the format of the input file name for each database partition should be *filename.xxx*, where *filename* is the name of the file specified in the **LOAD** command and *xxx* is the 3-digit database partition number. For the CURSOR file type, the name of the input file on each database partition is determined by the PART_FILE_LOCATION option. See the PART_FILE_LOCATION option for details on how to specify the location of the distribution file for each database partition.

  Note:

  1. This mode cannot be used for a CLI load operation, or when the **CLIENT** parameter of **LOAD** command is specified.
  2. If the table contains an identity column that is needed for distribution, then this mode is not supported, unless the **identityoverride** file type modifier is specified.
  3. This mode cannot be used for random distribution tables that use the random by generation method.
- LOAD_ONLY_VERIFY_PART. Data is assumed to be already distributed, but the data file does not contain a partition header. The distributing process is skipped, and the data is loaded simultaneously on the corresponding database partitions. During the load operation, each row is checked to verify that it is on the correct database partition. Rows containing database partition violations are placed in a dump file if the **dumpfile** file type modifier is specified. Otherwise, the rows are discarded. If database partition violations exist on a particular loading database partition, a single warning is written to the load message file for that database partition. The format of the input file name for each database partition should be *filename.xxx*, where *filename* is the name of the file specified in the **LOAD** command and *xxx* is the 3-digit database partition number. See the PART_FILE_LOCATION option for details on how to specify the location of the distribution file for each database partition.

  Note:

  1. This mode cannot be used for a CLI load operation, or when the **CLIENT** parameter of **LOAD** command is specified.
  2. If the table contains an identity column that is needed for distribution, then this mode is not supported, unless the **identityoverride** file type modifier is specified.
  3. This mode cannot be used for random distribution tables that use the random by generation method.
- ANALYZE. An optimal distribution map with even distribution across all database partitions is generated.

PART_FILE_LOCATION *X*

In the PARTITION_ONLY, LOAD_ONLY, and LOAD_ONLY_VERIFY_PART modes, this parameter can be used to specify the location of the distributed files. This location must exist on each database partition specified by the OUTPUT_DBPARTNUMS option. If the location specified is a relative path name, the path is appended to the current directory to create the location for the distributed files.

For the CURSOR file type, this option must be specified, and the location must refer to a fully qualified file name. This name is the fully qualified base file name of the distributed files that are created on each output database partition in the PARTITION_ONLY mode, or the location of the files to be read from for each database partition in the LOAD_ONLY mode. When using the PARTITION_ONLY mode, multiple files can be created with the specified base name if the target table contains LOB columns.

For file types other than CURSOR, if this option is not specified, the current directory is used for the distributed files.

OUTPUT_DBPARTNUMS *X*

X represents a list of database partition numbers. The database partition numbers represent the database partitions on which the load operation is to be performed. Any data that does not partition to any of the database partitions listed will not be loaded. Unless we are loading a random distribution table that uses random by generation method. In that case all data will be loaded into the set of database partitions listed.

The database partition numbers must be a subset of the database partitions on which the table is defined, except for column-organized tables, in which case all database partitions must be specified (SQL27906N). All database partitions are selected by default. The list must be enclosed in parentheses and the items in the list must be separated by commas. Ranges are permitted (for example, (0, 2 to 10, 15)).

PARTITIONING_DBPARTNUMS *X*

X represents a list of database partition numbers that are used in the distribution process. The list must be enclosed in parentheses and the items in the list must be separated by commas. Ranges are permitted (for example, (0, 2 to 10, 15)). The database partitions specified for the distribution process can be different from the database partitions being loaded. If PARTITIONING_DBPARTNUMS is not specified, the load utility determines how many database partitions are needed and which database partitions to use in order to achieve optimal performance.

If the **anyorder** file type modifier is not specified in the **LOAD** command, only one partitioning agent is used in the load session. Furthermore, if there is only one database partition specified for the OUTPUT_DBPARTNUMS option, or the coordinator partition of the load operation is not an element of OUTPUT_DBPARTNUMS, the coordinator partition of the load operation is used in the distribution process. Otherwise, the first database partition (not the coordinator partition) in OUTPUT_DBPARTNUMS is used in the distribution process.

If the **anyorder** file type modifier is specified, the number of database partitions used in the distribution process is determined as follows: (number of partitions in OUTPUT_DBPARTNUMS/4 + 1).

This option is ignored when loading random distribution tables using the random by generation method. That distribution method does not use partitioning agents.

MAX_NUM_PART_AGENTS *X*

Specifies the maximum numbers of partitioning agents to be used in a load session. The default is 25. This option has no affect when loading into a random distribution table using random by generation method. That distribution method does not use partitioning agents.

ISOLATE_PART_ERRS *X*

Indicates how the load operation reacts to errors that occur on individual database partitions. The default is LOAD_ERRS_ONLY, unless both the **ALLOW READ ACCESS** and **COPY YES** parameters of the **LOAD** command are specified, in which case the default is NO_ISOLATION. Valid values are:

- SETUP_ERRS_ONLY. Errors that occur on a database partition during setup, such as problems accessing a database partition, or problems accessing a table space or table on a database partition, cause the load operation to stop on the failing database partitions but to continue on the remaining database partitions. Errors that occur on a database partition while data is being loaded cause the entire operation to fail.
- LOAD_ERRS_ONLY. Errors that occur on a database partition during setup cause the entire load operation to fail. If an error occurs while data is being loaded, the load operation will stop on the database partition where the error occurred. The load operation continues on the remaining database partitions until a failure occurs or until all the data is loaded. The newly loaded data will not be visible until a load restart operation is performed and completes successfully.

  Note: This mode cannot be used when both the **ALLOW READ ACCESS** and the **COPY YES** parameters of the **LOAD** command are specified.
- SETUP_AND_LOAD_ERRS. In this mode, database partition-level errors during setup or loading data cause processing to stop only on the affected database partitions. As with the LOAD_ERRS_ONLY mode, when partition errors do occur while data is loaded, newly loaded data will not be visible until a load restart operation is performed and completes successfully.

  Note: This mode cannot be used when both the **ALLOW READ ACCESS** and the **COPY YES** options of the **LOAD** command are specified.
- NO_ISOLATION. Any error during the load operation causes the load operation to fail.

STATUS_INTERVAL *X*

X represents how often you are notified of the volume of data that has been read. The unit of measurement is megabytes (MB). The default is 100 MB. Valid values are whole numbers from 1 to 4000.

PORT_RANGE *X*

X represents the range of TCP ports used to create sockets for internal communications. The default range is from 49152 to 65535. If defined at the time of invocation, the value of the **DB2ATLD_PORTS** registry variable replaces the value of the PORT_RANGE load configuration option. For the **DB2ATLD_PORTS** registry variable, the range should be provided in the following format:

```
<lower-port-number:higher-port-number>
```

From the CLP, the format is:

```
( lower-port-number, higher-port-number )
```

CHECK_TRUNCATION
Specifies that the program should check for truncation of data records at input/output. The default behavior is that data is not checked for truncation at input/output.

MAP_FILE_INPUT *X*
X specifies the input file name for the distribution map. This parameter must be specified if the distribution map is customized, as it points to the file containing the customized distribution map. A customized distribution map can be created by using the **db2gpmap** program to extract the map from the database system catalog table, or by using the ANALYZE mode of the **LOAD** command to generate an optimal map. The map generated by using the ANALYZE mode must be moved to each database partition in your database before the load operation can proceed.

MAP_FILE_OUTPUT *X*
X represents the output filename for the distribution map. The output file is created on the database partition issuing the **LOAD** command assuming that database partition is participating in the database partition group where partitioning is performed. If the **LOAD** command is invoked on a database partition that is not participating in partitioning (as defined by **PARTITIONING_DBPARTNUMS**), the output file is created at the first database partition defined with the **PARTITIONING_DBPARTNUMS** parameter. Consider the following partitioned database environment setup:

```
1 serv1 0
2 serv1 1
3 serv2 0
4 serv2 1
5 serv3 0
```

Running the following **LOAD** command on serv3, creates the distribution map on serv1.

```
LOAD FROM file OF ASC METHOD L ( ...) INSERT INTO table CONFIG
MODE ANALYZE PARTITIONING_DBPARTNUMS(1,2,3,4)
MAP_FILE_OUTPUT '/home/db2user/distribution.map'
```

This parameter should be used when the ANALYZE mode is specified. An optimal distribution map with even distribution across all database partitions is generated. If this parameter is not specified and the ANALYZE mode is specified, the program exits with an error.

TRACE *X*
Specifies the number of records to trace when you require a review of a dump of the data conversion process and the output of the hashing values. The default is 0.

NEWLINE
Used when the input data file is an ASC file with each record delimited by a new line character and the **reclen** file type modifier is specified in the **LOAD** command. When this option is specified, each record is checked for a new line character. The record length, as specified in the **reclen** file type modifier, is also checked.

DISTFILE *X*
If this option is specified, the load utility generates a database partition distribution file with the given name. The database partition distribution file contains 32 768 integers: one for each entry in the distribution map for the target table. Each integer in the file represents the number of rows in the input files being loaded that hashed to the corresponding distribution map entry.

This information can help you identify skew in your data and also help you decide whether a new distribution map should be generated for the table using the ANALYZE mode of the utility. If this option is not specified, the default behavior of the load utility is to not generate the distribution file.

Note: When this option is specified, a maximum of one partitioning agent is used for the load operation. Even if you explicitly request multiple partitioning agents, only one is used.

OMIT_HEADER
Specifies that a distribution map header should not be included in the distribution file. If not specified, a header is generated.

RUN_STAT_DBPARTNUM *X*
If the **STATISTICS USE PROFILE** parameter is specified in the **LOAD** command, statistics are collected only on one database partition. This parameter specifies on which database partition to collect statistics. If the value is -1 or not specified at all, statistics are collected on the first database partition in the output database partition list.

Ingest utility

The ingest utility (sometimes referred to as continuous data ingest, or CDI) is a high-speed client-side Db2 utility that streams data from files and pipes into Db2 target tables. Because the ingest utility can move large amounts of real-time data without locking the target table, you do not need to choose between the data currency and availability.

The ingest utility ingests pre-processed data directly or from files output by ETL tools or other means. It can run continually and thus it can process a continuous data stream through pipes. The data is ingested at speeds that are high enough to populate even large databases in partitioned database environments.

An **INGEST** command updates the target table with low latency in a single step. The ingest utility uses row locking, so it has minimal interference with other user activities on the same table.

With this utility, you can perform DML operations on a table using a SQL-like interface without locking the target table. These ingest operations support the following SQL statements: INSERT, UPDATE, MERGE, REPLACE, and DELETE. The ingest utility also supports the use of SQL expressions to build individual column values from more than one data field.

Other important features of the ingest utility include:

- **Commit by time or number of rows.** You can use the **commit_count** ingest configuration parameter to have commit frequency determined by the number of written rows or use the default **commit_period** ingest configuration parameter to have commit frequency determined by a specified time.
- **Support for copying rejected records to a file or table, or discarding them.** You can specify what the **INGEST** command does with rows rejected by the ingest utility (using the **DUMPFILE** parameter) or by Db2 (using the **EXCEPTION TABLE** parameter).
- **Support for restart and recovery.** By default, all **INGEST** commands are restartable from the last commit point. In addition, the ingest utility attempts to recover from certain errors if you have set the **retry_count** ingest configuration parameter.

The **INGEST** command supports the following input data formats:

- Delimited text
- Positional text and binary
- Columns in various orders and formats

In addition to regular tables and nicknames, the **INGEST** command supports the following table types:

- multidimensional clustering (MDC) and insert time clustering (ITC) tables
- range-partitioned tables
- range-clustered tables (RCT)
- materialized query tables (MQTs) that are defined as MAINTAINED BY USER, including summary tables
- temporal tables
- updatable views (except typed views)

A single **INGEST** command goes through three major phases:

1. **Transport**
   The transporters read from the data source and put records on the formatter queues. For INSERT and MERGE operations, there is one transporter thread for each input source (for example, one thread for each input file). For UPDATE and DELETE operations, there is only one transporter thread.
2. **Format**
   The formatters parse each record, convert the data into the format that Db2 database systems require, and put each formatted record on one of the flusher queues for that record's partition. The number of formatter threads is specified by the **num_formatters** configuration parameter. The default is (number of logical CPUs)/2.
3. **Flush**
   The flushers issue the SQL statements to perform the operations on the Db2 tables. The number of flushers for each partition is specified by the **num_flushers_per_partition** configuration parameter. The default is max(1, ((number of logical CPUs)/2)/(number of partitions)).

Overview of ingest-related tasks

This section provides a high-level overview of the main setup and operational tasks related to using the ingest utility.

Setting up ingest middleware

1. Decide where to run the ingest utility
   You can run ingest jobs on an existing machine or from a stand-alone machine. For more information, see "Deciding where to run the ingest utility" on page 174
2. Install the ingest utility (part of the Db2 Data Server Runtime Client and the Db2 Data Server Client).
   If you decide to install the ingest utility on a new, stand-alone machine, run the install for the Db2 client image. For more information, see "Installing IBM data server clients (Linux, UNIX)" in *Installing IBM Data Server Clients*

Developing a process to populate a table

1. (If required) Address code page issues

   Depending on whether the same code page is used by the input data, the Db2 client, and the Db2 server, there may be some user actions to take before running an **INGEST** command. For more information, see "Code page considerations for the ingest utility" on page 189.
2. Set up to handle the restart of failed **INGEST** commands

   To make an ingest operation restartable, you need to create a restart log table before issuing the **INGEST** command. For more information, see "Creating the restart table" on page 175.
3. Write an **INGEST** command

   Issue the **INGEST** command along with the mandatory parameters, like the input source and data type, and various optional parameters. For a detailed description of the command syntax and usage, as well as examples, see "Ingesting data" on page 176 and "INGEST" in the *Command Reference*.
4. Set up to process an ongoing stream of ingest jobs

   If you want to easily call a pre-written **INGEST** command, create a script for the command and call it when necessary. For more information, see "Scenario: Processing a stream of files with the ingest utility" on page 193

Performing operational tasks

- (If required) Addressing a failed **INGEST** command

  If an ingest job fails, you have the option of restarting or terminating the command. For more information, see "Restarting a failed ingest operation" on page 183 or "Terminating a failed ingest operation" on page 185.
- Monitoring an **INGEST** command

  For more information, see "Monitoring ingest operations" on page 186.

(Optional) Optimizing performance

- Review tunable configuration parameters for the **INGEST** command.
- Modify your **INGEST** command to meet high performance requirements.

  For more information, see "Performance considerations for ingest operations" on page 189.

Deciding where to run the ingest utility

The ingest utility is included as a part of the Db2 client install. You can run it from either the client or the server.

About this task

There are two choices for where to run the ingest utility:

On an existing server in the data warehouse environment

There are two choices for where to run ingest jobs within this type of setup:

- On the Db2 coordinator partition (the database partition server to which applications will connect and on which the coordinating agent is located)
- On an existing ETL (extract, transform, and load) server

On a new server

There are two choices for where to run ingest jobs within this type of setup:

- On a server that is only running the ingest utility
- On a server that is also hosting an additional Db2 coordinator partition that is dedicated to the ingest utility.

There are a number of factors that can influence where you decide to install the ingest utility:

- Performance: Having the ingest utility installed on its own server has a significant performance benefit, so this would be suitable for environments with large data sets.
- Cost: Having the ingest utility installed on an existing server means that no additional expenses are incurred as a result of using it.
- Ease of administration

Creating the restart table

By default, failed **INGEST** commands are restartable from the last commit point; however you first need to create a restart table, which stores the information needed to resume an **INGEST** command.

About this task

You have to create the restart table only once, and that table will be used by all **INGEST** commands in the database.

The ingest utility will use this table to store information needed to resume an incomplete INGEST command from the last commit point.

Note: The restart table does not contain copies of the input rows, only some counters to indicate which rows have been committed.

Restrictions

- It is recommended that you place the restart table in the same tablespace as the target tables that the ingest utility updates. If this is not possible, you must ensure that the tablespace containing the restart table is at the same level as the tablespace containing the target table. For example, if you restore or roll forward one of the table spaces, you must restore or roll forward the other to the same level. If the table spaces are at different levels and you run an **INGEST** command with the **RESTART CONTINUE** option, the ingest utility could fail or ingest incorrect data.
- If your disaster recovery strategy includes replicating the target tables of ingest operations, you must also replicate the restart table so it is kept in sync with the target tables.

Procedure

To create the restart table:

- If you are using a Version 10.1 or Version 10.5 server, call the SYSPROC.SYSINSTALLOBJECTS stored procedure:

  ```
  db2 "CALL SYSPROC.SYSINSTALLOBJECTS('INGEST', 'C', tablespace-name, NULL)"
  ```

- If you are using a Version 9.5, Version 9.7, or Version 9.8 server, issue the following SQL statements:

  ```
  CREATE TABLE SYSTOOLS.INGESTRESTART (
    JOBID             VARCHAR(256) NOT NULL,
    APPLICATIONID     VARCHAR(256) NOT NULL,
    FLUSHERID         INT          NOT NULL,
  ```

```
    FLUSHERDISTID     INT          NOT NULL,
    TRANSPORTERID     INT          NOT NULL,
    BUFFERID          BIGINT       NOT NULL,
    BYTEPOS           BIGINT       NOT NULL,
    ROWSPROCESSED     INT          NOT NULL,
      PRIMARY KEY (JOBID, FLUSHERID, TRANSPORTERID, FLUSHERDISTID))
    IN <tablespace-name>
    DISTRIBUTE BY (FLUSHERDISTID);

  GRANT SELECT, INSERT, UPDATE, DELETE
     ON TABLE SYSTOOLS.INGESTRESTART TO PUBLIC;
```

Results

The restart table, SYSTOOLS.INGESTRESTART, should now be created in the specified table space, and you can now run restartable **INGEST** commands.

Example

A DBA intends to run all **INGEST** commands as restartable, so the DBA needs to first create a restart table:

1. The DBA connects to the database:
   ```
   db2 CONNECT TO sample
   ```
2. The DBA calls the stored procedure:
   ```
   db2 "CALL SYSPROC.SYSINSTALLOBJECTS('INGEST', 'C', NULL, NULL)"
   ```

What to do next

Ensure that any user who will modify the restart table has the appropriate authorization:

- If the **INGEST** command specifies `RESTART NEW`, the user must have SELECT, INSERT, UPDATE, and DELETE privilege on the restart table.
- If the **INGEST** command specifies `RESTART TERMINATE`, the user must have SELECT and DELETE privilege on the restart table.

Ingesting data

You can use the ingest utility to continuously pump data into Db2 tables using SQL array inserts, updates, and deletes until sources are exhausted.

Before you begin

Before invoking the ingest utility, you must be connected to the database into which the data will be imported.

By default, failed **INGEST** commands are restartable from the last commit point; however you must first create a restart table, otherwise you receive an error message notifying you that the command you issued is not restartable. The ingest utility uses this table to store information needed to resume an incomplete **INGEST** command from the last commit point. For more information about this, see "Creating the restart table" on page 175.

About this task

For a list of the required privileges and authorities, see the **INGEST** command authorization.

Restrictions

For a comprehensive list of restrictions for the ingest utility, see "Ingest utility restrictions and limitations" on page 187.

Procedure

Issue the **INGEST** command specifying, at a minimum, a source, the format, and the target table as in the following example:

```
INGEST FROM FILE <source_file>
    FORMAT DELIMITED
    INSERT INTO <table-name>;
```

It is recommended that you also specify a string with the **RESTART NEW** parameter on the **INGEST** command:

```
INGEST FROM FILE <source_file>
   FORMAT DELIMITED
   RESTART NEW 'CDIjob001'
   INSERT INTO <table-name>;
```

The string you specify can be up to 128 bytes. Because the string uniquely identifies the **INGEST** command, it must be unique across all **INGEST** commands in the current database that specified the **RESTART NEW** option and are not yet complete.

Example

Basic ingest examples

The following example inserts data from a delimited text file:

```
INGEST FROM FILE <source_file>
    FORMAT DELIMITED
    INSERT INTO <table-name>
```

The following example inserts data from a delimited text file with fields separated by a comma (the default). The fields in the file correspond to the table columns.

```
INGEST FROM FILE <source_file>
   FORMAT DELIMITED
   (
      $field1 INTEGER EXTERNAL,
      $field2 DATE 'mm/dd/yyyy',
      $field3 CHAR(32)
   )
   INSERT INTO <table-name>
      VALUES($field1, $field2, $field3);
```

Delimiter override example

The following example inserts data like the previous example, but the fields are separated by a vertical bar.

```
INGEST FROM FILE <source_file>
 FORMAT DELIMITED by '|'
 (
  $field1 INTEGER EXTERNAL,
  $field2 DATE 'mm/dd/yyyy',
  $field3 CHAR(32)
 )
 INSERT INTO <table-name>
  VALUES($field1, $field2, $field3);
```

Omitting the field definition and VALUES list example

In the following example, the table is defined as follows:

```
CREATE TABLE <table-name> (
    c1 VARCHAR(32),
    c2 INTEGER GENERATED BY DEFAULT AS IDENTITY,
    c3 INTEGER GENERATED ALWAYS AS (c2 + 1),
   );
```

The user issues the following **INGEST** command:

```
INGEST FROM FILE <source_file>
    FORMAT DELIMITED
    INSERT INTO <table-name>;
```

- The default field definition list will be:

  ```
  (
      $C1 CHARACTER(32),
      $C2 INTEGER EXTERNAL,
      $C3 INTEGER EXTERNAL
  )
  ```

- The default VALUES list on the INSERT statement is:

  ```
  VALUES($C1, $C2, DEFAULT)
  ```

  Note that the third value is DEFAULT because the column that corresponds to field $C3 is defined as GENERATED ALWAYS. The fourth value is omitted because it has no field.

Extra fields used to compute column values example

The following example is the same as the delimiter override example, but only the first two fields correspond to the first two table columns (PROD_ID and DESCRIPTION), whereas the value for the third table column (TOTAL_PRICE) is computed from the remaining three fields

```
INGEST FROM FILE <source_file>
 FORMAT DELIMITED BY '|'
 (
  $prod_ID     CHAR(8),
  $description CHAR(32),
  $price       DECIMAL(5,2) EXTERNAL,
  $sales_tax   DECIMAL(4,2) EXTERNAL,
  $shipping    DECIMAL(3,2) EXTERNAL
 )
 INSERT INTO <table-name>(prod_ID, description, total_price)
  VALUES($prod_id, $description, $price + $sales_tax + $shipping);
```

Filler fields example

The following example inserts data from a delimited text file with fields separated by a comma (the default). The fields in the file correspond to the table columns except that there are extra fields between the fields for columns 2 and 3 and columns 3 and 4.

```
INGEST FROM FILE <source_file>
 FORMAT DELIMITED
 (
  $field1  INTEGER,
  $field2  CHAR(8),
  $filler1 CHAR,
  $field3  CHAR(32),
  $filler2 CHAR,
  $field4  DATE
 )
 INSERT INTO <table-name> VALUES($field1, $field2, $field3, $field4);
```

Format modifiers example

The following example inserts data from a delimited text file in code page 850. Date fields are in American format and char fields are enclosed in equal signs.

```
INGEST FROM FILE <source_file>
 FORMAT DELIMITED
 INPUT CODEPAGE 850
 (
  $field1 INTEGER EXTERNAL,
  $field2 DATE 'mm/dd/yyyy',
  $field3 CHAR(32) ENCLOSED BY '='
 )
 INSERT INTO <table-name>
  VALUES($field1, $field2, $field3);
```

Positional example

The following example inserts data from a file with fields in the specified positions. The fields in the file correspond to the table columns.

```
INGEST FROM FILE <source_file>
    FORMAT POSITIONAL
 (
  $field1 POSITION(1:8)   INTEGER EXTERNAL,
  $field2 POSITION(10:19) DATE 'yyyy-mm-dd',
  $field3 POSITION(25:34) CHAR(10)
 )
 INSERT INTO <table-name>
  VALUES($field1, $field2, $field3);
```

DEFAULTIF examples

This example is similar to the previous example, except if the second field starts with a blank, the ingest utility inserts the default value:

```
INGEST FROM FILE <source_file>
 FORMAT POSITIONAL
 (
  $field1 POSITION(1:8) INTEGER EXTERNAL,
  $field2 POSITION(10:19) DATE 'yyyy-mm-dd' DEFAULTIF = ' ',
    $field3 POSITION(25:34) CHAR(10)
 )
 INSERT INTO <table-name>
  VALUES($field1, $field2, $field3);
```

This example is the same as the previous example, except that the default indicator is in the column after the data columns:

```
INGEST FROM FILE <source_file>
 FORMAT POSITIONAL
 (
  $field1 POSITION(1:8)   INTEGER EXTERNAL,
  $field2 POSITION(10:19) DATE 'yyyy-mm-dd' DEFAULTIF(35) = ' ',
    $field3 POSITION(25:34) CHAR(10)
 )
 INSERT INTO <table-name>
  VALUES($field1, $field2, $field3);
```

Multiple input sources example

This example inserts data from three delimited text files:

```
INGEST FROM FILE <source_file>, <source_file2>, <source_file3>
 FORMAT DELIMITED
 (
  $field1 INTEGER EXTERNAL,
  $field2 DATE 'mm/dd/yyyy',
```

```
 $field3 CHAR(32)
)
INSERT INTO <table-name>
 VALUES($field1, $field2, $field3);
```

Pipe example

This example inserts data from a pipe:

```
INGEST FROM PIPE my_pipe
 FORMAT DELIMITED
 (
  $field1 INTEGER EXTERNAL,
  $field2 DATE 'mm/dd/yyyy',
  $field3 CHAR(32)
 )
 INSERT INTO <table-name>
  VALUES($field1, $field2, $field3);
```

Options example

This example inserts data from a delimited text file with fields separated by a comma (the default). The fields in the file correspond to the table columns. The command specifies that write rows rejected by Db2 (for example, due to constraint violations) are to be written to table EXCP_TABLE, rows rejected due to other errors are to be discarded, and messages are to be written to file `messages.txt`.

```
INGEST FROM FILE <source_file>
 FORMAT DELIMITED
 (
  $field1 INTEGER EXTERNAL,
  $field2 DATE 'mm/dd/yyyy',
  $field3 CHAR(32)
 )
 EXCEPTION TABLE excp_table
    MESSAGES messages.txt
 INSERT INTO <table-name>
  VALUES($field1, $field2, $field3);
```

Restart example

This example issues an **INGEST** command (which is restartable, by default) with a specified ingest job id:

```
INGEST FROM FILE <source_file>
 FORMAT DELIMITED
 (
  $field1 INTEGER EXTERNAL,
  $field2 DATE 'mm/dd/yyyy',
  $field3 CHAR(32)
 )
 RESTART NEW 'ingestcommand001'
 INSERT INTO <table-name>
  VALUES($field1, $field2, $field3);
```

If the command terminates before completing, you can restart it with the following command:

```
INGEST FROM FILE <source_file>
 FORMAT DELIMITED
 (
  $field1 INTEGER EXTERNAL,
  $field2 DATE 'mm/dd/yyyy',
  $field3 CHAR(32)
 )
 RESTART CONTINUE 'ingestcommand001'
 INSERT INTO <table-name>
  VALUES($field1, $field2, $field3);
```

Restart terminate example

This example issues the same **INGEST** command as the previous "Restart example":

```
INGEST FROM FILE <source_file>
 FORMAT DELIMITED
 (
  $field1 INTEGER EXTERNAL,
  $field2 DATE 'mm/dd/yyyy',
  $field3 CHAR(32)
 )
 RESTART NEW 'ingestcommand001'
 INSERT INTO <table-name>
  VALUES($field1, $field2, $field3);
```

If the command terminates before completing and you do not plan to restart it, you can clean up the restart records with the following command.

```
INGEST FROM FILE <source_file>
 FORMAT DELIMITED
 (
  $field1 INTEGER EXTERNAL,
  $field2 DATE 'mm/dd/yyyy',
  $field3 CHAR(32)
 )
 RESTART TERMINATE 'ingestcommand001'
 INSERT INTO <table-name>
  VALUES($field1, $field2, $field3);
```

After issuing this command, you can no longer restart the **INGEST** command with the job id: "ingestcommand001", but you can reuse that string on the RESTART NEW parameter of a new **INGEST** command.

Reordering columns example

This example inserts data from a delimited text file with fields separated by a comma. The table has three columns and the fields in the input data are in the reverse order of the table columns.

```
INGEST FROM FILE <source_file>
 FORMAT DELIMITED
 (
  $field1 INTEGER EXTERNAL,
  $field2 DATE 'mm/dd/yyyy',
  $field3 CHAR(32)
 )
 INSERT INTO <table-name>
  VALUES($field3, $field2, $field1);
```

Basic UPDATE, MERGE, and DELETE examples

The following examples update the table rows whose primary key matches the corresponding fields in the input file.

```
INGEST FROM FILE <source_file>
 FORMAT DELIMITED
 (
  $key1  INTEGER EXTERNAL,
  $key2  INTEGER EXTERNAL,
  $data1 CHAR(8),
  $data2 CHAR(32),
  $data3 DECIMAL(5,2) EXTERNAL
  )
 UPDATE <table-name>
  SET (data1, data2, data3) = ($data1, $data2, $data3)
  WHERE (key1 = $key1) AND (key2 = $key2);
```

or

```
INGEST FROM FILE <source_file>
 FORMAT DELIMITED
 (
  $key1  INTEGER EXTERNAL,
  $key2  INTEGER EXTERNAL,
  $data1 CHAR(8),
  $data2 CHAR(32),
  $data3 DECIMAL(5,2) EXTERNAL
  )
 UPDATE <table-name>
  SET data1 = $data1, data2 = $data2, data3 = $data3
  WHERE (key1 = $key1) AND (key2 = $key2);
```

This example merges data from the input file into the target table. For input rows whose primary key fields match a table row, it updates that table row with the input row. For other input rows, it adds the row to the table.

```
INGEST FROM FILE <source_file>
 FORMAT DELIMITED
 (
  $key1  INTEGER EXTERNAL,
  $key2  INTEGER EXTERNAL,
  $data1 CHAR(8),
  $data2 CHAR(32),
  $data3 DECIMAL(5,2) EXTERNAL
  )
 MERGE INTO <table-name>
  ON (key1 = $key1) AND (key2 = $key2)
  WHEN MATCHED THEN
   UPDATE SET (data1, data2, data3) = ($data1, $data2, $data3)
  WHEN NOT MATCHED THEN
   INSERT VALUES($key1, $key2, $data1, $data2, $data3);
```

This example deletes table rows whose primary key matches the corresponding primary key fields in the input file.

```
INGEST FROM FILE <source_file>
 FORMAT DELIMITED
 (
  $key1  INTEGER EXTERNAL,
  $key2  INTEGER EXTERNAL
  )
 DELETE FROM <table-name>
  WHERE (key1 = $key1) AND (key2 = $key2);
```

Complex SQL examples

Consider the following example in which there is a table with columns KEY, DATA, and ACTION. The following command updates the DATA column of table rows where the primary key column (KEY) matches the corresponding field in the input file and the ACTION column is 'U':

```
INGEST FROM FILE <source_file>
 FORMAT DELIMITED
 (
  $key_fld  INTEGER EXTERNAL,
  $data_fld INTEGER EXTERNAL
 )
 UPDATE <table-name>
  SET data = $data_fld
   WHERE (key = $key_fld) AND (action = 'U');
```

The following example is the same as the previous example except that if the keys match and the ACTION column is 'D', then it deletes the row from the table:

```
INGEST FROM FILE <source_file>
 FORMAT DELIMITED
 (
  $key_fld  INTEGER EXTERNAL,
  $data_fld INTEGER EXTERNAL
 )
 MERGE INTO <table-name>
  ON (keyl = $key_fld)
  WHEN MATCHED AND (action = 'U') THEN
   UPDATE SET data = $data_fld
  WHEN MATCHED AND (action = 'D') THEN
   DELETE;
```

What to do next

If the **INGEST** command completes successfully, you can reuse the string specified with the **RESTART NEW** parameter.

If the **INGEST** command fails and you want to restart it, you must specify the **RESTART CONTINUE** option with the string you specified in the original command.

If you do not plan to restart the failed **INGEST** command and you want to clean up the entries in the restart table, rerun the **INGEST** command, specifying the **RESTART TERMINATE** option.

Restarting a failed ingest operation:

If an **INGEST** command fails before completing and you want to restart it, reissue the **INGEST** command with the RESTART CONTINUE option. This second **INGEST** command starts from the last commit point and is also restartable.

Before you begin

The userid restarting the failed **INGEST** command must have SELECT, INSERT, UPDATE, and DELETE privilege on the restart log table.

About this task

The **INGEST** utility considers a command to be complete when it reaches the end of the file or pipe. Under any other conditions, the **INGEST** utility considers the command incomplete. These can include:

- The **INGEST** command gets an I/O error while reading the input file or pipe.
- The **INGEST** command gets a critical system error from the Db2 database system.
- The **INGEST** command gets a Db2 database system error that is likely to prevent any further SQL statements in the **INGEST** command from succeeding (for example, if the table no longer exists).
- The **INGEST** command is killed or terminates abnormally.

Restrictions

1. If the target table and the restart table are in different table spaces, the two table spaces must be at the same level in terms of rollforward or restore operations.

2. You cannot modify the contents of the restart table, other than restoring the entire table to keep it in sync with the target table.
3. The **num_flushers_per_partition** configuration parameter must be the same as on the original command.
4. If the input is from files or pipes, the number of input files or pipes must be the same as on the original command.
5. The input file or pipes must provide the same records and in the same order as on the original command.
6. The following **INGEST** command parameters must be the same as on the original command:
   - input type (file or pipe)
   - the SQL statement
   - the field definition list, including the number of fields and all field attributes
7. The target table columns that the SQL command updates must have the same definition as they had at the time of the original command.
8. In a partitioned database environment, you cannot have added or removed database partitions.
9. In a partitioned database environment, you cannot have redistributed data across the partitions.
10. If an **INGEST** command specifies the DUMPFILE (BADFILE) parameter, the dump file is guaranteed to be complete only if the **INGEST** command completes normally in a single run. If an **INGEST** command fails and the restarted command succeeds, the combination of dump files from the two commands might be missing some records or might contain duplicate records.

If the third, fourth, fifth, or ninth restriction is violated, the ingest utility issues an error and ends the **INGEST** command. In the case of the other restrictions, the ingest utility does not issue an error, but the restarted **INGEST** command might produce different output rows than the original would have if it had completed.

Procedure

To restart a failed **INGEST** operation, do the following:

1. Use the available information to diagnose and correct the problem that caused the failure
2. Reissue the **INGEST** command, specifying the **RESTART CONTINUE** option with the appropriate job-id.

Results

Once the restarted **INGEST** command completes, you can reuse the job-id on a later **INGEST** command.

Example

The following **INGEST** command failed:

```
INGEST FROM FILE <source_file>
   FORMAT DELIMITED
   (
      $field1 INTEGER EXTERNAL,
      $field2 DATE 'mm/dd/yyyy',
      $field3 CHAR(32)
```

```
   )
   RESTART NEW 'ingestjob001'
   INSERT INTO <table-name>
      VALUES($field1, $field2, $field3);
```

The DBA corrects the problem that cause the failure and restarts the **INGEST** command (which starts from the last commit point) with the following command:

```
INGEST FROM FILE <source_file>
   FORMAT DELIMITED
   (
      $field1 INTEGER EXTERNAL,
      $field2 DATE 'mm/dd/yyyy',
      $field3 CHAR(32)
   )
   RESTART CONTINUE 'ingestjob001'
   INSERT INTO <table-name>
      VALUES($field1, $field2, $field3);
```

Terminating a failed ingest operation:

If an **INGEST** command fails before completing and you do not want to restart it, reissue the **INGEST** command with the **RESTART TERMINATE** option. This command option cleans up the log records for the failed **INGEST** command.

Before you begin

The user ID terminating the failed **INGEST** command must have SELECT and DELETE privilege on the restart log table.

Procedure

To terminate a failed **INGEST** operation, reissue the **INGEST** command. Specify the **RESTART TERMINATE** parameter with the appropriate string.

Results

After the restarted **INGEST** command completes, you can reuse the **RESTART NEW** string on a later **INGEST** command.

Example

The following **INGEST** command failed:

```
INGEST FROM FILE <source_file>
   FORMAT DELIMITED
   (
      $field1 INTEGER EXTERNAL,
      $field2 DATE 'mm/dd/yyyy',
      $field3 CHAR(32)
   )
   RESTART NEW 'ingestjob001'
   INSERT INTO <table-name>
      VALUES($field1, $field2, $field3);
```

The DBA does not want to restart the **INGEST** command, so they terminate it with the following command (which includes the **RESTART TERMINATE** parameter):

```
INGEST FROM FILE <source_file>
   FORMAT DELIMITED
   (
      $field1 INTEGER EXTERNAL,
      $field2 DATE 'mm/dd/yyyy',
```

```
   $field3 CHAR(32)
)
RESTART TERMINATE 'ingestjob001'
INSERT INTO <table-name>
   VALUES($field1, $field2, $field3);
```

Monitoring ingest operations

You can use the **INGEST LIST** or **INGEST GET STATS** commands to monitor the progress of **INGEST** commands.

Before you begin

To issue the **INGEST LIST** and **INGEST GET STATS** commands, you need a separate CLP session but they must be run on the same machine that the **INGEST** command is running on.

Procedure

There are a number of ways to monitor an ingest operation:

- To get basic information about all currently running **INGEST** commands, use the **INGEST LIST** command.
- To get more detailed information about a specific **INGEST** command or all currently running **INGEST** commands, use the **INGEST GET STATS** command.
- You can also query the following monitor elements by using an interface such as the MON_GET_CONNECTION table function:
  - **client_acctng**
  - **client_applname**
  - **appl_name**
  - **client_userid**
  - **client_wrkstnname**

Example

The following shows an example of what output to expect from an **INGEST LIST** command:

```
INGEST LIST

Ingest job ID       = DB21000:20101116.123456.234567:34567:45678
Ingest temp job ID  = 1
Database Name       = MYDB
Input type          = FILE
Target table        = MY_SCHEMA.MY_TABLE
Start Time          = 04/10/2010 11:54:45.773215
Running Time        = 01:02:03
Number of records processed = 30,000
```

The following shows an example of what output to expect from an **INGEST GET STATS** command:

```
INGEST GET STATS FOR 4

Ingest job ID = DB21000:20101116.123456.234567:34567:4567
Database      = MYDB
Target table  = MY_SCHEMA.MY_TABLE1

Records/sec      Flushes/sec       Records/sec     Flushes/sec
```

```
since start      since start       since last      since last       Total records
---------------- ----------------- --------------- ---------------- -------------
54321            65432             76543           87654            98765
```

The following shows an example of using the MON_GET_CONNECTION table function to get the number of rows modified and the number of commits:

```
SELECT client_acctng AS "Job ID",
      SUM(rows_modified) AS "Total rows modified",
      SUM(total_app_commits) AS "Total commits"
   FROM TABLE(MON_GET_CONNECTION(NULL, NULL))
   WHERE application_name = 'DB2_INGEST'
   GROUP BY client_acctng
   ORDER BY 1

Job ID                                          Total rows modified Total commits
----------------------------------------------- ------------------- -------------
DB21000:20101116.123456.234567:34567:45678                       92            52
DB21000:20101116.987654.234567:34567:45678                      172           132

  2 record(s) selected.
```

Ingest utility restrictions and limitations

There are a number of restrictions that you should be aware of when using the ingest utility.

Restartability

- If input data source type changed, the ingest utility might not be able to detect the change and will produce different output rows than the original failed command.

Table support

- The ingest utility supports operations against only Db2 tables.
- The ingest utility does not support operations on:
  - created or declared global temporary tables
  - typed tables
  - typed views

Input types, formats, and column types

- The ingest utility does not support the following column types:
  - large object types (LOB, BLOB, CLOB, DBCLOB)
  - XML
  - structured types
  - columns with a user-defined data type based on any of the types listed previously
- In addition, the ingest utility has the following restrictions on generated columns:
  - The ingest utility cannot assign a value to a column defined as GENERATED ALWAYS. If the SQL statement on the **INGEST** command is INSERT or UPDATE and the target table has a GENERATED ALWAYS column, the insert or update operation fails (SQL0798N) and the **INGEST** command ends unless you do one of the following:
    - Omit the column from the list of columns to update.
    - On the INSERT or UPDATE statement, specify DEFAULT as the value assigned to the column.

– The ingest utility cannot assign a combination of default values and specific values to a column defined as GENERATED BY DEFAULT AS IDENTITY or the RANDOM_DISTRIBUTION_KEY of a random distribution table using the random by generation method. If the SQL statement on the **INGEST** command is INSERT or UPDATE and the target table has a GENERATED BY DEFAULT AS IDENTITY column, the insert or update operation fails (SQL0407N) and the **INGEST** command rejects the record unless you do one of the following:
  - Omit the column from the list of columns to update.
  - On the INSERT or UPDATE statement, specify DEFAULT as the value assigned to the column.
  - Specify an expression that never evaluates to NULL as the value assigned to the column. For example, if the expression is $field1, then $field1 can never have a NULL value in the input records.

Restrictions related to using other Db2 features with the ingest utility

- Except for the **CONNECT_MEMBER** parameter, the **SET CLIENT** command (for connection settings) does not affect how the ingest utility connects.
- The **LIST HISTORY** command does not display ingest operations.
- The **SET UTIL_IMPACT_PRIORITY** command does not affect the **INGEST** command
- The **util_impact_lim** database manager configuration parameter does not affect the **INGEST** command
- Except for CURRENT SCHEMA, CURRENT TEMPORAL SYSTEM_TIME, and CURRENT TEMPORAL BUSINESS_TIME, the ingest utility ignores the settings of most special registers that affect SQL statement execution.

General ingest utility restrictions

- If you ingest into a view that has multiple base tables, any base tables that are protected by a security policy must be protected by the same security policy. (You can still have some base tables unprotected but those that are protected must use the same security policy.)

Nickname support

- If the **INGEST** command specifies or defaults to the `RESTART NEW` or `RESTART CONTINUE` option, and the target table is a nickname or an updatable view that updates a nickname, ensure that the DB2_TWO_PHASE_COMMIT server option is set to 'Y' for the server definition that contains the nickname.
- You cannot use the **SET SERVER OPTION** to enable two-phase commit before issuing the **INGEST** command because that command affects only the CLP connection, whereas the **INGEST** command establishes its own connection. You must set the server option in the server definition in the catalog.
- You cannot use the DB2_TWO_PHASE_COMMIT server option with the database partitioning feature, which means that the combination of partitioned database environment mode, a restartable ingest command, and ingesting into a nickname is not supported.
- The performance benefit of the utility is reduced when used on nicknames.

Additional considerations for ingest operations

Performance considerations for ingest operations

Use the following set of guidelines to help performance tune your ingest jobs.

Field type and column type
Define fields to be the same type as their corresponding column types. When the types are different, the ingest utility or Db2 must convert the input data to the column type.

Materialized query tables (MQTs)
If you ingest data into a table that is a base table of an MQT defined as REFRESH IMMEDIATE, performance can degrade significantly due to the time required to update the MQT.

Row size
For tables with a small row size, increase the setting of the **commit_count** ingest configuration parameter; for tables with a large row size, reduce the setting of the **commit_count** ingest configuration parameter.

Other workloads
If you are executing the ingest utility with another workload, increase the setting of the **locklist** database configuration parameter and reduce the setting of the **commit_count** ingest configuration parameter

Code page considerations for the ingest utility

When the ingest utility processes input data, there are three code pages involved: the application (client) code page, the input data code page, and the database code page.

Code page	How specified	Default
Application (client) code page, which is used in the CLP command file	Determined from the current locale	Determined from the current locale
Input data code page	`INPUT CODEPAGE` on the **INGEST** command	Application code page
Database code page	Specified on the **CREATE DATABASE** command	1208 (UTF-8 encoding of Unicode)

If the input data code page differs from the application code page, the ingest utility temporarily overrides the application code page with the input data code page so that Db2 converts the data directly from the input data code page to the database code page. Under some conditions, the ingest utility cannot override the application code page. In this case, the ingest utility converts character data that is not defined as FOR BIT DATA to the application code page before passing it to Db2. In all cases, if the column is not defined as FOR BIT DATA, Db2 converts the data to the database code page.

CLP command file code page
Except for hex constants, the ingest utility assumes that the text of the **INGEST** command is in the application code page. Whenever the ingest utility needs to compare strings specified on the **INGEST** command (for example, when comparing the DEFAULTIF character to a character in the input data), the ingest utility performs any necessary code page conversion to ensure the compared strings are in the same code page. Neither the ingest utility nor Db2 do any conversion of hex constants.

Input data code page

If both a field and the table column that it is assigned to are defined as FOR BIT DATA, then neither the ingest utility nor Db2 does any code page conversion. For example, suppose that the **INGEST** command assigns field $c1 to column C1 and both are defined as CHAR FOR BIT DATA. If the input field contains X'E9', then Db2 sets column C1 to X'E9', regardless of the input data code page or database code page.

It is strongly recommended that if a column definition omits FOR BIT DATA, then its corresponding field definition also omit FOR BIT DATA. Likewise, if a column definition specifies FOR BIT DATA, its corresponding field should also specify FOR BIT DATA. Otherwise, the value assigned to the column is unpredictable because it depends on whether the ingest utility can override the application code page.

The following example illustrates this situation:

- The input data code page is 819.
- The application code page is 850.
- The database code page is 1208 (UTF-8).
- The input data is "é" ("e" with an acute accent), which is X'E9' in code page 819, X'82' in code page 850, and X'C3A9' in UTF-8.

The following table shows what data ends up on the server depending on whether the field and/or column are defined as FOR BIT DATA and whether the ingest utility can override the application code page:

Table 22. Possible outcomes if the field and column definitions are defined as FOR BIT DATA

Field definition	Column definition	Input data (code page 819)	Data after the ingest utility converts it to application code page 850	Data on the server if the ingest utility can override the application code page	Data on the server if the ingest utility cannot override the application code page
CHAR	CHAR	X'E9'	X'82'	X'C3A9'	X'C3A9'
CHAR FOR BIT DATA	CHAR FOR BIT DATA	X'E9'	X'E9'	X'E9'	X'E9'
CHAR FOR BIT DATA	CHAR	X'E9'	X'E9'	**X'C3A9'**	**X'C39A' ("Ú")**
CHAR	CHAR FOR BIT DATA	X'E9'	X'82'	**X'E9'**	**X'82'**

The data in the fourth column is what the ingest utility sends to Db2 when it can override the application code page. The data in the fourth column is what the ingest utility sends when it cannot override the application code page. Note that when the FOR BIT DATA attribute of the field and column definitions are different, the results can vary as shown in the preceding table.

Code page errors

In cases where the input code page, application code page, or database code page differ, either the ingest utility or Db2 or both will perform code page conversion. If Db2 does not support the code page conversion in any of the following cases, the ingest utility issues an error and the command ends.

Conversion is required when...	In this case, conversion from...	To...	Is done by...
The **INGEST** command contains strings or SQL identifiers that need to be converted to the input data code page.	Application code page	Input data code page	Ingest utility
The utility can override the application code page to be the input data code page.	Input code page	Database code page	Db2
The utility cannot override the application code page.	Input code page	Application code page	Ingest utility
The utility cannot override the application code page.	Application code page	Database code page	Db2

Ingest operations in a Db2 pureScale environment

When you are using the ingest utility in a Db2 pureScale environment, there are some additional considerations to take into account.

When each flusher connects to a database on a Db2 pureScale instance:

- If the **SET CLIENT** command with the **CONNECT_MEMBER** option has been issued, the flusher connects to that member.
- Otherwise, the flusher does not specify the member to connect to. In this case, the Db2 client uses connection-level work load balancing (WLB) to select the member to connect to.

Multiple invocations of the ingest utility can operate on the same member or on different members, depending on whether the **SET CLIENT** command with the **CONNECT_MEMBER** option has been specified and which member WLB chooses.

Ingest operations in a partitioned database environment

You can use the ingest utility to move data into a partitioned database environment.

INGEST commands running on a partitioned database use one or more flushers for each partition, as specified by the **num_flushers_per_partition** configuration parameter. The default is as follows:

```
max(1, ((number of logical CPUs)/2)/(number of partitions) )
```

You can also set this parameter to 0, meaning one flusher for all partitions.

Each flusher connects directly to the partition to which it will send data. In order for the connection to succeed, all the Db2 server partitions must use the same port number to receive client connections.

If the target table is a type that has a distribution key, the ingest utility determines the partition that each record belongs to as follows:

1. Determine whether every distribution key has exactly one corresponding field or constant value. This will be true if:
   - For an INSERT statement, the column list contains every distribution key and for each distribution key, the corresponding item in the VALUES list is a field name or a constant.
   - For an UPDATE or DELETE statement, the WHERE predicate is of the form

     ```
     (dist-key-col1 = value1) AND (dist-key-col2 = value2) AND ...
     (dist-key-coln = valuen) [AND any-other-conditions]
     ```

     where *dist-keycol1* to *dist-key-coln* are all the distribution keys and each *value* is a field name or a constant.
   - For a MERGE statement, the search condition is of the form shown previously for UPDATE and DELETE.
2. If every distribution key has exactly one corresponding field or constant value, the ingest utility uses the distribution key to determine the partition number and then routes the record to one of that partition's flushers.

   Note: In the following cases, the ingest utility does not determine the record's partition. If there is more than 1 flusher, the ingest utility routes the record to a flusher chosen at random:
   - The target table is a type that has no distribution key.
   - The column list (INSERT) or predicate (UPDATE, MERGE, DELETE) does not specify all distribution keys. In the following example, key columns 2-8 are missing:

     ```
     UPDATE my_table SET data = $data
         WHERE (key1 = $key1) AND (key9 = $key9);
     ```
   - A distribution key corresponds to more than one field or value, as in the following example:

     ```
     UPDATE my_table SET data = $data
         WHERE key1 = $key11 OR key1 = $key12;
     ```
   - A distribution key corresponds to an expression, as in the following example

     ```
     INGEST FROM FILE ...
         INSERT INTO my_table(dist_key, col1, col2)
             VALUES($field1 + $field2, $col1, $col2);
     ```
   - A distribution key column has type DB2SECURITYLABEL.
   - A field that corresponds to a distribution key has a numeric type, but the distribution key column type is a different numeric type or has a different precision or scale.

Sample ingest utility scripts

You can use the ingest utility sample script to automate writing a new INGEST command each time there are new files to process.

The sample script `ingest_files.sh` is a shell script that automatically checks for new files and generates an INGEST command to process the files. The script performs the following tasks, in order:

1. Check the directory to see if there are new files to process. If there are no files, the script exits.

   Note: The script assumes that the specified directory only contains files for the table that you want to populate.
2. Obtain the names of the new files and then generate a separate INGEST command for each file

3. Run the INGEST command and handle the return code
4. Move the processed files to a success directory or a failed directory.

The script is provided in the `samples/admin_scripts` directory under your installation directory.

Modifying the script for your environment

You can use the `ingest_files.sh` script as a basis for your own script. The important modifications that you have to make to it are:

- Replace the sample values (namely, the database name, table name) with you own values
- Replace the sample INGEST command with your own command
- Create the directories specified in the script

The script processes files that contain data to populate a single table. To populate multiple tables, you can either replicate the mechanism for each table that you want to populate or generalize the mechanism to handle multiple tables.

Sample scenario

A sample scenario has been included in the documentation to show you how you can adapt the sample script to your data warehouse to automate the generation of new INGEST commands.

Scenario: Processing a stream of files with the ingest utility

The following scenario shows how you can configure your data warehouse to automatically ingest an ongoing stream of data files.

The problem: In some data warehouses, files arrive in an ongoing stream throughout the day and need to be processed as they arrive. This means that each time a new file arrives, another **INGEST** command needs to be run specifying the new file to process.

The solution: You can write a script that automatically checks for new files, generates a new **INGEST** command, and runs that command. The `ingest_files.sh` is a sample of such a script. You also need to create a crontab entry in order to specify how frequently the shell script is supposed to run.

Before the user implements this mechanism (that is, the script and the chrontab entry) for processing the stream of files, the user needs to have met the following prerequisites and dependencies:

- The target table has been created in the target database
- The ingest utility is ready to use (that is, it is installed and set up on a client machine)
- An INGEST command has been specified and verified by running it manually with a test file
- The objects, such as the exception table, referenced in the **INGEST** command have been created
- A crontab file has been created on the system on which the ingest utility is running
- The user has a process for creating the input files and moving them into the source directory that the script uses

1. The user creates a new script, using `ingest_files.sh` as a template by doing the following:
   a. Replace the following sample input values to reflect the user's values:
      - INPUT_FILES_DIRECTORY
      - DATABASE_NAME
      - SCHEMA_NAME
      - TABLE_NAME
      - SCRIPT_PATH
   b. Replace the sample **INGEST** command
   c. Save the script as `populate_table1_script`
2. The user adds an entry to the crontab file to specify how frequently the script is to run. Because the user wants the script to run once a minute, 24 hours a day, every day of the year, the user adds the following line:
   ```
   1 * * * * $HOME/bin/populate_table1_script
   ```
3. The user tests the script by creating new input files and adding them to the source directory.

Other data movement options

Moving tables online by using the ADMIN_MOVE_TABLE procedure

Using the ADMIN_MOVE_TABLE procedure, you can move tables by using an online or offline move. Use an online table move instead of an offline table move if you value availability more than cost, space, move performance, and transaction overhead.

Before you begin

Ensure there is sufficient disk space to accommodate the copies of the table and index, the staging table, and the additional log entries.

About this task

You can move a table online by calling the stored procedure once or multiple times, one call for each operation performed by the procedure. Using multiple calls provides you with additional options, such as cancelling the move or controlling when the target table is taken offline to be updated.

When you call the SYSPROC.ADMIN_MOVE_TABLE procedure, a shadow copy of the source table is created. During the copy phase, changes to the source table (updates, insertions, or deletions) are captured using triggers and placed in a staging table. After the copy phase is completed, the changes captured in the staging table are replayed to the shadow copy. Following that, the stored procedure briefly takes the source table offline and assigns the source table name and index names to the shadow copy and its indexes. The shadow table is then brought online, replacing the source table. By default, the source table is dropped, but you can use the KEEP option to retain it under a different name.

Avoid performing online moves for tables without indexes, particularly unique indexes. Performing a online move for a table without a unique index might result in deadlocks and complex or expensive replay.

Applications holding conflicting locks on the source table might fail with SQL0911N reason code 68, because ADMIN_MOVE_TABLE is more likely to be successful in lock timeout conflicts. To prevent deadlocks during the SWAP operation, the FORCE_ALL option can be used. See FORCE_ALL for more details.

Procedure

To move a table online:

1. Call the ADMIN_MOVE_TABLE procedure in one of the following ways:
   - Call the ADMIN_MOVE_TABLE procedure once, specifying at least the schema name of the source table, the source table name, and an operation type of MOVE. For example, use the following syntax to move the data to an existing table within the same table space:

     ```
     CALL SYSPROC.ADMIN_MOVE_TABLE (
     'schema name',
     'source table',
     '',
     '',
     '',
     '',
     '',
     '',
     '',
     '',
     '',
     'MOVE')
     ```

   - Call the ADMIN_MOVE_TABLE procedure multiple times, once for each operation, specifying at least the schema name of the source table, the source table name, and an operation name. For example, use the following syntax to move the data to a new table within the same table space:

     ```
     CALL SYSPROC.ADMIN_MOVE_TABLE (
     'schema name',
     'source table',
     '',
     '',
     '',
     '',
     '',
     '',
     '',
     '',
     '',
     'operation name')
     ```

     where *operation name* is one of the following values: INIT, COPY, REPLAY, VERIFY, or SWAP. You must call the procedure based on this order of operations, for example, you must specify INIT as the operation name in the first call.

     Note: The VERIFY operation is costly; perform this operation only if you require it for your table move.
2. If the online move fails, rerun it:
   a. Fix the problem that caused the table move to fail.
   b. Determine the stage that was in progress when the table move failed by querying the SYSTOOLS.ADMIN_MOVE_TABLE protocol table for the status.
   c. Call the stored procedure again, specifying the applicable option:
      - If the status of the procedure is INIT, use the INIT option.
      - If the status of the procedure is COPY, use the COPY option.

- If the status of the procedure is REPLAY, use the REPLAY or SWAP option.
- If the status of the procedure is CLEANUP, use the CLEANUP option.

If the status of an online table move is not COMPLETED or CLEANUP, you can cancel the move by specifying the CANCEL option for the stored procedure.

Examples

Example 1: Move the T1 table from schema SVALENTI, to the ACCOUNTING table space without taking T1 offline. Specify the DATA, INDEX, and LONG table spaces to move the table into a new table space.

```
CALL SYSPROC.ADMIN_MOVE_TABLE(
'SVALENTI',
'T1',
'ACCOUNTING',
'ACCOUNTING',
'ACCOUNTING',
'',
'',
'',
'',
'',
'MOVE')
```

Example 2: Move the T1 table from schema EBABANI to the ACCOUNTING table space without taking T1 offline, and keep a copy of the original table after the move. Use the COPY_USE_LOAD and LOAD_MSGPATH options to set the load message file path. Specify the DATA, INDEX, and LONG table spaces to move the table into a new table space. The original table will maintain a name similar to 'EBABANI'.'T1AAAAVxo'.

```
CALL SYSPROC.ADMIN_MOVE_TABLE(
'EBABANI',
'T1',
'ACCOUNTING',
'ACCOUNTING',
'ACCOUNTING',
'',
'',
'',
'',
'KEEP, COPY_USE_LOAD,LOAD_MSGPATH "/home/ebabani"',
'MOVE')
```

Example 3: Move the T1 table within the same table space. Change the C1 column within T1, which uses the deprecated datatype LONG VARCHAR to use a compatible data type.

```
CALL SYSPROC.ADMIN_MOVE_TABLE(
'SVALENTI',
'T1',
'',
'',
'',
'',
'',
'',
'C1 VARCHAR(1000), C2 INT(5), C3 CHAR(5), C4 CLOB',
'',
'MOVE')
```

Note: You cannot change the column name during this operation.

Example 4: You have the T1 table created by the following statement:

```
CREATE TABLE T1(C1 BIGINT,C2 BIGINT,C3 CHAR(20),C4 DEC(10,2),C5 TIMESTAMP,C6 BIGINT
 GENERATED ALWAYS AS (C1+c2),C7 GRAPHIC(10),C8 VARGRAPHIC(20),C9 XML
```

Move the table within the same table space and drop columns C5 and C6:

```
CALL SYSPROC.ADMIN_MOVE_TABLE(
'SVALENTI',
'T1',
'',
'',
'',
'',
'',
'',
'c1 BIGINT,c2 BIGINT ,c3 CHAR(20),c4 DEC(10,2),c7 GRAPHIC(10),c8 VARGRAPHIC(20),c9 XML',
'',
'MOVE')
```

Example 5: You have a range partitioned table with two ranges defined in tablespaces TS1 and TS2. Move the table to tablespace TS3, but leave the first range in TS1.

```
CREATE TABLE "EBABANI "."T1"  (
 "I1" INTEGER ,
 "I2" INTEGER )
  DISTRIBUTE BY HASH("I1")
 PARTITION BY RANGE("I1")
 (PART "PART0" STARTING(0) ENDING(100) IN "TS1",
 PART "PART1" STARTING(101) ENDING(MAXVALUE) IN "TS2");
```

Move the T1 table from schema EBABANI to the TS3 table space. Specify the partition definitions.

```
DB2 "CALL SYSPROC.ADMIN_MOVE_TABLE
('EBABANI',
'T1',
'TS3',
'TS3',
'TS3',
'',
'',
'(I1) (STARTING 0 ENDING 100 IN TS1 INDEX IN TS1 LONG IN TS1,
 STARTING 101 ENDING MAXVALUE IN TS3 INDEX IN TS3 LONG IN TS3)',
'',
'',
'MOVE')"
```

Related information:

(artname: sout.gif) Technical article: Convert row-organized tables to column-organized tables in Db2 with BLU Acceleration

The IBM Replication Tools by Component

IBM offers two primary replication solutions: Q replication and SQL replication.

The primary components of Q replication are the Q Capture program and the Q Apply program. The primary components of SQL replication are the Capture program and Apply program. Both types of replication share the Replication Alert

Monitor tool. You can set up and administer these replication components using the Replication Center and the ASNCLP command-line program.

The following list briefly summarizes these replication components:

Q Capture program

Reads the Db2 recovery log looking for changes to Db2 source tables and translates committed source data into WebSphere® MQ messages that can be published in XML format to a subscribing application, or replicated in a compact format to the Q Apply program.

Q Apply program

Takes WebSphere MQ messages from a queue, transforms the messages into SQL statements, and updates a target table or stored procedure. Supported targets include Db2 databases or subsystems and Oracle, Sybase, Informix® and Microsoft SQL Server databases that are accessed through federated server nicknames.

Capture program

Reads the Db2 recovery log for changes made to registered source tables or views and then stages committed transactional data in relational tables called change-data (CD) tables, where they are stored until the target system is ready to copy them. SQL replication also provides Capture triggers that populate a staging table called a consistent-change-data (CCD) table with records of changes to non-Db2 source tables.

Apply program

Reads data from staging tables and makes the appropriate changes to targets. For non-Db2 data sources, the Apply program reads the CCD table through that table's nickname on the federated database and makes the appropriate changes to the target table.

Replication Alert Monitor

A utility that checks the health of the Q Capture, Q Apply, Capture, and Apply programs. It checks for situations in which a program terminates, issues a warning or error message, reaches a threshold for a specified value, or performs a certain action, and then issues notifications to an email server, pager, or the z/OS console.

Use the Replication Center to:

- Define registrations, subscriptions, publications, queue maps, alert conditions, and other objects.
- Start, stop, suspend, resume, and reinitialize the replication programs.
- Specify the timing of automated copying.
- Specify SQL enhancements to the data.
- Define relationships between the source and the target tables.

Copying schemas

The **db2move** utility and the ADMIN_COPY_SCHEMA procedure allow you to quickly make copies of a database schema. Once a model schema is established, you can use it as a template for creating new versions.

Procedure

- Use the ADMIN_COPY_SCHEMA procedure to copy a single schema within the same database.
- Use the **db2move** utility with the **-co COPY** action to copy a single schema or multiple schemas from a source database to a target database. Most database objects from the source schema are copied to the target database under the new schema.

Troubleshooting tips

Both the ADMIN_COPY_SCHEMA procedure and the **db2move** utility invoke the **LOAD** command. While the load is processing, the table spaces wherein the database target objects reside are put into backup pending state.

ADMIN_COPY_SCHEMA procedure

Using this procedure with the COPYNO option places the table spaces wherein the target object resides into backup pending state, as described in the previous note. To get the table space out of the set integrity pending state, this procedure issues a SET INTEGRITY statement. In situations where a target table object has referential constraints defined, the target table is also placed in the set integrity pending state. Because the table spaces are already in backup pending state, the attempt by the ADMIN_COPY_SCHEMA procedure to issue a SET INTEGRITY statement fails.

To resolve this situation, issue a **BACKUP DATABASE** command to get the affected table spaces out of backup pending state. Next, look at the `Statement_text` column of the error table generated by this procedure to find a list of tables in the set integrity pending state. Then issue the SET INTEGRITY statement for each of the tables listed to take each table out of the set integrity pending state.

db2move utility

This utility attempts to copy all allowable schema objects except for the following types:

- table hierarchy
- staging tables (not supported by the load utility in multiple partition database environments)
- jars (Java™ routine archives)
- nicknames
- packages
- view hierarchies
- object privileges (All new objects are created with default authorizations)
- statistics (New objects do not contain statistics information)
- index extensions (user-defined structured type related)
- user-defined structured types and their transform functions

Unsupported type errors

If an object of one of the unsupported types is detected in the source schema, an entry is logged to an error file. The error file indicates that an unsupported object type is detected. The COPY operation still succeeds; the logged entry is meant to inform you of objects not copied by this operation.

Objects not coupled with schemas
Objects that are not coupled with a schema, such as table spaces and event monitors, are not operated on during a copy schema operation. You should create them on the target database before the copy schema operation is invoked.

Replicated tables
When copying a replicated table, the new copy of the table is not enabled for replication. The table is recreated as a regular table.

Different instances
The source database must be cataloged if it does not reside in the same instance as the target database.

SCHEMA_MAP option
When using the SCHEMA_MAP option to specify a different schema name on the target database, the copy schema operation will perform only minimal parsing of the object definition statements to replace the original schema name with the new schema name. For example, any instances of the original schema that appear inside the contents of an SQL procedure are not replaced with the new schema name. Thus the copy schema operation might fail to recreate these objects. Other examples might include staging table, result table, materialized query table. You can use the DDL in the error file to manually recreate these failed objects after the copy operation completes.

Interdependencies between objects
The copy schema operation attempts to recreate objects in an order that satisfies the interdependencies between these objects. For example, if a table T1 contains a column that references a user-defined function U1, then it will recreate U1 before recreating T1. However, dependency information for procedures is not readily available in the catalogs, so when re-creating procedures, the copy schema operation will first attempt to re-create all procedures, then try to re-create those that failed again (on the assumption that if they depended on a procedure that was successfully created during the previous attempt, then during a subsequent attempt they will be re-created successfully). The operation will continually try to recreate these failed procedures as long as it is able to successfully recreate one or more during a subsequent attempt. During every attempt at recreating a procedure, an error (and DDL) is logged into the error file. You might see many entries in the error file for the same procedures, but these procedures might have even been successfully recreated during a subsequent attempt. You should query the SYSCAT.PROCEDURES table upon completion of the copy schema operation to determine if these procedures listed in the error file were successfully recreated.

For more information, see the ADMIN_COPY_SCHEMA procedure and the **db2move** utility.

Examples of schema copy by using the db2move utility

Use the **db2move** utility with the **-co COPY** action to copy one or more schemas from a source database to a target database. After a model schema is established, you can use it as a template for creating new versions.

Example 1: Using the -c COPY options
The following example of the **db2move -co COPY** options copies the schema BAR and renames it FOO from the sample database to the target database:

```
db2move sample COPY -sn BAR -co target_db  target schema_map
  "((BAR,FOO))" -u userid -p password
```

The new (target) schema objects are created by using the same object names as the objects in the source schema, but with the target schema qualifier. It is possible to create copies of tables with or without the data from the source table. The source and target databases can be on different systems.

Example 2: Specifying table space name mappings during the COPY operation

The following example shows how to specify specific table space name mappings to be used instead of the table spaces from the source system during a **db2move COPY** operation. You can specify the SYS_ANY keyword to indicate that the target table space must be chosen by using the default table space selection algorithm. In this case, the **db2move** utility chooses any available table space to be used as the target:

```
db2move sample COPY -sn BAR -co target_db  target schema_map
  "((BAR,FOO))" tablespace_map "(SYS_ANY)" -u userid -p password
```

The SYS_ANY keyword can be used for all table spaces, or you can specify specific mappings for some table spaces, and the default table space selection algorithm for the remaining:

```
db2move sample COPY -sn BAR -co target_db  target schema_map "
  ((BAR,FOO))" tablespace_map "((TS1, TS2),(TS3, TS4), SYS_ANY)"
  -u userid -p password
```

This indicates that table space TS1 is mapped to TS2, TS3 is mapped to TS4, but the remaining table spaces use a default table space selection algorithm.

Example 3: Changing the object owners after the COPY operation

You can change the owner of each new object created in the target schema after a successful COPY. The default owner of the target objects is the connect user. If this option is specified, ownership is transferred to a new owner as demonstrated:

```
db2move sample COPY -sn BAR -co target_db  target schema_map
  "((BAR,FOO))" tablespace_map "(SYS_ANY)" owner jrichards
  -u userid -p password
```

The new owner of the target objects is jrichards.

The **db2move** utility must be started on the target system if source and target schemas are found on different systems. For copying schemas from one database to another, this action requires a list of schema names to be copied from a source database, separated by commas, and a target database name.

To copy a schema, issue **db2move** from an operating system command prompt as follows:

```
db2move dbname COPY -co COPY-options
 -u userid -p password
```

db2move - Database movement tool

The **DB2MOVE** command, when used in the `EXPORT`, `IMPORT`, or `LOAD` mode, facilitates the movement of large numbers of tables between Db2 databases located on workstations. When the **DB2MOVE** command is used in the `COPY` mode, this tool facilitates the duplication of a schema.

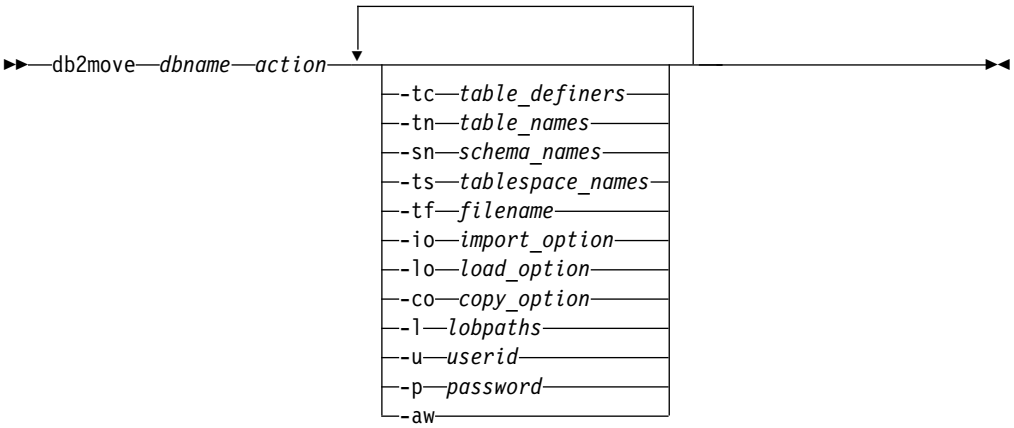
The tool queries the system catalog tables for a particular database and compiles a list of all user tables. It then exports these tables in PC/IXF format. The PC/IXF files can be imported or loaded to another local Db2 database on the same system,

or can be transferred to another workstation platform and imported or loaded to a Db2 database on that platform. Tables with structured type columns are not moved when this tool is used.

Authorization

This tool calls the Db2 export, import, and load APIs, depending on the action requested by the user. Therefore, the requesting user ID must have the authorization that the APIs require, or the request fails.

Command syntax



Command parameters

dbname
Specifies the name of the database.

action Specifies an action. Values are as follows:

EXPORT
Exports all tables that meet the filtering criteria according to the option specified. If you do not specify an option then all tables are exported. Internal staging information is stored in the `db2move.lst` file.

IMPORT
Imports all tables listed in the `db2move.lst` internal staging file. Use the **-io** option for `IMPORT` specific actions.

LOAD
Loads all tables listed in the internal staging file `db2move.lst`. Use the **-lo** option for `LOAD` specific actions.

COPY Duplicates schemas into a target database. The target database must be a local database. Use the **-sn** option to specify one or more schemas. See the **-co** option for `COPY` specific options. Use the **-tn** or **-tf** option to filter tables in LOAD_ONLY mode. You must use the SYSTOOLSPACE table space if you use the ADMIN_COPY_SCHEMA() stored procedure or if you use the **db2move** command with the **COPY** parameter.

-tc ***table_definers***
Specifies one or more table definers (creators).

This parameter applies only to the **EXPORT** action. If you specify the **-tc** parameter, only those tables that were created by the specified definers are exported. If you do not specify this parameter, all definers are used. If you specify multiple definers, you must separate them with commas; no blanks are allowed between definer IDs. You can use this parameter with the **-tn** *table_names* parameter to select the tables for export.

You can use an asterisk (*) as a wildcard character anywhere in the string.

-tn *table_names*
Specifies one or more table names. This parameter applies only to the **EXPORT** and **COPY** actions.

If you specify the **-tn** parameter with the **EXPORT** action, only those tables whose names match those in the specified string are exported. If you do not specify this parameter, all user tables are used. If you specify multiple table names, you must separate them with commas; no blanks are allowed between table names. Table names must be listed unqualified. To filter schemas, you should use the **-sn** parameter.

For export, you can use an asterisk (*) as a wildcard character anywhere in the string.

If you specify the **-tn** parameter with the **COPY** action, you must also specify the **-co "MODE" LOAD_ONLY** *copy_option* parameter, and only the specified tables are repopulated in the target database. The table names must be listed with their schema qualifiers in the format "*schema*"."*table*".

-sn *schema_names*
Specifies one or more schema names. If you specify this parameter, only those tables whose schema names match those in the specified string are exported or copied. The default for the **EXPORT** action is all schemas. The default does not apply to the **COPY** action.

If you specify multiple schema names, you must separate them with commas; no blanks are allowed between schema names. Schema names of fewer than 8 characters are padded to 8 characters in length.

In the case of the **EXPORT** action, if you use the asterisk (*) wildcard character in the schema names, it is changed to a percent sign (%), and the table name (with the percent sign) is used in the LIKE predicate of the WHERE clause. If you use the **-sn** parameter with the **-tn** or **-tc** parameter, the **db2move** command acts on only those tables whose schemas match the specified schema names or whose definers match the specified definers. A schema name `fred` has to be specified as `-sn fr*d*` instead of `-sn fr*d` when using an asterisk.

Note: The **-sn** option is not supported on Db2 for z/OS.

-ts *tablespace_names*
Specifies a list of table space names. This parameter applies only to the **EXPORT** action.

If you specify the **-ts** parameter, only those tables in the specified table space are exported. If you use the asterisk (*) wildcard character in the table space name, it is changed to a percent sign (%), and the table name (with the percent sign) is used in the LIKE predicate in the WHERE clause. If you do not specify the **-ts** parameter, all table spaces are used. If you specify multiple table space names, you must separate them with commas; no blanks are allowed between table space names. Table space names with

fewer than 8 characters are padded to 8 characters in length. To specify a table space name `mytb`, it has to be specified as`-ts my*b*` instead of `-sn my*b` when using an asterisk.

-tf *filename*

Specifies a file name. This parameter applies only to the **EXPORT** and **COPY** actions. If you specify the **-tf** parameter with the **EXPORT** action, only those tables whose names match those in the specified file are exported. In the file, you should list one table per line, and you should fully qualify each table name. Wildcard characters are not allowed in the strings. Sample file contents are as follows:

```
"SCHEMA1"."TABLE NAME1"
"SCHEMA NAME77"."TABLE155"
```

If you do not specify the **-tf** parameter, all user tables are used.

If you specify this parameter with the **COPY** action, you must also specify the **-co "MODE" LOAD_ONLY** *copy_option* parameter, and only those tables that you specify in the file are repopulated in the target database. In the file, you should list the table names with their schema qualifier in the format "*schema*"."*table*".

-io *import_option*

Specifies options for the **IMPORT** action. Valid options are `INSERT`, `INSERT_UPDATE`, `REPLACE`, `CREATE`, and `REPLACE_CREATE`. The default is `REPLACE_CREATE`. For limitations of the import create function, see "IMPORT command options CREATE and REPLACE_CREATE are deprecated" .

-lo *load_option*

Specifies options for the **LOAD** action. Valid options are `INSERT` and `REPLACE`. The default is `INSERT`.

-co Specifies options for the **COPY** action.

"TARGET_DB *db name* **[USER** *userid* **USING** *password***]"**

Specifies the name of the target database, user ID, and password. (The source database userid and password can be specified using the existing **-p** and **-u** options). The **USER USING** clause is optional. If **USER** specifies a userid, then the password must either be supplied following the **USING** clause, or if it is not specified, then **db2move** will prompt for the password information. The reason for prompting is for security reasons discussed in the following section. **TARGET_DB** is a mandatory option for the `COPY` action. The **TARGET_DB** cannot be the same as the source database and must be a local database. The ADMIN_COPY_SCHEMA procedure can be used for copying schemas within the same database. The `COPY` action requires inputting at least one schema (**-sn**) or one table (**-tn** or **-tf**).

Running multiple **db2move** commands to copy schemas from one database to another will result in deadlocks. Only one **db2move** command should be issued at a time. Changes to tables in the source schema during copy processing may mean that the data in the target schema is not identical following a copy.

"MODE"

This option is optional.

DDL_AND_LOAD
Creates all supported objects from the source schema, and populates the tables with the source table data. This is the default option.

DDL_ONLY
Creates all supported objects from the source schema, but does not repopulate the tables.

LOAD_ONLY
Loads all specified tables from the source database to the target database. The tables must already exist on the target. The `LOAD_ONLY` mode requires inputting at least one table using the **-tn** or **-tf** option.

This is an optional option that is only used with the `COPY` action.

"SCHEMA_MAP"
Renames a schema when copying to a target. This option is optional.

To use this option, provide a list of the source-target schema mappings, separated by commas, surrounded by parentheses, for example, `schema_map ((s1, t1), (s2, t2))`. In this case, objects from schema s1 are copied to schema t1 on the target, and objects from schema s2 are copied to schema t2 on the target. The default and recommended target schema name is the source schema name. The reason is that the **db2move** command does not attempt to modify the schema of any qualified objects within object bodies. Therefore, using a different target schema name might lead to problems if there are qualified objects within the object body.

Consider the following example, which creates a view called v1:`create view FOO.v1 as 'select c1 from FOO.t1'`

In this case, copy of schema FOO to BAR, v1 will be regenerated as:`create view BAR.v1 as 'select c1 from FOO.t1'`

This will either fail since schema FOO does not exist on the target database, or have an unexpected result due to FOO being different than BAR. Maintaining the same schema name as the source will avoid these issues. If there are cross dependencies between schemas, all inter-dependent schemas must be copied or there may be errors copying the objects with the cross dependencies.

For example:`create view FOO.v1 as 'select c1 from BAR.t1'`

In this case, the copy of v1 will either fail if BAR is not copied as well, or have an unexpected result if BAR on the target is different than BAR from the source. **db2move** will not attempt to detect cross schema dependencies.

This is an optional option that is only used with the `COPY` action.

If a target schema already exists, the utility will fail. Use the ADMIN_DROP_SCHEMA procedure to drop the schema and all objects associated with that schema.

"NONRECOVERABLE"
This option allows the user to override the default behavior of the load to be done with COPY-NO. With the default behavior, the user will be forced to take backups of each table space that was

loaded into. When specifying this **NONRECOVERABLE** keyword, the user will not be forced to take backups of the table spaces immediately. It is, however, highly recommended that the backups be taken as soon as possible to ensure the newly created tables will be properly recoverable. This is an optional option available to the COPY action.

"OWNER"
Allows the user to change the owner of each new object created in the target schema after a successful COPY. The default owner of the target objects will be the connect user; if this option is specified, ownership will be transferred to the new owner. This is an optional option available to the COPY action.

"TABLESPACE_MAP"
The user may specify table space name mappings to be used instead of the table spaces from the source system during a copy. This will be an array of table space mappings surrounded by brackets. For example, `tablespace_map ((TS1, TS2),(TS3, TS4))`. This would mean that all objects from table space TS1 will be copied into table space TS2 on the target database and objects from table space TS3 will be copied into table space TS4 on the target. In the case of `((T1, T2),(T2, T3))`, all objects found in T1 on the source database will be re-created in T2 on the target database and any objects found in T2 on the source database will be re-created in T3 on the target database. The default is to use the same table space name as from the source, in which case, the input mapping for this table space is not necessary. If the specified table space does not exist, the copy of the objects using that table space will fail and be logged in the error file.

The user also has the option of using the SYS_ANY keyword to indicate that the target table space should be chosen using the default table space selection algorithm. In this case, **db2move** will be able to choose any available table space to be used as the target. The SYS_ANY keyword can be used for all table spaces, example: `tablespace_map SYS_ANY`. In addition, the user can specify specific mappings for some table spaces, and the default table space selection algorithm for the remaining. For example, `tablespace_map ((TS1, TS2),(TS3, TS4), SYS_ANY)`. This indicates that table space TS1 is mapped to TS2, TS3 is mapped to TS4, but the remaining table spaces will be using a default table space target. The SYS_ANY keyword is being used since it's not possible to have a table space starting with "SYS".

This is an optional option available to the COPY action.

"PARALLEL" ***number_of_threads***
Specify this option to have the load operations for the tables in the schema(s) spread across a number of threads. The value range for *number_of_threads* is 0-16

- If PARALLEL is not specified, no threads are used and the load operations are performed serially.
- If PARALLEL is specified without a number of threads, the **db2move** utility will choose an appropriate value.
- If PARALLEL is specified and *number_of_threads* is provided, the specified number of threads is used. If *number_of_threads* is 0 or 1, the load operation is performed serially.

- The maximum value that can be specified for *number_of_threads* is 16.

This is an optional option available to the COPY action.

-l ***lobpaths***

For IMPORT and EXPORT, if this option is specified, it will be also used for XML paths. The default is the current directory.

This option specifies the absolute path names where LOB or XML files are created (as part of EXPORT) or searched for (as part of IMPORT or LOAD). When specifying multiple paths, each must be separated by commas; no blanks are allowed between paths. If multiple paths are specified, EXPORT will use them in round-robin fashion. It will write one LOB document to the first path, one to the second path, and so on up to the last, then back to the first path. The same is true for XML documents. If files are not found in the first path (during IMPORT or LOAD), the second path will be used, and so on.

-u ***userid***

The default is the logged on user ID.

Both user ID and password are optional. However, if one is specified, the other must be specified. If the command is run on a client connecting to a remote server, user ID and password should be specified.

-p ***password***

The default is the logged on password. Both user ID and password are optional. However, if one is specified, the other must be specified. When the **-p** option is specified, but the password not supplied, **db2move** will prompt for the password. This is done for security reasons. Inputting the password through command line creates security issues. For example, a **ps -ef** command would display the password. If, however, **db2move** is invoked through a script, then the passwords will have to be supplied. If the command is issued on a client connecting to a remote server, user ID and password should be specified.

-aw Allow Warnings. When **-aw** is not specified, tables that experience warnings during export are not included in the db2move.lst file (although that table's .ixf file and .msg file are still generated). In some scenarios (such as data truncation) the user might want to allow such tables to be included in the db2move.lst file. Specifying this option allows tables which receive warnings during export to be included in the .lst file.

Examples

- To export all tables in the SAMPLE database (using default values for all options), issue:

```
db2move sample export
```

- To export all tables created by userid1 or user IDs LIKE us%rid2, and with the name tbname1 or table names LIKE %tbname2, issue:

```
db2move sample export -tc userid1,us*rid2 -tn tbname1,*tbname2
```

- To import all tables in the SAMPLE database (LOB paths D:\LOBPATH1 and C:\LOBPATH2 are to be searched for LOB files; this example is applicable to Windows operating systems only), issue:

```
db2move sample import -l D:\LOBPATH1,C:\LOBPATH2
```

- To load all tables in the SAMPLE database (/home/userid/lobpath subdirectory and the tmp subdirectory are to be searched for LOB files; this example is applicable to Linux and UNIX systems only), issue:

```
db2move sample load -l /home/userid/lobpath,/tmp
```

- To import all tables in the SAMPLE database in REPLACE mode using the specified user ID and password, issue:

```
db2move sample import -io replace -u userid -p password
```

- To duplicate schema schema1 from source database dbsrc to target database dbtgt, issue:

```
db2move dbsrc COPY -sn schema1 -co TARGET_DB dbtgt USER myuser1 USING mypass1
```

- To duplicate schema schema1 from source database dbsrc to target database dbtgt, rename the schema to newschema1 on the target, and map source table space ts1 to ts2 on the target, issue:

```
db2move dbsrc COPY -sn schema1 -co TARGET_DB dbtgt USER myuser1 USING mypass1
     SCHEMA_MAP ((schema1,newschema1)) TABLESPACE_MAP ((ts1,ts2), SYS_ANY))
```

Usage notes

- When copying one or more schemas into a target database the schemas must be independent of each other. If not, some of the objects might not be copied successfully into the target database
- Loading data into tables containing XML columns is only supported for the **LOAD** and not for the **COPY** action. The workaround is to manually issue the **IMPORT** or **EXPORT** commands, or use the **db2move Export** and **db2move Import** behaviour. If these tables also contain GENERATED ALWAYS identity columns, data cannot be imported into the tables.
- A **db2move EXPORT**, followed by a **db2move IMPORT** or **db2move LOAD**, facilitates the movement of table data. It is necessary to manually move all other database objects associated with the tables (such as aliases, views, or triggers) as well as objects that these tables may depend on (such as user-defined types or user-defined functions).
- If the **IMPORT** action with the **CREATE** or **REPLACE_CREATE** option is used to create the tables on the target database (both options are deprecated and may be removed in a future release), then the limitations outlined in "Imported table re-creation" are imposed. If unexpected errors are encountered during the **db2move** import phase when the **REPLACE_CREATE** option is used, examine the appropriate tabnnn.msg message file and consider whether the errors might be the result of the limitations on table creation.
- Tables that contain GENERATED ALWAYS identity columns cannot be imported or loaded using **db2move**. You can, however, manually import or load these tables. For more information, see "Identity column load considerations" or "Identity column import considerations".
- When export, import, or load APIs are called by **db2move**, the **FileTypeMod** parameter is set to lobsinfile. That is, LOB data is kept in files that are separate from the PC/IXF file, for every table.
- The **LOAD** command must be run locally on the machine where the database and the data file reside.
- When using **db2move LOAD** and **logarchmeth1** is enabled for the database (the database is recoverable):
  - If the **NONRECOVERABLE** option is not specified, then **db2move** will invoke the db2Load API using the default **COPY NO** option, and the table spaces where the loaded tables reside are placed in the Backup Pending state upon completion of the utility (a full database or table space backup is required to take the table spaces out of the Backup Pending state).
  - If the **NONRECOVERABLE** option is specified, the table spaces are not placed in backup-pending state, however if rollforward recovery is performed later, the

table is marked inaccessible and it must be dropped. For more information aboutLoad recoverability options, see "Options for improving load performance".

- Performance for the **db2move** command with the **IMPORT** or **LOAD** actions can be improved by altering the default buffer pool, IBMDEFAULTBP, and by updating the configuration parameters **sortheap**, **util_heap_sz**, **logfilsiz**, and **logprimary**.
- When running data movement utilities such as **export** and **db2move**, the query compiler might determine that the underlying query will run more efficiently against an MQT than the base table or tables. In this case, the query will execute against a refresh deferred MQT, and the result of the utilities might not accurately represent the data in the underlying table.
- The **db2move** command is not available with Db2 clients. If you issue the **db2move** command from a client machine, you will receive a `db2move is not recognized as an internal or external command, operable program or batch file` error message. To avoid this issue, you can issue the **db2move** command directly on the server.
- The **db2move COPY** command and the ADMIN_COPY_SCHEMA procedure perform similar tasks. The ADMIN_COPY_SCHEMA procedure copies schemas within the same database, and the **db2copy COPY** command copies from one database to another. Many of the usage notes, behaviors, and restrictions that are covered in ADMIN_COPY_SCHEMA procedure - Copy a specific schema and its objects, also apply to the **db2copy COPY** command.
- Row and Column Access Control (RCAC) applies for any SQL access to tables protected with row permissions and column masks. RCAC includes SQL in applications and utilities like **IMPORT** and **EXPORT**. For example, when exporting data from a table that is protected with row permissions and column masks that use the **EXPORT** utility, only the data that you are authorized to access are exported. If your intent is to export the full content of the table, you need to make sure the SECADM grants you the proper authorization.

Files Required/Generated When Using EXPORT

- Input: None.
- Output:

  EXPORT.out
  The summarized result of the `EXPORT` action.

  db2move.lst
  The list of original table names, their corresponding PC/IXF file names (tabnnn.ixf), and message file names (tabnnn.msg). This list, the exported PC/IXF files, and LOB files (tabnnnc.yyy) are used as input to the **db2move** `IMPORT` or `LOAD` action.

  tab*nnn*.ixf
  The exported PC/IXF file of a specific table.

  tab*nnn*.msg
  The export message file of the corresponding table.

  tab*nnnc.yyy*
  The exported LOB files of a specific table.

  nnn is the table number. *c* is a letter of the alphabet. *yyy* is a number ranging from `001` to `999`.

These files are created only if the table being exported contains LOB data. If created, these LOB files are placed in the *lobpath* directories. There are a total of 26,000 possible names for the LOB files.

system.msg
The message file containing system messages for creating or deleting file or directory commands. This is only used if the action is `EXPORT`, and a LOB path is specified.

Files Required/Generated When Using IMPORT

- Input:

  db2move.lst
  An output file from the `EXPORT` action.

  tab*nnn*.ixf
  An output file from the `EXPORT` action.

  tab*nnnc.yyy*
  An output file from the `EXPORT` action.

- Output:

  IMPORT.out
  The summarized result of the `IMPORT` action.

  tab*nnn*.msg
  The import message file of the corresponding table.

Files Required/Generated When Using LOAD

- Input:

  db2move.lst
  An output file from the `EXPORT` action.

  tab*nnn*.ixf
  An output file from the `EXPORT` action.

  tab*nnnc.yyy*
  An output file from the `EXPORT` action.

- Output:

  LOAD.out
  The summarized result of the `LOAD` action.

  tab*nnn*.msg
  The **`LOAD`** message file of the corresponding table.

Files Required/Generated When Using COPY

- Input: None
- Output:

  COPYSCHEMA.msg
  An output file containing messages generated during the `COPY` operation.

  COPYSCHEMA.err
  An output file containing an error message for each error encountered during the `COPY` operation, including DDL statements for each object which could not be re-created on the target database.

LOADTABLE.msg
An output file containing messages generated by each invocation of the Load utility (used to repopulate data on the target database).

LOADTABLE.err
An output file containing the names of tables that either encountered a failure during Load or still need to be populated on the target database. See the "Restarting a failed copy schema operation" topic for more details.

These files are timestamped and all files that are generated from one run will have the same timestamp.

Performing a redirected restore using an automatically generated script

When you perform a redirected restore operation, you must specify the locations of physical containers that are stored in the backup image and provide the complete set of containers for each table space that you are altering.

Before you begin

You can perform a redirected restore only if the database was previously backed up using the Db2 backup utility.

About this task

- If the database exists, you must be able to connect to it in order to generate the script. Therefore, if the database requires an upgrade or crash recovery, this must be done before you attempt to generate a redirected restore script.
- If you are working in a partitioned database environment, and the target database does not exist, you cannot run the command to generate the redirected restore script concurrently on all database partitions. Instead, the command to generate the redirected restore script must be run one database partition at a time, starting from the catalog partition.

  Alternatively, you can first create a dummy database with the same name as your target database. After the dummy database is created, you can then generate the redirected restore script concurrently on all database partitions.
- Even if you specify the **REPLACE EXISTING** parameter when you issue the **RESTORE DATABASE** command to generate the script, the **REPLACE EXISTING** parameter is commented out in the script.
- For security reasons, your password does not appear in the generated script. You need to enter the password manually.
- The restore script includes the storage group associations for every table space that you restore.

Procedure

To perform a redirected restore using a script:

1. Use the restore utility to generate a redirected restore script. The restore utility can be invoked through the command line processor (CLP) or the db2Restore application programming interface (API). The following is an example of the **RESTORE DATABASE** command with the **REDIRECT** parameter and the **GENERATE SCRIPT** parameter:

   ```
   db2 restore db test from /home/jseifert/backups taken at 20050304090733
         redirect generate script test_node0000.clp
   ```

This creates a redirected restore script on the client called `test_node0000.clp`.

2. Open the redirected restore script in a text editor to make any modifications that are required. You can modify:
   - Restore options
   - Automatic storage paths
   - Container layout and paths
3. Run the modified redirected restore script. For example:

```
db2 -tvf test_node0000.clp
```

RESTORE DATABASE

Restores a database that has been backed up using the Db2 backup utility. The restored database is in the same state that it was in when the backup copy was made. The **Restore Database** command can also be used to encrypt an existing database.

This utility can also perform the following services:

- Overwrite a database with a different image or restore the backup copy to a new database.
- Restore backup images in Db2 Version 11.1 that were created in Db2 Versions 9.7, 10.1, or 10.5.
  - If a database upgrade is required, it is invoked automatically at the end of the restore operation.
- If, at the time of the backup operation, the database was enabled for rollforward recovery, the database can be brought to its previous state by starting the rollforward utility after successful completion of a restore operation.
- Restore a table space level backup.
- Transport a set of table spaces, storage groups, and SQL schemas from database backup image to a database by using the **TRANSPORT** option (in Db2 Version 9.7 Fix Pack 2 and later fix packs). The **TRANSPORT** option is not supported in the Db2 pureScale environment, or in partitioned database environments.
- If the database name exists when this command is issued, it replaces and redefines all storage groups as they were at the time the backup image was produced, unless otherwise redirected by the user.

For more information about the restore operations that are supported by Db2 database systems between different operating systems and hardware platforms, see "Backup and restore operations between different operating systems and hardware platforms" in the *Data Recovery and High Availability Guide and Reference*.

Incremental images and images only capturing differences from the time of the previous capture (called a "delta image") cannot be restored when there is a difference in operating systems or word size (32-bit or 64-bit).

Following a successful restore operation from one environment to a different environment, no incremental or delta backups are allowed until a non-incremental backup is taken. (This is not a limitation following a restore operation within the same environment).

Even with a successful restore operation from one environment to a different environment, some considerations exist: packages must be rebound before use (using the **BIND** command, the **REBIND** command, or the **db2rbind** utility); SQL

procedures must be dropped and re-created; and all external libraries must be rebuilt on the new platform. (These are not considerations when restoring to the same environment).

A restore operation that is run over an existing database and existing containers reuses the same containers and table space map.

A restore operation that is run against a new database reacquires all containers and rebuilds an optimized table space map. A restore operation that is run over an existing database with one or more missing containers also reacquires all containers and rebuilds an optimized table space map.

Scope

This command only affects the node on which it is run.

You cannot restore SYSCATSPACE online.

Authorization

To restore to an existing database, one of the following authorities:

- SYSADM
- SYSCTRL
- SYSMAINT

To restore to a new database, one of the following authorities:

- SYSADM
- SYSCTRL

If a user name is specified, this user requires CONNECT authority on the database.

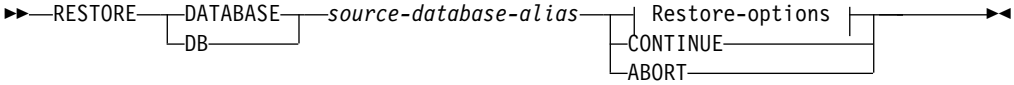
Required connection

The required connection varies based on the type of restore action:

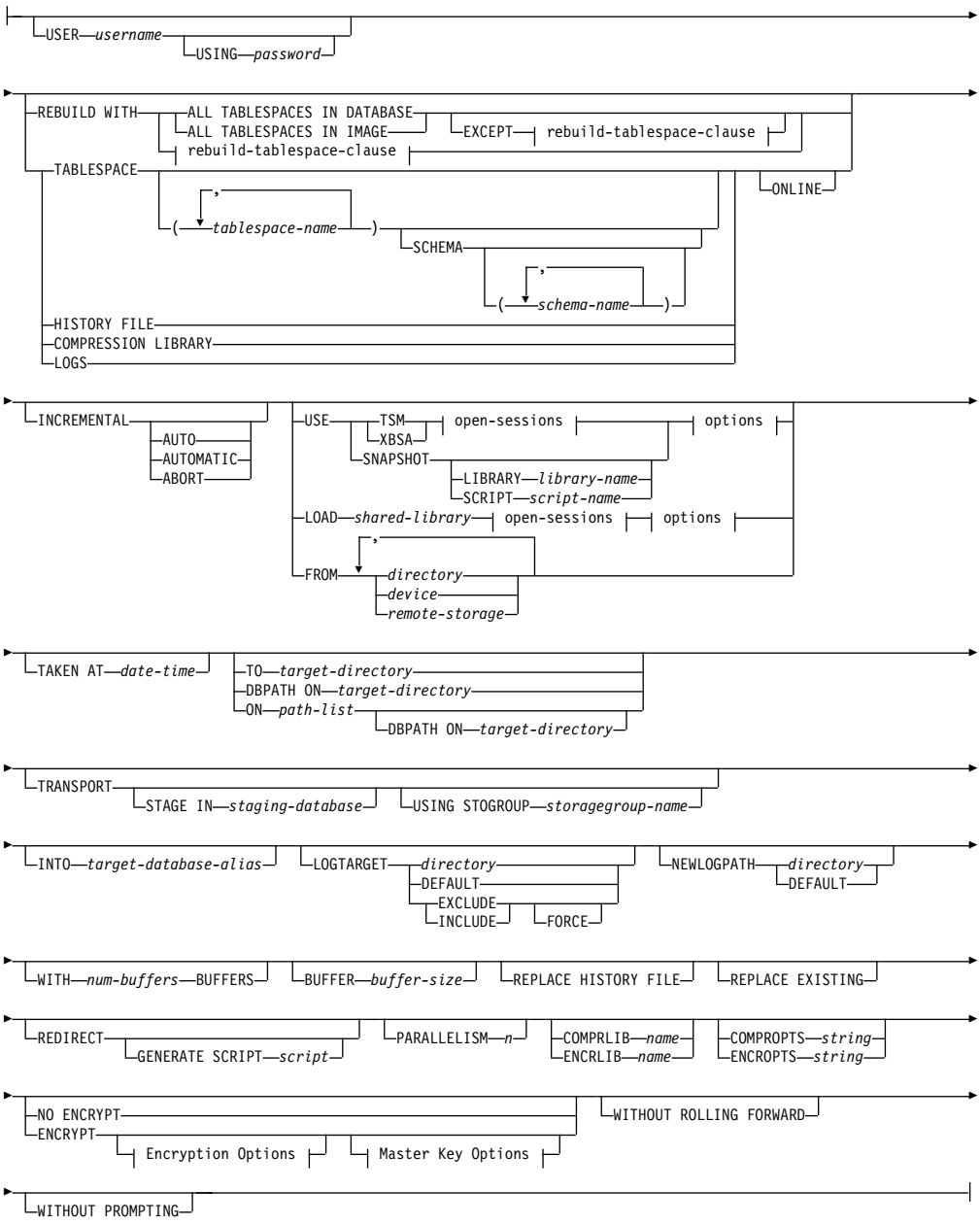
- You require a database connection to restore to an existing database. This command automatically establishes an exclusive connection to the specified database.
- You require an instance and a database connection to restore to a new database. The instance attachment is required to create the database.

  To restore to a new database at an instance different from the current instance, it is necessary to first attach to the instance where the new database resides. The new instance can be local or remote. The current instance is defined by the value of the **DB2INSTANCE** environment variable.
- For snapshot restore, *instance* and *database* connections are required.

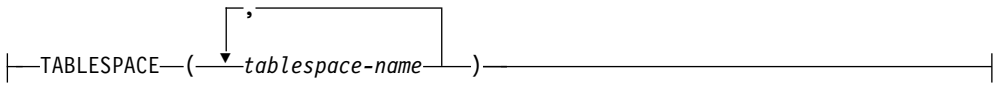
Command syntax



Restore-options:



Rebuild-tablespace-clause:

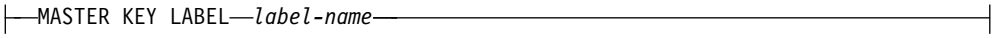


Open-sessions:

Options:

Encryption Options:

Master Key Options:



Command parameters

DATABASE *source-database-alias*
Alias of the source database from which the backup was taken.

CONTINUE
Specifies that the containers have been redefined, and that the final step in a redirected restore operation should be performed.

ABORT
This parameter:

- Stops a redirected restore operation. This is useful when an error has occurred that requires one or more steps to be repeated. After **RESTORE DATABASE** with the **ABORT** option has been issued, each step of a redirected restore operation must be repeated, including **RESTORE DATABASE** with the **REDIRECT** option.
- Terminates an incremental restore operation before completion.

USER *username*
Specifies the user name to be used when attempting a connection to the database.

USING *password*
The password that is used to authenticate the user name. If the password is omitted, the user is prompted to enter it.

REBUILD WITH ALL TABLE SPACES IN DATABASE
Restores the database with all the table spaces that are known to the database at the time of the image being restored. This restore overwrites a database if it already exists.

REBUILD WITH ALL TABLE SPACES IN DATABASE EXCEPT *rebuild-tablespace-clause*
Restores the database with all the table spaces that are known to the database at the time of the image being restored except for those specified in the list. This restore overwrites a database if it already exists.

REBUILD WITH ALL TABLE SPACES IN IMAGE
Restores the database with only the table spaces in the image being restored. This restore overwrites a database if it already exists.

REBUILD WITH ALL TABLE SPACES IN IMAGE EXCEPT *rebuild-tablespace-clause*
Restores the database with only the table spaces in the image being restored except for those specified in the list. This restore overwrites a database if it already exists.

REBUILD WITH *rebuild-tablespace-clause*
Restores the database with only the list of table spaces specified. This restore overwrites a database if it already exists.

TABLE SPACE *tablespace-name*
A list of names that are used to specify the table spaces that are to be restored.

Table space names are required when the **TRANSPORT** option is specified. This option may take as much time as a full restore operation.

SCHEMA *schema-name*
A list of names that are used to specify the schemas that are to be restored.

Schema names are required if the **TRANSPORT** option is specified. The **SCHEMA** option is only valid when the **TRANSPORT** option is specified.

ONLINE
This keyword, applicable only when performing a table space-level restore operation, is specified to allow a backup image to be restored online. This means that other agents can connect to the database while the backup image is being restored, and that the data in other table spaces is available while the specified table spaces are being restored.

HISTORY FILE
This keyword is specified to restore only the history file from the backup image.

COMPRESSION LIBRARY
This keyword is specified to restore only the compression library from the backup image. If the object exists in the backup image, it is restored into the database directory. If the object does not exist in the backup image, the restore operation fails.

LOGS This keyword is specified to restore only the set of log files that are contained in the backup image. If the backup image does not contain any log files, the restore operation fails. If this option is specified, the **LOGTARGET** option must also be specified. This option might take as much time as a full restore operation.

INCREMENTAL
Without additional parameters, **INCREMENTAL** specifies a manual cumulative restore operation. During manual restore the user must issue each restore command manually for each image that is involved in the restore. Do so according to the following order: last, first, second, third, and so on, up to and including the last image.

INCREMENTAL AUTOMATIC/AUTO
Specifies an automatic cumulative restore operation.

INCREMENTAL ABORT
Specifies abortion of an in-progress manual cumulative restore operation.

USE

TSM Specifies that the database is to be restored by using Tivoli Storage Manager (TSM) as the target device.

XBSA Specifies that the XBSA interface is to be used. Backup Services APIs (XBSA) are an open application programming interface for applications or facilities needing data storage management for backup or archiving purposes.

SNAPSHOT
Specifies that the data is to be restored from a snapshot backup.

You cannot use the **SNAPSHOT** parameter with any of the following parameters:

- **TABLESPACE**
- **INCREMENTAL**
- **TO**
- **ON**
- **DBPATH ON**
- **INTO**
- **NEWLOGPATH**
- **WITH** *num-buffers* **BUFFERS**
- **BUFFER**
- **REDIRECT**
- **REPLACE HISTORY FILE**
- **COMPRESSION LIBRARY**
- **PARALLELISM**
- **COMPRLIB**
- **OPEN** *num-sessions* **SESSIONS**
- **HISTORY FILE**
- **LOGS**

Also, you cannot use the **SNAPSHOT** parameter with any restore operation that involves a table space list, which includes the **REBUILD WITH** option.

The default behavior when you restore data from a snapshot backup image is a full database offline restore of all paths that make up the database, including all containers, the local volume directory, and the database path (DBPATH). The logs are excluded from a snapshot restore unless you specify the **LOGTARGET INCLUDE** parameter; the **LOGTARGET EXCLUDE** parameter is the default for all snapshot restores. If you provide a time stamp, the snapshot backup image with that time stamp is used for the restore.

LIBRARY *library-name*
Integrated into IBM Data Server is a Db2 ACS API driver for the following storage hardware:

- IBM TotalStorage SAN Volume Controller
- IBM Enterprise Storage Server® Model 800
- IBM Storwize® V7000
- IBM System Storage® DS6000™
- IBM System Storage DS8000®
- IBM System Storage N Series
- IBM XIV®

If you have other storage hardware, and a Db2 ACS API driver for that storage hardware, you can use the **LIBRARY** parameter to specify the Db2 ACS API driver.

The value of the **LIBRARY** parameter is a fully qualified library file name.

SCRIPT *script-name*
The name of the executable script capable of performing a snapshot restore operation. The script name must be a fully qualified file name.

OPTIONS

"*options-string*"
Specifies options to be used for the restore operation. The string is passed exactly as it was entered, without the double quotation marks.

@file-name
Specifies that the options to be used for the restore operation are contained in a file that is located on the Db2 server. The string is passed to the vendor support library. The file must be a fully qualified file name.

You cannot use the **VENDOROPT** database configuration parameter to specify vendor-specific options for snapshot restore operations. You must use the **OPTIONS** parameter of the restore utilities instead.

OPEN *num-sessions* **SESSIONS**
Specifies the number of I/O sessions that are to be used with TSM or the vendor product.

FROM *directory/device*/*remote-storage*
The fully qualified path name of the directory or device on which the backup image resides. If **USE TSM**, **FROM**, and **LOAD** are omitted, the default value is the current working directory of the client machine. This target directory or device must exist on the target server/instance.

To restore from files on remote storage, such as IBM SoftLayer® Object Storage or Amazon Simple Storage Service (S3), you can specify a remote storage location using a storage access alias. Local staging space is required to temporarily store the backup image that is transferred from the remote storage server; refer to Remote storage requirements. The syntax for specifying remote storage is:

```
DB2REMOTE://<alias>//<storage-path>/<file-name>
```

If several items are specified, and the last item is a tape device, the user is prompted for another tape. Valid response options are:

c Continue. Continue using the device that generated the warning message (for example, continue when a new tape has been mounted).

d Device terminate. Stop using *only* the device that generated the warning message (for example, terminate when there are no more tapes).

t Terminate. Abort the restore operation after the user has failed to perform some action requested by the utility.

LOAD *shared-library*
The name of the shared library (DLL on Windows operating systems)

containing the vendor backup and restore I/O functions to be used. The name can contain a full path. If the full path is not given, the value defaults to the path on which the user exit program resides.

TAKEN AT *date-time*
The time stamp of the database backup image. The time stamp is displayed after successful completion of a backup operation, and is part of the path name for the backup image. It is specified in the form *yyyymmddhhmmss*. A partial time stamp can also be specified. For example, if two different backup images with time stamps 20021001010101 and 20021002010101 exist, specifying `20021002` causes the image with time stamp `20021002010101` to be used. If a value for this parameter is not specified, there must be only one backup image on the source media.

TO *target-directory*
This parameter states the target database directory. This parameter is ignored if the utility is restoring to an existing database. The drive and directory that you specify must be local. If the backup image contains a database that is enabled for automatic storage, then only the database directory changes. The storage paths that are associated with the database do not change.

DBPATH ON *target-directory*
This parameter states the target database directory. This parameter is ignored if the utility is restoring to an existing database. The drive and directory that you specify must be local. If the backup image contains a database that is enabled for automatic storage and the parameter is not specified**ON**, then this parameter is synonymous with the **TO** parameter and only the database directory changes. The storage paths that are associated with the database do not change.

ON *path-list*

This parameter redefines the storage paths that are associated with a database. If the database contains multiple storage groups this option will redirect all storage groups to the specified paths, such that every defined storage group uses *path-list* as its new storage group paths. Using this parameter with a database that has no storage groups defined or is not enabled for automatic storage results in an error (SQL20321N). The existing storage paths as defined within the backup image are no longer used and automatic storage table spaces are automatically redirected to the new paths. If this parameter is not specified for an automatic storage database, then the storage paths remain as they are defined within the backup image. Without this parameter, while the path might not change, it is possible for the data and containers on the paths to be rebalanced during the restore. For rebalancing conditions, see .

One or more paths can be specified, each separated by a comma. Each path must have an absolute path name and it must exist locally.

If this option is specified with the **REDIRECT** option, then this option takes effect before the initial **RESTORE ... REDIRECT** command returns to the caller, and before any SET STOGROUP PATHS or SET TABLESPACE CONTAINERS statements are issued. Subsequently, if any storage group paths are redirected, those modifications override any paths specified in the initial **RESTORE ... ON** *path-list* command.

Any storage groups that have their paths redefined during a restore operation do not have any storage path-related operations replayed during a subsequent rollforward operation.

If the database does not already exist on disk and the **DBPATH ON** parameter is not specified, then the first path is used as the target database directory.

For a multi-partition database, the **ON path-list** option can only be specified on the catalog partition. The catalog partition must be restored before any other partitions are restored when the ON option is used. The restore of the catalog-partition with new storage paths places all non-catalog database partitions in a RESTORE_PENDING state. The non-catalog database partitions can then be restored in parallel without specifying the **ON** clause in the **RESTORE** command.

In general, the same storage paths must be used for each partition in a multi-partition database and they must all exist before running the **RESTORE DATABASE** command. One exception to this is where database partition expressions are used within the storage path. Doing this allows the database partition number to be reflected in the storage path such that the resulting path name is different on each partition.

Using the RESTORE command with the ON clause has the same implications as a redirected restore operation.

In an HADR environment if the primary database is defined over multiple storage paths, the RESTORE command to initialize the standby database can use the **ON path-list** option to specify these storage paths. These paths must be listed in the same order as the primary database (the order can be found through the db2pd **-db** dbname **-storagepaths** command).

You cannot use the **ON** parameter to redefine storage paths for schema transport. Schema transport will use existing storage paths on the target database.

INTO *target-database-alias*
The target database alias. If the target database does not exist, it is created.

When you restore a database backup to an existing database, the restored database inherits the alias and database name of the existing database. When you restore a database backup to a nonexistent database, the new database is created with the alias and database name that you specify. This new database name must be unique on the system where you restore it.

TRANSPORT INTO *target-database-alias*
Specifies the existing target database alias for a transport operation. The table spaces and schemas being transported are added to the database.

The **TABLESPACE** and **SCHEMA** options must specify table space names and schema names that define a valid transportable set or the transport operation fails. SQLCODE=SQL2590N rc=1.

The system catalogs cannot be transported. SQLCODE=SQL2590N rc=4.

After the schemas have been validated by the **RESTORE** command, the system catalog entries describing the objects in the table spaces being transported are created in the target database. After completion of the schema recreation, the target database takes ownership of the physical table space containers.

The physical and logical objects that are contained in the table spaces being restored are re-created in the target database and the table space definitions and containers are added to the target database. Failure during the creation of an object, or the replay of the DDL returns an error.

STAGE IN *staging-database*
Specifies the name of a temporary staging database for the backup image that is the source for the transport operation. If the **STAGE IN** option is specified, the temporary database is not dropped after the transport operation completes. The database is no longer required after the transport has completed and can be dropped by the DBA.

The following is true if the **STAGE IN** option is not specified:

- The database name is of the form SYSTG*xxx* where *xxx* is an integer value.
- The temporary staging database is dropped after the transport operation completes.

USING STOGROUP *storagegroup-name*
For automatic storage table spaces, this specifies the target storage group that will be associated with all table spaces being transported. If the storage group is not specified, then the currently designated default storage group of the target database is used. This clause only applies to automatic storage table spaces and is only valid during a schema `transport` operation.

Identifies the storage group in which table space data will be stored. *storagegroup-name* must identify a storage group that exists at the *target-database-alias* of the **TRANSPORT** operation. (SQLSTATE 42704). This is a one-part name.

LOGTARGET *directory*
Non-snapshot restores:

The absolute path name of an existing directory on the database server to be used as the target directory for extracting log files from a backup image. If this option is specified, any log files that are contained within the backup image will be extracted into the target directory. If this option is not specified, log files that are contained within a backup image will not be extracted. To extract only the log files from the backup image, specify the **LOGS** option. This option automatically appends the database partition number and a log stream ID to the path.

DEFAULT
Restore log files from the backup image into the database's default log directory, for example `/home/db2user/db2inst/NODE0000/SQL00001/LOGSTREAM0000`.

Snapshot restores:

INCLUDE
Restore log directory volumes from the snapshot image. If this option is specified and the backup image contains log directories, then they will be restored. Existing log directories and log files on disk will be left intact if they do not conflict with the log directories in the backup image. If existing log directories on disk conflict with the log directories in the backup image, then an error will be returned.

EXCLUDE
Do not restore log directory volumes. If this option is specified, then no log directories will be restored from the backup image. Existing log directories and log files on disk will be left intact if they do not conflict with the log directories in the backup image. If a path belonging to the database is restored and a log directory

will implicitly be restored because of this, thus causing a log directory to be overwritten, an error will be returned.

FORCE
Allow existing log directories in the current database to be overwritten and replaced when restoring the snapshot image. Without this option, existing log directories and log files on disk which conflict with log directories in the snapshot image will cause the restore to fail. Use this option to indicate that the restore can overwrite and replace those existing log directories.

Note: Use this option with caution, and always ensure that you have backed up and archived all logs that might be required for recovery.

For snapshot restores, the default value of the directory option is **LOGTARGET EXCLUDE**.

NEWLOGPATH *directory*
The absolute path name of a directory that will be used for active log files after the restore operation. This parameter has the same function as the **newlogpath** database configuration parameter. The parameter can be used when the log path in the backup image is not suitable for use after the restore operation; for example, when the path is no longer valid, or is being used by a different database.

Note: When the **newlogpath** command parameter is set, the node number is automatically appended to the value of **logpath** parameter. The node number is also automatically appended to the value of the **logpath** parameter when the **newlogpath** database configuration parameter is updated. For more information, see **newlogpath** - Change the database log path.

DEFAULT
After the restore completes, the database should use the default log directory: `/home/db2user/db2inst/NODE0000/SQL00001/LOGSTREAM0000` for logging.

WITH *num-buffers* **BUFFERS**
The number of buffers to be used. The Db2 database system will automatically choose an optimal value for this parameter unless you explicitly enter a value. A larger number of buffers can be used to improve performance when multiple sources are being read from, or if the value of **PARALLELISM** has been increased.

BUFFER *buffer-size*
The size, in pages, of the buffer used for the restore operation. The Db2 database system will automatically choose an optimal value for this parameter unless you explicitly enter a value. The minimum value for this parameter is eight pages.

The restore buffer size must be a positive integer multiple of the backup buffer size that is specified during the backup operation. If an incorrect buffer size is specified, the buffers are allocated to be of the smallest acceptable size.

REPLACE HISTORY FILE
Specifies that the restore operation should replace the history file on disk with the history file from the backup image.

REPLACE EXISTING

If a database with the same alias as the target database alias already exists, this parameter specifies that the restore utility is to replace the existing database with the restored database. This is useful for scripts that invoke the restore utility because the command line processor will not prompt the user to verify deletion of an existing database. If the **WITHOUT PROMPTING** parameter is specified, it is not necessary to specify **REPLACE EXISTING**, but in this case, the operation will fail if events occur that normally require user intervention.

REDIRECT

Specifies a redirected restore operation. To complete a redirected restore operation, this command should be followed by one or more **SET TABLESPACE CONTAINERS** commands or **SET STOGROUP PATHS** commands, and then by a **RESTORE DATABASE** command with the **CONTINUE** option. For example:

```
RESTORE DB SAMPLE REDIRECT

SET STOGROUP PATHS FOR sg_hot ON '/ssd/fs1', '/ssd/fs2'
SET STOGROUP PATHS FOR sg_cold ON '/hdd/path1', '/hdd/path2'

RESTORE DB SAMPLE CONTINUE
```

If a storage group has been renamed since the backup image was produced, the storage group name that is specified on the **SET STOGROUP PATHS** command refers to the storage group name from the backup image, not the most recent name.

All commands that are associated with a single redirected restore operation must be invoked from the same window or CLP session.

GENERATE SCRIPT *script*

Creates a redirect restore script with the specified file name. The script name can be relative or absolute and the script will be generated on the client side. If the file cannot be created on the client side, an error message (SQL9304N) will be returned. If the file already exists, it will be overwritten. For more information, see the following examples.

WITHOUT ROLLING FORWARD

Specifies that the database is not to be put in rollforward pending state after it has been successfully restored.

If, following a successful restore operation, the database is in rollforward pending state, the **ROLLFORWARD** command must be invoked before the database can be used again.

If this option is specified when restoring from an online backup image, error SQL2537N will be returned.

If the backup image is of a recoverable database, then **WITHOUT ROLLING FORWARD** cannot be specified with **REBUILD** option.

PARALLELISM *n*

Specifies the number of buffer manipulators that are to be created during the restore operation. The Db2 database system will automatically choose an optimal value for this parameter unless you explicitly enter a value.

COMPRLIB | ENCRLIB *name*

Indicates the name of the library that is used to decompress or decrypt a backup image. The path to the following libraries is `$HOME/sqllib/lib`.

- Encryption libraries: `libdb2encr.so` (for Linux or UNIX based operating systems); `libdb2encr.a` (for AIX®); and `db2encr.dll` (for Windows operating systems)
- Compression library: `libdb2compr.so` (for Linux or UNIX based operating systems); `libdb2compr.a` (for AIX); and `db2compr.dll` (for Windows operating systems)
- Encryption and compression libraries: `libdb2compr_encr.so` (for Linux or UNIX based operating systems); `libdb2compr_encr.a` (for AIX); and `db2compr_encr.dll` (for Windows operating systems)

The name must be a fully qualified path that refers to a file on the server. If this parameter is not specified, the Db2 database system attempts to use the library that is stored in the image. If the backup image is not compressed or encrypted, the value of this parameter is ignored. If the specified library cannot be loaded, the operation fails.

COMPROPTS | ENCROPTS *string*

Describes a block of binary data that is passed to the initialization routine in the decompression or decryption library. The Db2 database system passes this string directly from the client to the server. Any byte reversal or code page conversion issues are handled by the library. If the first character of the data block is "@", the remainder of the data is interpreted by the Db2 database system as the name of a file that is found on the server. The Db2 database system then replaces the contents of the data block with the contents of this file and passes the new value to the initialization routine instead. The maximum length for the string is 1024 bytes.

For the default Db2 libraries `libdb2compr_encr.so` (compression and encryption) or `libdb2encr.so` (encryption only), the format of the **ENCROPTS** variable is as follows:

```
Master Key Label=label-name
```

The master key label is optional. If no master key label is specified, the database manager looks in the keystore for a master key label that was used to create the backup image. If you are using other libraries, the format of the **ENCROPTS** variable depends on those libraries.

NO ENCRYPT

Specifies that an encrypted database is to be restored into a non-encrypted new or existing database. This option does not work on table space restore unless schema transport is specified with table space restore and the target database is not encrypted.

ENCRYPT

Specifies that the restored database is to be encrypted. Encryption includes all system, user, and temporary table spaces, indexes, and all transaction log data. All data types within those table spaces are encrypted, including long field data, LOBs, and XML data. You cannot specify this option when restoring into an existing database; for table space-level restore operations; when the `TRANSPORT` option is specified; or when the `USE SNAPSHOT` option is specified.

CIPHER

Specifies the encryption algorithm that is to be used for encrypting the database. You can choose one of the following FIPS 140-2 approved options:

AES Advanced Encryption Standard (AES) algorithm. This is the default.

3DES Triple Data Encryption Standard (3DES) algorithm.

MODE CBC
Specifies the encryption algorithm mode that is to be used for encrypting the database. CBC (Cipher Block Chaining) is the default mode.

KEY LENGTH *key-length*
Specifies the length of the key that is to be used for encrypting the database. The length can be one of the following values, which are specified in bits:

128 Available with AES only

168 Available with 3DES only

192 Available with AES only

256 Available with AES only

MASTER KEY LABEL
Specifies a label for the master key that is used to protect the key that is used to encrypt the database. The encryption algorithm that is used for encrypting with the master key is always AES. If the master key is automatically generated by the Db2 data server, it is always a 256-bit key.

label-name
Uniquely identifies the master key within the keystore that is identified by the value of the **keystore_type** database manager configuration parameter. The maximum length of *label-name* is 255 bytes.

WITHOUT PROMPTING
Specifies that the restore operation is to run unattended. Actions that normally require user intervention will return an error message. When using a removable media device, such as tape or diskette, the user is prompted when the device ends, even if this option is specified.

Examples

1. In the following example, the database WSDB is defined on all 4 database partitions, numbered 0 - 3. The path `/dev3/backup` is accessible from all database partitions. The following offline backup images are available from `/dev3/backup`:

```
wsdb.0.db2inst1.DBPART000.200802241234.001
wsdb.0.db2inst1.DBPART001.200802241234.001
wsdb.0.db2inst1.DBPART002.200802241234.001
wsdb.0.db2inst1.DBPART003.200802241234.001
```

To restore the catalog partition first, then all other database partitions of the WSDB database from the `/dev3/backup` directory, issue the following commands from one of the database partitions:

```
db2_all '<<+0< db2 RESTORE DATABASE wsdb FROM /dev3/backup
TAKEN AT 200802241234
  INTO wsdb REPLACE EXISTING'
db2_all '<<+1< db2 RESTORE DATABASE wsdb FROM /dev3/backup
TAKEN AT 200802241234
  INTO wsdb REPLACE EXISTING'
db2_all '<<+2< db2 RESTORE DATABASE wsdb FROM /dev3/backup
```

```
TAKEN AT 200802241234
  INTO wsdb REPLACE EXISTING'
db2_all '<<+3< db2 RESTORE DATABASE wsdb FROM /dev3/backup
TAKEN AT 200802241234
  INTO wsdb REPLACE EXISTING'
```

The **db2_all** utility issues the restore command to each specified database partition. When performing a restore using db2_all, you should always specify **REPLACE EXISTING** and/or **WITHOUT PROMPTING**. Otherwise, if there is prompting, the operation will look like it is hanging. This is because **db2_all** does not support user prompting.

2. Following is a typical redirected restore scenario for a database whose alias is MYDB:
   a. Issue a **RESTORE DATABASE** command with the **REDIRECT** option.

      ```
      restore db mydb replace existing redirect
      ```

      After successful completion of step 1, and before completing step 3, the restore operation can be aborted by issuing:

      ```
      restore db mydb abort
      ```

   b. Issue a **SET TABLESPACE CONTAINERS** command for each table space whose containers must be redefined. For example:

      ```
      set tablespace containers for 5 using
         (file 'f:\ts3con1' 20000, file 'f:\ts3con2' 20000)
      ```

      To verify that the containers of the restored database are the ones specified in this step, issue the **LIST TABLESPACE CONTAINERS** command.

   c. After successful completion of steps 1 and 2, issue:

      ```
      restore db mydb continue
      ```

      This is the final step of the redirected restore operation.

   d. If step 3 fails, or if the restore operation has been aborted, the redirected restore can be restarted, beginning at step 1.

3. following example is a sample weekly incremental backup strategy for a recoverable database. It includes a weekly full database backup operation, a daily non-cumulative (delta) backup operation, and a mid-week cumulative (incremental) backup operation:

   ```
   (Sun) backup db mydb use tsm
   (Mon) backup db mydb online incremental delta use tsm
   (Tue) backup db mydb online incremental delta use tsm
   (Wed) backup db mydb online incremental use tsm
   (Thu) backup db mydb online incremental delta use tsm
   (Fri) backup db mydb online incremental delta use tsm
   (Sat) backup db mydb online incremental use tsm
   ```

   For an automatic database restore of the images created on Friday morning, issue:

   ```
   restore db mydb incremental automatic taken at (Fri)
   ```

   For a manual database restore of the images created on Friday morning, issue:

   ```
   restore db mydb incremental taken at (Fri)
   restore db mydb incremental taken at (Sun)
   restore db mydb incremental taken at (Wed)
   restore db mydb incremental taken at (Thu)
   restore db mydb incremental taken at (Fri)
   ```

4. To produce a backup image, which includes logs, for transportation to a remote site:

```
backup db sample online to /dev3/backup include logs
```

To restore that backup image, supply a **LOGTARGET** path and specify this path during **ROLLFORWARD**:

```
restore db sample from /dev3/backup logtarget /dev3/logs
rollforward db sample to end of logs and stop overflow log path /dev3/logs
```

5. To retrieve only the log files from a backup image that includes logs:

   ```
   restore db sample logs from /dev3/backup logtarget /dev3/logs
   ```

6. In the following example, three identical target directories are specified for a backup operation on database SAMPLE. The data will be concurrently backed up to the three target directories, and three backup images will be generated with extensions .001, .002, and .003.

   ```
   backup db sample to /dev3/backup, /dev3/backup, /dev3/backup
   ```

   To restore the backup image from the target directories, issue:

   ```
   restore db sample from /dev3/backup, /dev3/backup, /dev3/backup
   ```

7. The **USE TSM OPTIONS** keywords can be used to specify the TSM information to use for the restore operation. On Windows platforms, omit the `-fromowner` option.

   - Specifying a delimited string:

     ```
     restore db sample use TSM options '"-fromnode=bar -fromowner=dmcinnis"'
     ```

   - Specifying a fully qualified file:

     ```
     restore db sample use TSM options @/u/dmcinnis/myoptions.txt
     ```

     The file `myoptions.txt` contains the following information: `-fromnode=bar -fromowner=dmcinnis`

8. The following is a simple restore of a multi-partition automatic-storage-enabled database with new storage paths. The database was originally created with one storage path, `/myPath0`:

   - On the catalog partition issue: `restore db mydb on /myPath1,/myPath2`
   - On all non-catalog partitions issue: `restore db mydb`

9. A script output of the following command on a non-auto storage database:

   ```
   restore db sample from /home/jseifert/backups taken at 20050301100417 redirect
   generate script SAMPLE_NODE0000.clp
   ```

   would look like this:

```
-- *********************************************************************************
-- ** automatically created redirect restore script
-- *********************************************************************************
UPDATE COMMAND OPTIONS USING S ON Z ON SAMPLE_NODE0000.out V ON;
SET CLIENT ATTACH_DBPARTITIONNUM  0;
SET CLIENT CONNECT_DBPARTITIONNUM 0;
-- *********************************************************************************
-- ** initialize redirected restore
-- *********************************************************************************
RESTORE DATABASE SAMPLE
-- USER  '<username>'
-- USING '<password>'
FROM '/home/jseifert/backups'
TAKEN AT 20050301100417
-- DBPATH ON '<target-directory>'
INTO SAMPLE
-- NEWLOGPATH '/home/jseifert/jseifert/SAMPLE/NODE0000/LOGSTREAM0000/'
-- WITH <num-buff> BUFFERS
-- BUFFER <buffer-size>
-- REPLACE HISTORY FILE
```

```
-- REPLACE EXISTING
REDIRECT
-- PARALLELISM <n>
-- WITHOUT ROLLING FORWARD
-- WITHOUT PROMPTING
;
-- ******************************************************************************
-- ** tablespace definition
-- ******************************************************************************
-- ******************************************************************************
-- ** Tablespace name                          = SYSCATSPACE
-- **   Tablespace ID                          = 0
-- **   Tablespace Type                        = System managed space
-- **   Tablespace Content Type                = Any data
-- **   Tablespace Page size (bytes)           = 4096
-- **   Tablespace Extent size (pages)         = 32
-- **   Using automatic storage                = No
-- **   Total number of pages                  = 5572
-- ******************************************************************************
SET TABLESPACE CONTAINERS FOR 0
-- IGNORE ROLLFORWARD CONTAINER OPERATIONS
USING (
  PATH   'SQLT0000.0'
);
-- ******************************************************************************
-- ** Tablespace name                          = TEMPSPACE1
-- **   Tablespace ID                          = 1
-- **   Tablespace Type                        = System managed space
-- **   Tablespace Content Type                = System Temporary data
-- **   Tablespace Page size (bytes)           = 4096
-- **   Tablespace Extent size (pages)         = 32
-- **   Using automatic storage                = No
-- **   Total number of pages                  = 0
-- ******************************************************************************
SET TABLESPACE CONTAINERS FOR 1
-- IGNORE ROLLFORWARD CONTAINER OPERATIONS
USING (
  PATH   'SQLT0001.0'
);
-- ******************************************************************************
-- ** Tablespace name                          = USERSPACE1
-- **   Tablespace ID                          = 2
-- **   Tablespace Type                        = System managed space
-- **   Tablespace Content Type                = Any data
-- **   Tablespace Page size (bytes)           = 4096
-- **   Tablespace Extent size (pages)         = 32
-- **   Using automatic storage                = No
-- **   Total number of pages                  = 1
-- ******************************************************************************
SET TABLESPACE CONTAINERS FOR 2
-- IGNORE ROLLFORWARD CONTAINER OPERATIONS
USING (
  PATH   'SQLT0002.0'
);
-- ******************************************************************************
-- ** Tablespace name                          = DMS
-- **   Tablespace ID                          = 3
-- **   Tablespace Type                        = Database managed space
-- **   Tablespace Content Type                = Any data
-- **   Tablespace Page size (bytes)           = 4096
-- **   Tablespace Extent size (pages)         = 32
-- **   Using automatic storage                = No
-- **   Auto-resize enabled                    = No
-- **   Total number of pages                  = 2000
-- **   Number of usable pages                 = 1960
-- **   High water mark (pages)                = 96
-- ******************************************************************************
```

```
SET TABLESPACE CONTAINERS FOR 3
-- IGNORE ROLLFORWARD CONTAINER OPERATIONS
USING (
  FILE   /tmp/dms1                                                    1000
, FILE   /tmp/dms2                                                    1000
);
-- *********************************************************************************
-- ** Tablespace name                          = RAW
-- **   Tablespace ID                          = 4
-- **   Tablespace Type                        = Database managed space
-- **   Tablespace Content Type                = Any data
-- **   Tablespace Page size (bytes)           = 4096
-- **   Tablespace Extent size (pages)         = 32
-- **   Using automatic storage                = No
-- **   Auto-resize enabled                    = No
-- **   Total number of pages                  = 2000
-- **   Number of usable pages                 = 1960
-- **   High water mark (pages)                = 96
-- *********************************************************************************
SET TABLESPACE CONTAINERS FOR 4
-- IGNORE ROLLFORWARD CONTAINER OPERATIONS
USING (
  DEVICE '/dev/hdb1'                                          1000
, DEVICE '/dev/hdb2'                                          1000
);
-- *********************************************************************************
-- ** start redirect restore
-- *********************************************************************************
RESTORE DATABASE SAMPLE CONTINUE;
-- *********************************************************************************
-- ** end of file
-- *********************************************************************************
```

10. A script output of the following command on an automatic storage database:

```
restore db test from /home/jseifert/backups taken at 20050304090733 redirect
generate script TEST_NODE0000.clp
```

would look like this:

```
-- *********************************************************************************
-- ** automatically created redirect restore script
-- *********************************************************************************
UPDATE COMMAND OPTIONS USING S ON Z ON TEST_NODE0000.out V ON;
SET CLIENT ATTACH_MEMBER  0;
SET CLIENT CONNECT_MEMBER 0;
-- *********************************************************************************
-- ** initialize redirected restore
-- *********************************************************************************
RESTORE DATABASE TEST
-- USER  '<username>'
-- USING '<password>'
FROM '/home/jseifert/backups'
TAKEN AT 20050304090733
ON '/home/jseifert'
-- DBPATH ON <target-directory>
INTO TEST
-- NEWLOGPATH '/home/jseifert/jseifert/TEST/NODE0000/LOGSTREAM0000/'
-- WITH <num-buff> BUFFERS
-- BUFFER <buffer-size>
-- REPLACE HISTORY FILE
-- REPLACE EXISTING
REDIRECT
-- PARALLELISM <n>
-- WITHOUT ROLLING FORWARD
-- WITHOUT PROMPTING
;
-- *********************************************************************************
-- ** storage group definition
-- **   Default storage group ID                = 0
-- **   Number of storage groups                = 3
-- *********************************************************************************
```

```
-- *****************************************************************************
-- ** Storage group name                          = SG_DEFAULT
-- **   Storage group ID                          = 0
-- **   Data tag                                  = None
-- *****************************************************************************
-- SET STOGROUP PATHS FOR SG_DEFAULT
-- ON '/hdd/path1'
-- ,  '/hdd/path2'
-- ;
-- *****************************************************************************
-- ** Storage group name                          = SG_HOT
-- **   Storage group ID                          = 1
-- **   Data tag                                  = 1
-- *****************************************************************************
-- SET STOGROUP PATHS FOR SG_HOT
-- ON '/ssd/fs1'
-- ,  '/ssd/fs2'
-- ;
-- *****************************************************************************
-- ** Storage group name                          = SG_COLD
-- **   Storage group ID                          = 2
-- **   Data tag                                  = 9
-- *****************************************************************************
-- SET STOGROUP PATHS FOR SG_COLD
-- ON '/hdd/slowpath1'
-- ;
-- *****************************************************************************
-- ** tablespace definition
-- *****************************************************************************
-- *****************************************************************************
-- ** Tablespace name                            = SYSCATSPACE
-- **   Tablespace ID                             = 0
-- **   Tablespace Type                           = Database managed space
-- **   Tablespace Content Type                   = Any data
-- **   Tablespace Page size (bytes)              = 4096
-- **   Tablespace Extent size (pages)            = 4
-- **   Using automatic storage                   = Yes
-- **   Storage group ID                          = 0
-- **   Source storage group ID                   = -1
-- **   Data tag                                  = None
-- **   Auto-resize enabled                       = Yes
-- **   Total number of pages                     = 6144
-- **   Number of usable pages                    = 6140
-- **   High water mark (pages)                   = 5968
-- *****************************************************************************
-- *****************************************************************************
-- ** Tablespace name                            = TEMPSPACE1
-- **   Tablespace ID                             = 1
-- **   Tablespace Type                           = System managed space
-- **   Tablespace Content Type                   = System Temporary data
-- **   Tablespace Page size (bytes)              = 4096
-- **   Tablespace Extent size (pages)            = 32
-- **   Using automatic storage                   = Yes
-- **   Total number of pages                     = 0
-- *****************************************************************************
-- *****************************************************************************
-- ** Tablespace name                            = USERSPACE1
-- **   Tablespace ID                             = 2
-- **   Tablespace Type                           = Database managed space
-- **   Tablespace Content Type                   = Any data
-- **   Tablespace Page size (bytes)              = 4096
-- **   Tablespace Extent size (pages)            = 32
-- **   Using automatic storage                   = Yes
-- **   Storage group ID                          = 1
-- **   Source storage group ID                   = -1
-- **   Data tag                                  = 1
-- **   Auto-resize enabled                       = Yes
-- **   Total number of pages                     = 256
-- **   Number of usable pages                    = 224
-- **   High water mark (pages)                   = 96
-- *****************************************************************************
-- *****************************************************************************
-- ** Tablespace name                            = DMS
-- **   Tablespace ID                             = 3
-- **   Tablespace Type                           = Database managed space
```

```
-- **   Tablespace Content Type                 = Any data
-- **   Tablespace Page size (bytes)            = 4096
-- **   Tablespace Extent size (pages)          = 32
-- **   Using automatic storage                 = No
-- **   Storage group ID                        = 2
-- **   Source storage group ID                 = -1
-- **   Data tag                                = 9
-- **   Auto-resize enabled                     = No
-- **   Total number of pages                   = 2000
-- **   Number of usable pages                  = 1960
-- **   High water mark (pages)                 = 96
-- ******************************************************************************
SET TABLESPACE CONTAINERS FOR 3
-- IGNORE ROLLFORWARD CONTAINER OPERATIONS
USING (
  FILE   '/tmp/dms1'                                        1000
, FILE   '/tmp/dms2'                                        1000
);
-- ******************************************************************************
-- ** Tablespace name                         = RAW
-- **   Tablespace ID                           = 4
-- **   Tablespace Type                         = Database managed space
-- **   Tablespace Content Type                 = Any data
-- **   Tablespace Page size (bytes)            = 4096
-- **   Tablespace Extent size (pages)          = 32
-- **   Using automatic storage                 = No
-- **   Auto-resize enabled                     = No
-- **   Total number of pages                   = 2000
-- **   Number of usable pages                  = 1960
-- **   High water mark (pages)                 = 96
-- ******************************************************************************
SET TABLESPACE CONTAINERS FOR 4
-- IGNORE ROLLFORWARD CONTAINER OPERATIONS
USING (
  DEVICE '/dev/hdb1'                                        1000
, DEVICE '/dev/hdb2'                                        1000
);
-- ******************************************************************************
-- ** start redirect restore
-- ******************************************************************************
RESTORE DATABASE TEST CONTINUE;
-- ******************************************************************************
-- ** end of file
-- ******************************************************************************
```

11. The following are examples of the **RESTORE DB** command using the **SNAPSHOT** option:

    Restore log directory volumes from the snapshot image and do not prompt.

    ```
    db2 restore db sample use snapshot LOGTARGET INCLUDE without prompting
    ```

    Do not restore log directory volumes and do not prompt.

    ```
    db2 restore db sample use snapshot LOGTARGET EXCLUDE without prompting
    ```

    Do not restore log directory volumes and do not prompt. When **LOGTARGET** is not specified, then the default is **LOGTARGET EXCLUDE**.

    ```
    db2 restore db sample use snapshot without prompting
    ```

    Allow existing log directories in the current database to be overwritten and replaced when restoring the snapshot image containing conflicting log directories, without prompting.

    ```
    db2 restore db sample use snapshot LOGTARGET EXCLUDE FORCE without prompting
    ```

    Allow existing log directories in the current database to be overwritten and replaced when restoring the snapshot image containing conflicting log directories, without prompting.

    ```
    db2 restore db sample use snapshot LOGTARGET INCLUDE FORCE without prompting
    ```

12. The following are examples of a transport operation using the **RESTORE** command with the **TRANSPORT REDIRECT** option:

Given a source database (TT_SRC) backup image, with storage paths on /src , and a target database (TT_TGT) with storage paths on /tgt :

```
> RESTORE DB TT_SRC TABLESPACE (AS1) SCHEMA (KRODGER)
     TRANSPORT INTO TT_TGT REDIRECT

SQL1277W A redirected restore operation is being performed.  Table space
configuration can now be viewed and table spaces that do not use automatic
storage can have their containers reconfigured.
DB20000I The RESTORE DATABASE command completed successfully.
```

Table space 'AS1' is transported into a container path, similar to: /tgt/krodger/NODE0000/TT_TGT/T0000003/C0000000.LRG

To specify a target storage group for the transported table spaces, the **USING STOGROUP** option of the **RESTORE** command can be used. In the following example both table spaces TS1 and TS2 will be restored into the SG_COLD storage group:

```
> RESTORE DB TT_SRC TABLESPACE (TS1, TS2) SCHEMA (KRODGER)
     TRANSPORT INTO TT_TGT USING STOGROUP SG_COLD
```

Note: The **USING STOGROUP** option of the **RESTORE** command is only valid during a transport operation, and cannot be used to specify a target storage group during any other restore operation.
To perform a transport into the default storage group of the target database, the **USING STOGROUP** option does not need to be specified:

```
> RESTORE  DB TT_SRC TABLESPACE (TS3) SCHEMA (KRODGER)
     TRANSPORT INTO TT_TGT
```

The storage group name that is specified on the **RESTORE** command during the TRANSPORT operation must currently be defined in the target database. It does not need to be defined within the backup image or source database.

13. The following examples show how to specify encryption options.

    Restore into a new encrypted database named CCARDS by using the default encryption options:

    ```
    RESTORE DATABASE ccards ENCRYPT;
    ```

    Restore into the same database by using explicitly provided encryption options to decrypt the backup image:

    ```
    RESTORE DATABASE ccards
      ENCRLIB 'libdb2encr.so'
      ENCROPTS 'Master key Label=mylabel.mydb.myinstance.myserver';
    ```

    If you cannot remember what master key label was used to protect a backup image, run the **RESTORE DATABASE** command with the SHOW MASTER KEY DETAILS encryption option; its output is the equivalent of running the ADMIN_GET_ENCRYPTION_INFO table function. The database is not restored. For example:

    ```
    RESTORE DATABASE ccards
      ENCRLIB 'libdb2encr.so'
      ENCROPTS 'show master key details'
    ```

    The command returns the label for each master key that was used to protect the backup image. The command also returns information about the location of the master key at the time that the backup was taken. This information is available in the sqllib/db2dump directory in a file whose name has the following format:

    db-name.*inst-type*.*inst-name*.
      db-partition.*timestamp*.masterKeyDetails

If the parameter **AT DBPARTITIONNUM** is used to re-create a database partition that was dropped (because it was damaged), the database at this database partition will be in the restore-pending state. After re-creating the database partition, the database must immediately be restored on this database partition.

Usage notes

- In a Db2 pureScale environment, both the **RESTORE** operation using the **REBUILD** option, as well as the ensuing database **ROLLFORWARD** operation, must be performed on a member that exists within the database member topology of every backup image involved in the operation. For example, suppose the **RESTORE REBUILD** operation uses two backup images: backup-image-A has database member topology {0,1}, and backup-image-B has database member topology {0, 1, 2, 3}. Then, both the **RESTORE** operation and the ensuing **ROLLFORWARD** operation must be performed on either member-0 or member-1 (which exist in all backup images).
- A **RESTORE DATABASE** command of the form `db2 restore db` *name* will perform a full database restore with a database image and will perform a table space restore operation of the table spaces that are found in a table space image. A **RESTORE DATABASE** command of the form `db2 restore db` *name* `tablespace` performs a table space restore of the table spaces that are found in the image. In addition, if a list of table spaces is provided with such a command, the explicitly listed table spaces are restored.
- Following the restore operation of an online backup, you must perform a rollforward recovery.
- You can use the **OPTIONS** parameter to enable restore operations in TSM environments supporting proxy nodes. For more information, see the "Configuring a Tivoli Storage Manager client" topic.
- If a backup image is compressed, the Db2 database system detects this and automatically decompresses the data before restoring it. If a library is specified on the db2Restore API, it is used for decompressing the data. Otherwise, a check is made to see if a library is stored in the backup image and if the library exists, it is used. Finally, if a library is not stored in the backup image, the data cannot be decompressed and the restore operation fails.
- If the compression library is to be restored from a backup image (either explicitly by specifying the **COMPRESSION LIBRARY** option or implicitly by performing a normal restore of a compressed backup), the restore operation must be done on the same platform and operating system that the backup was taken on. If the platform the backup was taken on is not the same as the platform that the restore is being done on, the restore operation will fail, even if the Db2 database system normally supports cross-platform restores involving the two systems.
- A backed-up SMS table space can only be restored into an SMS table space. You cannot restore it into a DMS table space, or vice versa.
- To restore log files from the backup image that contains them, the **LOGTARGET** option must be specified, providing the fully qualified and valid path that exists on the Db2 server. If those conditions are satisfied, the restore utility will write the log files from the image to the target path. If a **LOGTARGET** is specified during a restore of a backup image that does not include logs, the restore operation will return an error before attempting to restore any table space data. A restore operation will also fail with an error if an invalid, or read-only, **LOGTARGET** path is specified.
- If any log files exist in the **LOGTARGET** path at the time the **RESTORE DATABASE** command is issued, a warning prompt will be returned to the user. This warning will not be returned if **WITHOUT PROMPTING** is specified.

- During a restore operation where a **LOGTARGET** is specified, if any log file cannot be extracted, the restore operation will fail and return an error. If any of the log files being extracted from the backup image have the same name as an existing file in the **LOGTARGET** path, the restore operation will fail and an error will be returned. The restore database utility will not overwrite existing log files in the **LOGTARGET** directory.
- You can also restore only the saved log set from a backup image. To indicate that only the log files are to be restored, specify the **LOGS** option in addition to the **LOGTARGET** path. Specifying the **LOGS** option without a **LOGTARGET** path will result in an error. If any problem occurs while restoring log files in this mode of operation, the restore operation will terminate immediately and an error will be returned.
- During an automatic incremental restore operation, only the log files included in the target image of the restore operation will be retrieved from the backup image. Any log files that are included in intermediate images referenced during the incremental restore process will not be extracted from those intermediate backup images. During a manual incremental restore operation, the **LOGTARGET** path should only be specified with the final restore command to be issued.
- Offline full database backups as well as offline incremental database backups can be restored to a later database version, whereas online backups cannot. For multi-partition databases, the catalog partition must first be restored individually, followed by the remaining database partitions (in parallel or serial). However, the implicit database upgrade that is done by the restore operation can fail. In a multi-partition database, it can fail on one or more database partitions. In this case, you can follow the **RESTORE DATABASE** command with a single **UPGRADE DATABASE** command issued from the catalog partition to upgrade the database successfully.
- In a partitioned database environment, a table space can have a different storage group association on different database partitions. When a redirected restore modifies table space containers from DMS to automatic storage, the table space is associated with the default storage group. If a new default storage group is selected in between redirected restores of different database partitions, then the table space will have an inconsistent storage group association across the partitioned database environment. If this occurs, then use the ALTER TABLESPACE statement to alter the table space to use automatic storage on all database partitions, and rebalance if necessary.
- The **TRANSPORT** option is supported only when the client and database code page are equal.
- The first path that is passed in must contain the first image sequence. If a specified path contains more than one backup image sequence, they must be listed sequentially and continuously.
- For the Db2 Developer-C Edition, restoring a backup database that has a total size of all table spaces greater than the defined storage size, or restoring on an SMS table space will result in a fail.

Snapshot restore

Like a traditional (non-snapshot) restore, the default behavior when restoring a snapshot backup image will be to NOT restore the log directories —**LOGTARGET EXCLUDE**.

If the Db2 database manager detects that any log directory's group ID is shared among any of the other paths to be restored, then an error is returned. In this case, **LOGTARGET INCLUDE** or **LOGTARGET INCLUDE FORCE** must be specified, as the log directories must be part of the restore.

The Db2 database manager will make all efforts to save existing log directories (primary, mirror and overflow) before the restore of the paths from the backup image takes place.

If you want the log directories to be restored and the Db2 database manager detects that the pre-existing log directories on disk conflict with the log directories in the backup image, then the Db2 database manager will report an error. In such a case, if you have specified **LOGTARGET INCLUDE FORCE**, then this error will be suppressed and the log directories from the image will be restored, deleting whatever existed beforehand.

There is a special case in which the **LOGTARGET EXCLUDE** option is specified and a log directory path resides under the database directory (for example, /NODE*xxxx*/SQL*xxxxx*/LOGSTREAM*xxxxx*/). In this case, a restore would still overwrite the log directory as the database path, and all of the contents beneath it, would be restored. If the Db2 database manager detects this scenario and log files exist in this log directory, then an error will be reported. If you specify **LOGTARGET EXCLUDE FORCE**, then this error will be suppressed and those log directories from the backup image will overwrite the conflicting log directories on disk.

Transporting table spaces and schemas

The complete list of table spaces and schemas must be specified.

The target database must be active at the time of transport.

If an online backup image is used, then the staging database is rolled forward to the end of the backup. If an offline backup image is used, then no rollforward processing is performed.

A staging database consisting of the system catalog table space from the backup image is created under the path specified by the **dftdbpath** database parameter. This database is dropped when the **RESTORE DATABASE** command completes. The staging database is required to extract the DDL used to regenerate the objects in the table spaces being transported.

When transporting table spaces, the Db2 database manager attempts to assign the first available buffer pool of matching page size to the table space that is transported. If the target database does not have buffer pools that match the page size of the table spaces being transported, then a hidden buffer pool might be assigned. Hidden buffer pools are temporary place holders for transported table spaces. You can check the buffer pools assigned to the transported table spaces after transport completes. You can issue the **ALTER TABLESPACE** command to update buffer pools.

If database rollforward detects a table space schema transport log record, the corresponding transported table space will be taken offline and moved into drop pending state. This is because database does not have complete logs of transported table spaces to rebuild transported table spaces and their contents. You can take a full backup of the target database after transport completes, so subsequent rollforward does not pass the point of schema transport in the log stream.

The **TRANSPORT** option to transport table spaces and schemas from the database backup image to the target database is not supported if a schema being transported includes an index with an expression-based key.

Transporting storage groups

A transport operation cannot modify the currently defined storage groups of the target database, and new storage groups cannot be explicitly created during a transport.

The default target storage group of the transport is the default storage group of the target database of the operation. It is also possible to explicitly redirect all table spaces being restored during a transport operation into a specific storage group on the target database.

During a transport operation, when a **RESTORE** command using the **TRANSPORT REDIRECT** option is issued, the default storage group configuration for automatic storage table spaces is not the configuration that is contained in the backup image, but instead the storage groups and storage group paths of the target database. This is because automatic storage table spaces must be restored and redirected directly into existing storage group paths, as defined on the target database.

Db2 native encryption

When you restore a database backup image to an existing database, the encryption settings of the existing database are always preserved. If you specify the **ENCRYPT** option, an error is returned because the settings on the**RESTORE** command will not be used.

When you restore into a new database in a partitioned database environment, restore the catalog partition first, specifying the encryption options. You can then restore the other partitions without specifying the encryption optionsbecause the database already exists. When you use the **db2_all** command, target the catalog partitions first.

A backup image that was encrypted with Db2 native encryption must be restored into a database server that has Db2 native encryption available. If you want to restore into a server that is using a Db2 version that does not include Db2 native encryption, you must use an unencrypted backup image.

High availability through suspended I/O and online split mirror support

IBM Db2 server suspended I/O support enables you to split mirrored copies of your primary database without taking the database offline. You can use this to very quickly create a standby database to take over if the primary database fails.

Disk mirroring is the process of writing data to two separate hard disks at the same time. One copy of the data is called a mirror of the other. Splitting a mirror is the process of separating the two copies.

You can use disk mirroring to maintain a secondary copy of your primary database. You can use Db2 server suspended I/O functionality to split the primary and secondary mirrored copies of the database without taking the database offline. Once the primary and secondary databases copies are split, the secondary database can take over operations if the primary database fails.

If you would rather not back up a large database using the Db2 server backup utility, you can make copies from a mirrored image by using suspended I/O and the split mirror function. This approach also:

- Eliminates backup operation overhead from the production machine
- Represents a fast way to clone systems

- Represents a fast implementation of idle standby failover. There is no initial restore operation, and if a rollforward operation proves to be too slow, or encounters errors, reinitialization is very fast.

The **db2inidb** command initializes the split mirror so that it can be used:

- As a clone database
- As a standby database
- As a backup image

This command can only be issued against a split mirror, and it must be run before the split mirror can be used.

In a partitioned database environment, you do not have to suspend I/O writes on all database partitions simultaneously. You can suspend a subset of one or more database partitions to create split mirrors for performing offline backups. If the catalog partition is included in the subset, it must be the last database partition to be suspended.

In a partitioned database environment, the **db2inidb** command must be run on every database partition before the split image from any of the database partitions can be used. The tool can be run on all database partitions simultaneously using the **db2_all** command. If; however, you are using the `RELOCATE USING` option, you cannot use the **db2_all** command to run **db2inidb** on all of the database partitions simultaneously. A separate configuration file must be supplied for each database partition, that includes the NODENUM value of the database partition being changed. For example, if the name of a database is being changed, every database partition will be affected and the **db2relocatedb** command must be run with a separate configuration file on each database partition. If containers belonging to a single database partition are being moved, the **db2relocatedb** command only needs to be run once on that database partition.

Note: Ensure that the split mirror contains all containers and directories which comprise the database, including the volume directory. To gather this information, refer to the DBPATHS administrative view, which shows all the files and directories of the database that need to be split.

db2inidb - Initialize a mirrored database

Initializes a mirrored database in a split-mirror environment. The mirrored database can be initialized as a clone of the primary database, placed in rollforward pending state, or used as a backup image to restore the primary database.

If the instance that a database belongs to is changing, you must do the following to ensure that changes to the instance and database support files are made. If a database is being moved to another instance, create the new instance. The new instance must be at the same release level as the instance where the database currently resides.

You must issue this command before you can use a split-mirror database.

Authorization

one of the following authorities:

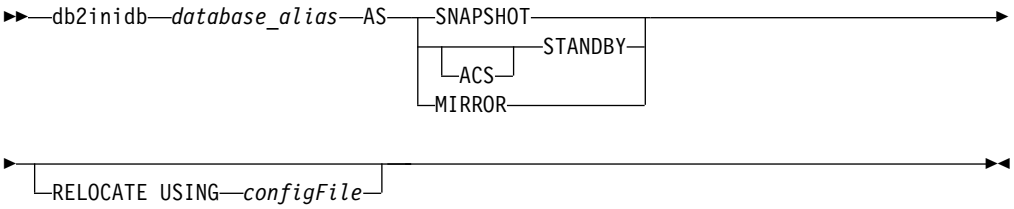
- SYSADM
- SYSCTRL

- SYSMAINT

Required connection

None

Command syntax



Command parameters

database_alias
Specifies the alias of the database to be initialized.

SNAPSHOT
Specifies that the mirrored database will be initialized as a clone of the primary database.

STANDBY
Specifies that the database will be placed in rollforward pending state. New logs from the primary database can be fetched and applied to the standby database. The standby database can then be used in place of the primary database if it goes down.

ACS
Specifies that the **db2inidb** command is to be used against an ACS snapshot copy of the database to perform the **STANDBY** action. This option is required because the **db2inidb** command can only be issued against split mirror database snapshots created with the **SET WRITE SUSPEND | RESUME** command.

Together, the use of the **ACS STANDBY** option initiates the ACS snapshot copy to be placed into a rollforward pending state so that the Db2 **BACKUP** command can be successfully issued against the snapshot image. Without this, any attempt to connect to the snapshot image results in that copy of the database being placed in the RESTORE_PENDING state, removing its usefulness as a backup copy for future recovery.

This feature was introduced specifically for interfacing with storage managers such as IBM Tivoli Storage FlashCopy® Manager, for the purpose of producing an offloaded Db2 backup that is based upon an ACS snapshot. Using this option for any other purpose, to mount or modify the contents of an ACS snapshot copy, even including the production of a Db2 backup, can lead to undefined behavior in the future.

MIRROR Specifies that the mirrored database is to be a backup image which you can use to restore the primary database.

RELOCATE USING ***configFile***
Specifies that the database files are to be relocated based on the

information listed in the *configFile* before the database is initialized as a snapshot, standby, or mirror. The format of *configFile* is described in "db2relocatedb - Relocate database" on page 240.

Usage notes

Do not issue the `db2 connect to` *database-alias* operation before issuing the `db2inidb` *database_alias* `as mirror` command. Attempting to connect to a split mirror database before initializing it erases the log files needed during roll forward recovery. The connect sets your database back to the state it was in when you suspended the database. If the database is marked as consistent when it was suspended, the Db2 database system concludes there is no need for crash recovery and empties the logs for future use. If the logs have been emptied, attempting to roll forward results in the SQL4970N error message being returned.

In partitioned database environments, the **db2inidb** command must be issued on every database partition before the split mirror from any of the database partitions can be used. **db2inidb** can be run on all database partitions simultaneously using the **db2_all** command.

If, However, if you are using the **RELOCATE USING** option, you cannot use the **db2_all** command to run **db2inidb** on all of the partitions simultaneously. A separate configuration file must be supplied for each partition, that includes the NODENUM value of the database partition being changed. For example, if the name of a database is being changed, every database partition will be affected and the **db2relocatedb** command must be run with a separate configuration file on each database partition. If containers belonging to a single database partition are being moved, the **db2relocatedb** command only needs to be run once on that database partition.

If the **RELOCATE USING** *configFile* parameter is specified and the database is relocated successfully, the specified *configFile* will be copied into the database directory and renamed to `db2path.cfg`. During a subsequent crash recovery or rollforward recovery, this file will be used to rename container paths as log files are being processed.

If a clone database is being initialized, the specified *configFile* will be automatically removed from the database directory after a crash recovery is completed.

If a standby database or mirrored database is being initialized, the specified *configFile* file is automatically removed from the database directory after a rollforward recovery is completed or canceled. New container paths can be added to the `db2path.cfg` file after **db2inidb** has been run. This would be necessary when CREATE or ALTER TABLESPACE operations are done on the original database and different paths must be used on the standby database.

When performing an initialization of a split mirror database taken from an HADR primary or standby, use the **STANDBY** parameter if one of the following apply:

- The new database is going to act in an HADR pair and the HADR configuration settings of the new pair are not identical to the settings of the original pair.
- The database is to be initialized as a stand-alone database.

In Db2 pureScale environments, you can issue the **db2inidb** command from any member and have to issue the command only once.

db2relocatedb - Relocate database

This command renames a database, or relocates a database or part of a database (for example, the container and the log directory) as specified in the configuration file provided by the user. This tool makes the necessary changes to the Db2 instance and database support files.

The target database must be offline before running the **db2relocatedb** command to modify the control files and metadata of the target database.

The changes that the **db2relocatedb** command makes to files and control structures of a database are not logged and are therefore not recoverable. A good practice is to make a full backup after running the command against a database, especially if the database is recoverable with log files being retained.

Authorization

None

Prerequisite

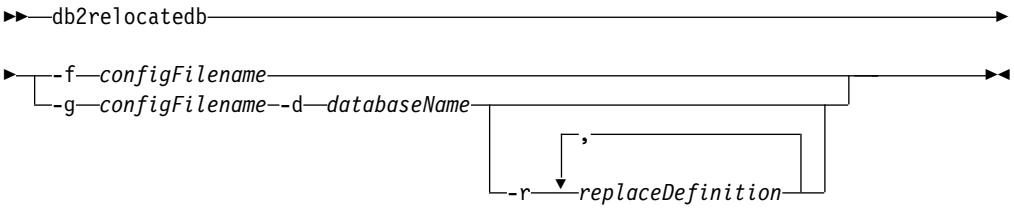
If automatic storage for the database is enabled, you must move the data from each storage path to a new location by issuing the following command:

```
$ mv old_storage_path_N/inst_name/NODE0000/X/old_storage_path_N/inst_name/NODE0000/Y
```

where *old_storage_path_N* represents the old storage path name, *inst_name* represents the instance name, *X* represents the old database name and *Y* represents the new database name.

You must perform this step to ensure that the **db2relocatedb** command executes without generating an error message.

Command syntax



Command parameters

-f *configFilename*

Specifies the name of the file containing the configuration information necessary for relocating the database. This can be a relative or absolute file name. The format of the configuration file is:

```
DB_NAME=oldName,newName
DB_PATH=oldPath,newPath
INSTANCE=oldInst,newInst
NODENUM=nodeNumber
LOG_DIR=oldDirPath,newDirPath
CONT_PATH=oldContPath1,newContPath1
CONT_PATH=oldContPath2,newContPath2
...
STORAGE_PATH=oldStoragePath1,newStoragePath1
STORAGE_PATH=oldStoragePath2,newStoragePath2
```

```
...
FAILARCHIVE_PATH=newDirPath
LOGARCHMETH1=newDirPath
LOGARCHMETH2=newDirPath
MIRRORLOG_PATH=newDirPath
OVERFLOWLOG_PATH=newDirPath
...
```

Where:

DB_NAME
Specifies the name of the database being relocated. If the database name is being changed, both the old name and the new name must be specified. This is a required field.

DB_PATH
Specifies the original path of the database being relocated. If the database path is changing, both the old path and new path must be specified. This is a required field.

INSTANCE
Specifies the instance where the database exists. If the database is being moved to a new instance, both the old instance and new instance must be specified. This is a required field.

NODENUM
Specifies the node number for the database node being changed. The default is 0.

LOG_DIR
Specifies a change in the location of the log path. If the log path is being changed, both the old path and new path must be specified. This specification is optional if the log path resides under the database path, in which case the path is updated automatically.

CONT_PATH
Specifies a change in the location of table space containers. Both the old and new container path must be specified. Multiple **CONT_PATH** lines can be provided if there are multiple container path changes to be made. This specification is optional if the container paths reside under the database path, in which case the paths are updated automatically. If you are making changes to more than one container where the same old path is being replaced by a common new path, a single **CONT_PATH** entry can be used. In such a case, an asterisk (*) could be used both in the old and new paths as a wildcard.

STORAGE_PATH
Specifies a change in the location of one of the storage paths for the database. Both the old storage path and the new storage path must be specified. Multiple **STORAGE_PATH** lines can be given if there are several storage path changes to be made. You can specify this parameter to modify any storage path in all storage groups. However, you cannot specify this parameter to modify the storage paths for an individual storage group.

Note: This parameter is not applicable to a database created with the AUTOMATIC STORAGE NO clause. Although, you can create a database specifying the AUTOMATIC STORAGE NO clause, the AUTOMATIC STORAGE clause is deprecated and might be removed from a future release.

FAILARCHIVE_PATH
Specifies a new location to archive log files if the database manager fails to archive the log files to either the primary or the secondary archive locations. You should only specify this field if the database being relocated has the **failarchpath** configuration parameter set.

LOGARCHMETH1
Specifies a new primary archive location. You should only specify this field if the database being relocated has the **logarchmeth1** configuration parameter set.

LOGARCHMETH2
Specifies a new secondary archive location. You should only specify this field if the database being relocated has the **logarchmeth2** configuration parameter set.

MIRRORLOG_PATH
Specifies a new location for the mirror log path. The string must point to a path name, and it must be a fully qualified path name, not a relative path name. You should only specify this field if the database being relocated has the **mirrorlogpath** configuration parameter set.

OVERFLOWLOG_PATH
Specifies a new location to find log files required for a rollforward operation, to store active log files retrieved from the archive, and to find and store log files required by the db2ReadLog API. You should only specify this field if the database being relocated has the **overflowlogpath** configuration parameter set.

Blank lines or lines beginning with a comment character (#) are ignored.

-g *configFilename*
Generates a configuration file and specifies the name of the file containing the configuration information. This can be a relative or absolute file name. Without the option -r, the output looks as follows:

```
DB_NAME=oldName,oldName
DB_PATH=oldPath,oldPath
INSTANCE=oldInst,oldInst
NODENUM=nodeNumber
LOG_DIR=oldDirPath,oldDirPath
CONT_PATH=oldContPath1,oldContPath1
CONT_PATH=oldContPath2,oldContPath2
...
STORAGE_PATH=oldStoragePath1,oldStoragePath1
STORAGE_PATH=oldStoragePath2,oldStoragePath2
...
FAILARCHIVE_PATH=oldDirPath
LOGARCHMETH1=oldDirPath
LOGARCHMETH2=oldDirPath
MIRRORLOG_PATH=oldDirPath
OVERFLOWLOG_PATH=oldDirPath
...
```

-d *databaseName*
Specifies the database name for which the file has to be generated.

-r *replaceDefinition*
With this option you replace strings in the generated script. Parameter *replaceDefinition* must have the format *regularExpression=replacement*. See example below.

Examples

Example 1

To change the name of the database TESTDB to PRODDB in the instance db2inst1 that resides on the path /home/db2inst1, create the following configuration file:

```
DB_NAME=TESTDB,PRODDB
DB_PATH=/home/db2inst1
INSTANCE=db2inst1
NODENUM=0
```

When the configuration file is created, you must alter any automatic storage paths to match the new database name:

```
rename /home/db2inst1/db2inst1/TESTDB /home/db2inst1/db2inst1/PRODDB
```

Save the configuration file as relocate.cfg and use the following command to make the changes to the database files:

```
db2relocatedb -f relocate.cfg
```

Example 2

To move the database DATAB1 from the instance jsmith on the path /dbpath to the instance prodinst do the following:

1. Move the files in the directory /dbpath/jsmith to /dbpath/prodinst.
2. Use the following configuration file with the **db2relocatedb** command to make the changes to the database files:

```
DB_NAME=DATAB1
DB_PATH=/dbpath
INSTANCE=jsmith,prodinst
NODENUM=0
```

Example 3

The database PRODDB exists in the instance inst1 on the path /databases/PRODDB. The location of two table space containers needs to be changed as follows:

- SMS container /data/SMS1 needs to be moved to /DATA/NewSMS1.
- DMS container /data/DMS1 needs to be moved to /DATA/DMS1.

After the physical directories and files have been moved to the new locations, the following configuration file can be used with the **db2relocatedb** command to make changes to the database files so that they recognize the new locations:

```
DB_NAME=PRODDB
DB_PATH=/databases/PRODDB
INSTANCE=inst1
NODENUM=0
CONT_PATH=/data/SMS1,/DATA/NewSMS1
CONT_PATH=/data/DMS1,/DATA/DMS1
```

Example 4

The database TESTDB exists in the instance db2inst1 and was created on the path /databases/TESTDB. Table spaces were then created with the following containers:

```
TS1
TS2_Cont0
TS2_Cont1
```

```
/databases/TESTDB/TS3_Cont0
/databases/TESTDB/TS4/Cont0
/Data/TS5_Cont0
/dev/rTS5_Cont1
```

TESTDB is to be moved to a new system. The instance on the new system will be newinst and the location of the database will be /DB2.

When moving the database, all of the files that exist in the /databases/TESTDB/db2inst1 directory must be moved to the /DB2/newinst directory. This means that the first 5 containers will be relocated as part of this move. (The first 3 are relative to the database directory and the next 2 are relative to the database path.) Since these containers are located within the database directory or database path, they do not need to be listed in the configuration file. If the 2 remaining containers are to be moved to different locations on the new system, they must be listed in the configuration file.

After the physical directories and files have been moved to their new locations, the following configuration file can be used with **db2relocatedb** to make changes to the database files so that they recognize the new locations:

```
DB_NAME=TESTDB
DB_PATH=/databases/TESTDB,/DB2
INSTANCE=db2inst1,newinst
NODENUM=0
CONT_PATH=/Data/TS5_Cont0,/DB2/TESTDB/TS5_Cont0
CONT_PATH=/dev/rTS5_Cont1,/dev/rTESTDB_TS5_Cont1
```

Example 5

The database TESTDB has two database partitions on database partition servers 10 and 20. The instance is servinst and the database path is /home/servinst on both database partition servers. The name of the database is being changed to SERVDB and the database path is being changed to /databases on both database partition servers. In addition, the log directory is being changed on database partition server 20 from /testdb_logdir to /servdb_logdir.

Since changes are being made to both database partitions, a configuration file must be created for each database partition and **db2relocatedb** must be run on each database partition server with the corresponding configuration file.

On database partition server 10, the following configuration file will be used:

```
DB_NAME=TESTDB,SERVDB
DB_PATH=/home/servinst,/databases
INSTANCE=servinst
NODENUM=10
```

On database partition server 20, the following configuration file will be used:

```
DB_NAME=TESTDB,SERVDB
DB_PATH=/home/servinst,/databases
INSTANCE=servinst
NODENUM=20
LOG_DIR=/testdb_logdir,/servdb_logdir
```

Example 6

The database MAINDB exists in the instance mainst on the path /home/mainst. The location of four table space containers needs to be changed as follows:

```
/maininst_files/allconts/C0 needs to be moved to /MAINDB/C0
/maininst_files/allconts/C1 needs to be moved to /MAINDB/C1
/maininst_files/allconts/C2 needs to be moved to /MAINDB/C2
/maininst_files/allconts/C3 needs to be moved to /MAINDB/C3
```

After the physical directories and files are moved to the new locations, the following configuration file can be used with the **db2relocatedb** command to make changes to the database files so that they recognize the new locations.

A similar change is being made to all of the containers; that is, /maininst_files/allconts/ is being replaced by /MAINDB/ so that a single entry with the wildcard character can be used:

```
DB_NAME=MAINDB
DB_PATH=/home/maininst
INSTANCE=maininst
NODENUM=0
CONT_PATH=/maininst_files/allconts/*, /MAINDB/*
```

Example 7

The database MULTIDB exists in the instance inst1 on the path /database/MULTIDB . The partitioned storage path '/home/olddbpath $N' needs to be changed to '/home/newdbpath $N'.

To be able to correctly move the partitioned storage path, the parameterized storage path need to be specified in the STORAGE_PATH field with double quotation mark around it. After the physical directories and files are moved to the new locations, the following configuration file can be used with the **db2relocatedb** command to make changes to the database files so that they recognize the new locations.

```
DB_NAME=MULTIDB
DB_PATH=/database/MULTIDB
INSTANCE=inst1
NODENUM=0
STORAGE_PATH="/home/olddbpath $N" , "/home/newdbpath $N"
```

Example 8

The database PRD exists in the instance db2prd on the database path /db2/PRD and storage paths /db2/PRD/sapdata1 and /db2/PRD/sapdata2. To generate an unmodified script use the following command that creates the output file relocate.cfg:

```
db2relocatedb -g relocate.cfg -d PRD
```

The contents of the output file relocate.cfg look as follows:

```
DB_NAME=PRD,PRD
DB_PATH=/db2/PRD,/db2/PRD
INSTANCE=db2prd,db2prd
NODENUM=0
STORAGE_PATH=/db2/PRD/sapdata1,/db2/PRD/sapdata1
STORAGE_PATH=/db2/PRD/sapdata2,/db2/PRD/sapdata2
```

If you want to relocate this database, to change the database name to QAS, to use the instance db2qas, and to change the autostorage paths accordingly, you can use the following command:

```
db2relocatedb -g relocate.cfg -d PRD -r PRD=QAS,db2prd=db2qas
```

The contents of the output file relocate.cfg look as follows:

```
DB_NAME=PRD,QAS
DB_PATH=/db2/PRD,/db2/QAS
INSTANCE=db2prd,db2qas
NODENUM=0
STORAGE_PATH=/db2/PRD/sapdata1,/db2/QAS/sapdata1
STORAGE_PATH=/db2/PRD/sapdata2,/db2/QAS/sapdata2
```

Usage notes

If the instance that a database belongs to is changing, the following must be done before running this command to ensure that changes to the instance and database support files are made:

- If a database is being moved to another instance, create the new instance. The new instance must be at the same release level as the instance where the database currently resides.
- If the new instance has a different owner than the current instance, grant access to the new instance owner.
- Copy the files and devices belonging to the databases being copied onto the system where the new instance resides. The path names must be changed as necessary. However, if there are already databases in the directory where the database files are moved to, you can mistakenly overwrite the existing `sqldbdir` file, thereby removing the references to the existing databases. In this scenario, the **db2relocatedb** utility cannot be used. Instead of **db2relocatedb**, an alternative is a redirected restore operation.
- Change the permission of the files/devices that were copied so that they are owned by the instance owner.

When moving a database from a database path where more than one database resides, the `sqldbdir` directory within that database path must be copied and not moved. This directory is still needed in the old location for Db2 to locate the databases that are not moving. After copying the `sqldbdir` directory to the new location, a `LIST DB DIRECTORY ON` *newPath* command lists databases that were not moved. These references cannot be removed and new databases with those names cannot be created on this same path. However, databases can be created with those names on a different path.

The **db2relocatedb** command cannot be used to move existing user created containers for a table space that was converted to use automatic storage using the ALTER TABLESPACE MANAGED BY AUTOMATIC STORAGE statement.

If the instance is changing, the command must be run by the new instance owner.

In a partitioned database environment, this tool must be run against every database partition that requires changes. A separate configuration file must be supplied for each database partition, that includes the `NODENUM` value of the database partition being changed. For example, if the name of a database is being changed, every database partition will be affected and the **db2relocatedb** command must be run with a separate configuration file on each database partition. If containers belonging to a single database partition are being moved, the **db2relocatedb** command only needs to be run once on that database partition.

You cannot use the **db2relocatedb** command to relocate a database that has a load in progress or is waiting for the completion of a **LOAD RESTART** or **LOAD TERMINATE** command.

After you run the **db2relocatedb** command, you must recycle the Db2 instance to allow the changes to take effect. To recycle the Db2 instance, perform the following steps:

1. Issue the **db2stop** command.
2. Issue the **db2start** command.

Limitation: In a partitioned database environment, you cannot relocate an entire node if that node is one of two or more logical partitions that reside on the same device.

db2look - Db2 statistics and DDL extraction tool

Extracts the Data Definition Language (DDL) statements that are required to reproduce the database objects of a production database on a test database.

The **db2look** command generates the DDL statements by object type. Note that this command ignores all objects under SYSTOOLS schema except user-defined functions and stored procedures.

It is often advantageous to have a test system that contains a subset of the data of a production system, but access plans selected for such a test system are not necessarily the same as those that would be selected for the production system. However, using the **db2look** tool, you can create a test system with access plans that are similar to those that would be used on the production system. You can use this tool to generate the UPDATE statements that are required to replicate the catalog statistics on the objects in a production database on a test database. You can also use this tool to generate **UPDATE DATABASE CONFIGURATION**, **UPDATE DATABASE MANAGER CONFIGURATION**, and **db2set** commands so that the values of query optimizer-related configuration parameters and registry variables on a test database match those of a production database.

You should check the DDL statements that are generated by the **db2look** command because they might not reproduce all characteristics of the original SQL objects. For table spaces on partitioned database environments, DDL might not be complete if some database partitions are not active. Make sure all database partitions are active using the **ACTIVATE DATABASE** command.

Authorization

SELECT privilege on the system catalog tables.

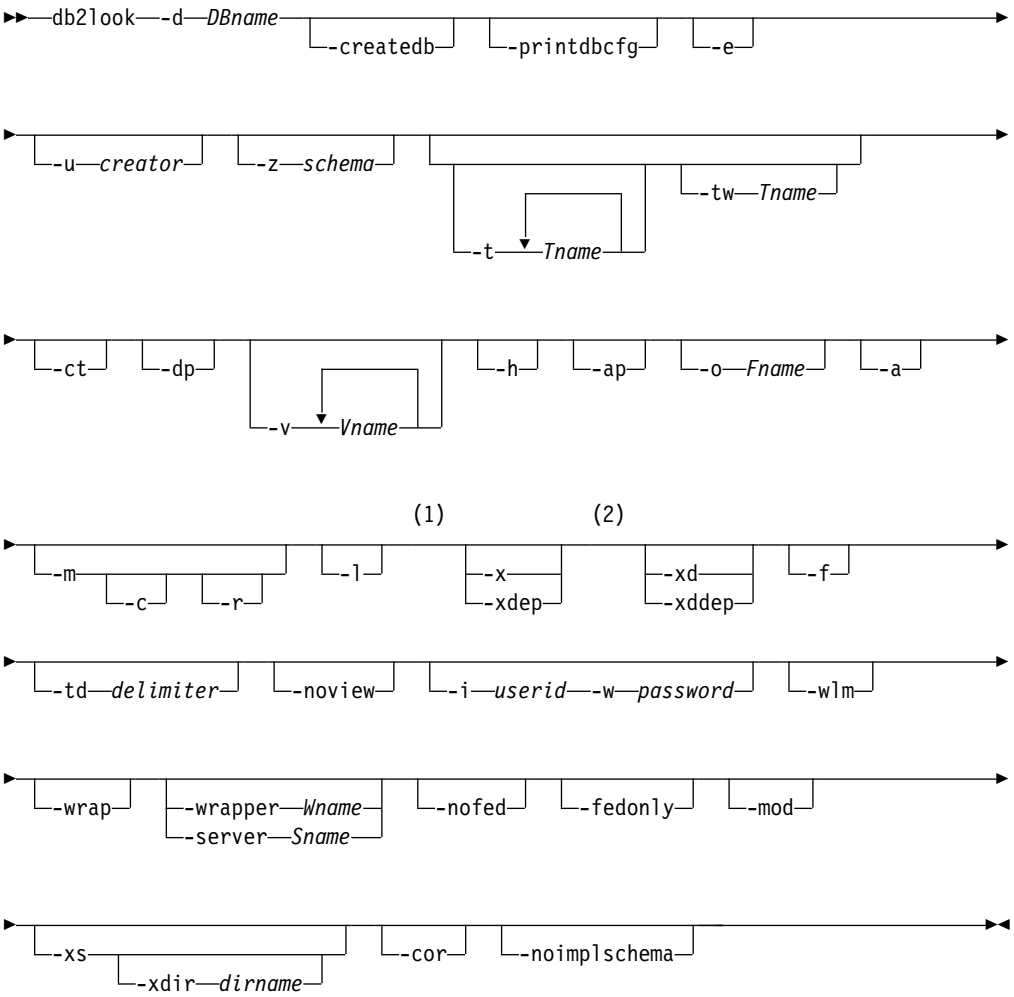
In some cases, such as generating table space container DDL, you will require one of the following authorities:

- SYSADM
- SYSCTRL
- SYSMAINT
- SYSMON
- DBADM
- EXECUTE privilege on the ADMIN_GET_STORAGE_PATHS table function

Required connection

None

Command syntax



Notes:

1 You cannot specify both the **-x** parameter and **-xdep** parameter

2 You cannot specify both the **-xd** parameter and **-xddep** parameter

Command parameters

-d *DBname*

Alias name of the production database that is to be queried. *DBname* can be the name of a Db2 or Db2 for z/OS database. If the *DBname* is a Db2 for z/OS database, the **db2look** command generates the following statements:

- DDL statements for tables, indexes, views, and user-defined distinct types
- UPDATE statistics statements for tables, columns, column distributions, and indexes

These DDL and UPDATE statistics statements are applicable to a Db2 database and not to a Db2 for z/OS database. These statements are useful if you want to extract Db2 for z/OS objects and re-create them in a Db2 database.

-createdb

Generates the **CREATE DATABASE** command that was used to create the source database.

The generated **CREATE DATABASE** command contains the usual parameters and options found in the CREATE DATABASE syntax except the following parameters:

- ALIAS
- NUMSEGS
- RESTRICTIVE
- WITH
- AUTOCONFIGURE

-printdbcfg

Generates UPDATE DB CFG commands for the database configuration parameters. The **printdbcfg** command generates UPDATE DB CFG commands in a similar order as the results returned from the GET DB CFG command.

For the parameters that support the AUTOMATIC value, you might need to add AUTOMATIC at the end of the generated UPDATE DB CFG command.

The generated **UPDATE DB CFG** command contains the usual parameters and options found in the UPDATE DATABASE CONFIGURATION syntax except for the following parameters:

- PAGE_AGE_TRGT_MCR
- DFT_TABLE_ORG
- STRING_UNITS
- NCHAR_MAPPING
- EXTENDED_ROW_SZ
- CONNECT_PROC

-e Extracts DDL statements for the following database objects:

- Aliases
- Audit policies
- Check constraints
- Function mappings
- Function templates
- Global variables
- Indexes (including partitioned indexes on partitioned tables)
- Index specifications
- Materialized query tables (MQTs)
- Nicknames
- Primary key, referential integrity, and check constraints
- Referential integrity constraints
- Roles
- Schemas
- Security labels
- Security label components
- Security policies

- Sequences
- Servers
- Stored procedures
- Tables

  Note: Values from column STATISTICS_PROFILE in the SYSIBM.SYSTABLES catalog table are not included.
- Triggers
- Trusted contexts
- Type mappings
- User mappings
- User-defined distinct types
- User-defined functions
- User-defined methods
- User-defined structured types
- User-defined transforms
- Views
- Wrappers

If you use DDL statements that are generated by the **db2look** command to re-create a user-defined function, the source code that the function references (the EXTERNAL NAME clause, for example) must be available for the function to be usable.

-u *creator*
Generates DDL statements for objects that were created with the specified creator ID. Limits output to objects that were created with the specified creator ID. The output does not include any inoperative objects. To display inoperative objects, use the **-a** parameter. If you specify the **-a** parameter, the **-u** parameter is ignored.

-z *schema*
Generates DDL statements for objects that have the specified schema name. Limits output to objects that have the specified schema name. The output does not include any inoperative objects. To display inoperative objects, use the **-a** parameter. If you do not specify the **-z** parameter, objects with all schema names are extracted. If you specify the **-a** parameter, the **-z** parameter is ignored. This parameter is also ignored for federated DDL statements.

-t *Tname1 Tname2 ... TnameN*
Generates DDL statements for the specified tables and their dependent objects. Limits output to the tables that are specified in the table list and generates the DDL statements for all dependent objects of all user specified tables. The maximum number of tables is 30.

The dependent objects include:

- Comments
- Indexes
- Primary keys
- Unique keys
- Aliases
- Foreign key constraints

- Check constraints
- Views
- Triggers

Specify the list as follows:

- Separate table names by a blank space.
- Enclose case-sensitive names and double-byte character set (DBCS) names with the backslash (\) and double quotation marks (" ") (for example, `\" MyTabLe \"`).
- Enclose multiword table names with the backslash and two sets of double quotation marks (for example, `"\"My Table\""`) to prevent the pairing from being evaluated word-by-word by the command line processor (CLP). If you use only one set of double quotation marks (for example, `"My Table"`), all words are converted into uppercase, and the **db2look** command looks for an uppercase table name (for example, MY TABLE).

If you specify the **-t** parameter with the **-l** parameter, partitioned tables are supported.

You can use two-part table names of the format *schema.table* to fully qualify a table name without using the **-z** *schema* parameter. Use a two-part table name when the table has dependent objects that are in a different schema than that of the table and you require DDL statements to be generated for the dependent objects. If you use the **-z** *schema* parameter to specify the schema, the parameter excludes dependent objects that do not have the same parent schema, thereby preventing the generation of DDL statements for the dependent objects.

-tw *Tname*
Generates DDL statements for tables with names that match the pattern that you specify with *Tname* and generates the DDL statements for all dependent objects of those tables. *Tname* must be a single value only. The underscore character (_) in *Tname* represents any single character. The percent sign (%) represents a string of zero or more characters. When **-tw** is specified, the **-t** option is ignored.

You can use two-part table names of the format *schema.table* to fully qualify a table name without using the **-z** *schema* parameter. Use a two-part table name when the table has dependent objects that are in a different schema than that of the table and you require DDL statements to be generated for the dependent objects. If you use the **-z** *schema* parameter to specify the schema, the parameter excludes dependent objects that do not have the same parent schema, thereby preventing the generation of DDL statements for the dependent objects.

-ct Generates DDL statements by object creation time. The object DDL statements might not be displayed in correct dependency order. If you specify the **-ct** parameter, the **db2look** command supports only the following additional parameters: **-e**, **-a**, **-u**, **-z**, **-t**, **-tw**, **-v**, **-l**, **-noview**, and **-wlm**. If you use the **-ct** parameter with the **-z** and **-t** parameters, the **db2look** command generates the required UPDATE statements to replicate the statistics on tables, statistical views, columns, and indexes.

-dp Generates a DROP statement before a CREATE statement. The DROP statement might not work if there is an object that depends on the dropped object. For example, you cannot drop a schema if there is a table that depends on the schema, and you cannot drop a user-defined type or

user-defined function if there is a type, function, trigger, or table that depends on that user-defined type or user-defined function. For typed tables, the DROP TABLE HIERARCHY statement is generated for the root table only. A DROP statement is not generated for indexes, primary and foreign keys, and constraints because they are always dropped when the table is dropped. You cannot drop a table that has the RESTRICT ON DROP attribute.

-v *Vname1 Vname2 ... VnameN*
Generates DDL statements for the specified views, but not for their dependent objects. The maximum number of views is 30. The rules governing case-sensitive, DBCS, and multiword table names also apply to view names. If you specify the **-t** parameter, the **-v** parameter is ignored.

You can use a two-part view name of the format *schema.view* to fully qualify a view.

-h Display help information. If you specify this parameter, all other parameters are ignored.

-ap Generates the AUDIT USING statements that are required to associate audit policies with other database objects.

-o *Fname*
Writes the output to the *Fname* file. If you do not specify an extension, the `.sql` extension is used. If you do not specify this parameter, output is written to standard output.

-a Generates DDL statements for objects that were created by any user, including inoperative objects. For example, if you specify this parameter with the **-e** parameter, DDL statements are extracted for all objects in the database. If you specify this parameter with the **-m** parameter, UPDATE statistics statements are extracted for all user-created tables and indexes in the database.

If you do not specify either the **-u** or the **-a** parameter, the USER environment variable is used. On UNIX operating systems, you do not have to explicitly set this variable. However, on Windows operating systems the USER environment variable does not have a default value. Therefore, you must set a user variable in the SYSTEM variables or issue the **set USER=***username* command for the session.

-m Generates the UPDATE statements that are required to replicate the statistics on tables, statistical views, columns, and indexes. Using the **-m** parameter is referred to as running in mimic mode.

-c If you specify this option, the **db2look** command does not generate COMMIT, CONNECT, and CONNECT RESET statements. The default action is to generate these statements. This option is ignored unless you also specify the **-m** or **-e** parameter.

-r If you specify this option with the **-m** parameter, the **db2look** command does not generate the **RUNSTATS** command. The default action is to generate the **RUNSTATS** command.

Important: If you intend to run the command processor script that is created using the **db2look** command with the **-m** parameter against another database (for example, to make the catalog statistics of the test database match those in production), both databases must use the same codeset, territory, collation, and uniqueness determination.

-l Generates DDL statements for the following database objects:

- User-defined table spaces
- User-defined storage groups
- User-defined database partition groups
- User-defined buffer pools

-x Generates authorization DDL statements such as GRANT statements.

The supported authorizations include the following ones:

- Columns: UPDATE, REFERENCES
- Databases: ACCESSCTRL, BINDADD, CONNECT, CREATETAB, CREATE_EXTERNAL_ROUTINE, CREATE_NOT_FENCED_ROUTINE, DATAACCESS, DBADM, EXPLAIN, IMPLICIT_SCHEMA, LOAD, QUIESCE_CONNECT, SECADM, SQLADM, WLMADM
- Exemptions
- Global variables
- Indexes: CONTROL
- Packages: CONTROL, BIND, EXECUTE
- Roles
- Schemas: CREATEIN, DROPIN, ALTERIN
- Security labels
- Sequences: USAGE, ALTER
- Stored procedures: EXECUTE
- Tables: ALTER, SELECT, INSERT, DELETE, UPDATE, INDEX, REFERENCE, CONTROL
- Views: SELECT, INSERT, DELETE, UPDATE, CONTROL
- User-defined functions (UDFs): EXECUTE
- User-defined methods: EXECUTE
- Table spaces: USE
- Workloads: USAGE

Note: This parameter does not generate authorization DDL statements for dependent objects when used with either the **-t** or **-tw** parameter. Use the **-xdep** parameter to generate authorization DDL statements for parent and dependent objects.

-xdep Generates authorization DDL statements, for example, GRANT statements, for parent and dependent objects. This parameter is ignored if either the **-t** or **-tw** parameter is not specified. The supported authorizations include the following ones:

- Columns: UPDATE, REFERENCES
- Indexes: CONTROL
- Stored procedures: EXECUTE
- Tables: ALTER, SELECT, INSERT, DELETE, UPDATE, INDEX, REFERENCE, CONTROL
- Table spaces: USE
- User-defined functions (UDFs): EXECUTE
- User-defined methods: EXECUTE
- Views: SELECT, INSERT, DELETE, UPDATE, CONTROL

-xd Generates authorization DDL statements, including authorization DDL

statements for objects whose authorizations were granted by SYSIBM at object creation time. It does not generate the authorization DDLs for system catalog tables and catalog views.

Note: This parameter does not generate authorization DDL statements for dependent objects when used with either the **-t** or **-tw** parameter. Use the **-xddep** parameter to generate authorization DDL statements for parent and dependent objects.

-xddep
Generates all authorization DDL statements for parent and dependent objects, including authorization DDL statements for objects whose authorizations were granted by SYSIBM at object creation time. This parameter is ignored if either the **-t** or **-tw** parameter is not specified.

-f Extracts the configuration parameters and registry variables that affect the query optimizer.

-td *delimiter*
Specifies the statement delimiter for SQL statements that are generated by the **db2look** command. The default delimiter is the semicolon (;). Use this parameter if you specify the **-e** parameter because the extracted objects might contain triggers or SQL routines.

-noview
Specifies that CREATE VIEW DDL statements will not be extracted.

-i *userid*
Specifies the user ID that the **db2look** command uses to log on to a remote system. When you specify this parameter and the **-w** parameter, the **db2look** command can run against a database on a remote system. The local and remote database must use the same Db2 for z/OS version.

-w *password*
Specifies the password that the **db2look** command uses to log on to a remote system. When you specify this parameter and the **-i** parameter, the **db2look** command can run against a database on a remote system. The local and remote database must use the same Db2 for z/OS version.

-wlm Generates WLM-specific DDL output, which can serve to generate CREATE and ALTER statements for the following items:

- Histograms
- Service classes
- Thresholds
- WLM event monitors
- Workloads
- Work action sets
- Work class sets

-wrap Generates obfuscated versions of DDL statements for routines, triggers, views, and PL/SQL packages.

-wrapper *Wname*
Generates DDL statements for federated objects that apply to the specified wrapper. The federated DDL statements that might be generated include the following ones:

- CREATE FUNCTION ... AS TEMPLATE
- CREATE FUNCTION MAPPING

- CREATE INDEX SPECIFICATION
- CREATE NICKNAME
- CREATE SERVER
- CREATE TYPE MAPPING
- CREATE USER MAPPING
- CREATE WRAPPER
- GRANT (privileges to nicknames, servers, indexes)

An error is returned if you do not specify a wrapper name or specify more than one.

-server *Sname*
Generates DDL statements for federated objects that apply to the specified server. The federated DDL statements that might be generated include the following ones:

- CREATE FUNCTION ... AS TEMPLATE
- CREATE FUNCTION MAPPING
- CREATE INDEX SPECIFICATION
- CREATE NICKNAME
- CREATE SERVER
- CREATE TYPE MAPPING
- CREATE USER MAPPING
- CREATE WRAPPER
- GRANT (privileges to nicknames, servers, indexes)

An error is returned if you do not specify a server name or specify more than one.

-nofed Specifies that no federated DDL statements will be generated. If you specify this parameter, the **-wrapper** and **-server** parameters are ignored.

-fedonly
Specifies that only federated DDL statements will be generated.

-mod Generates DDL statements for each module, and for all of the objects that are defined in each module.

-xs Exports all files that are necessary to register XML schemas and DTDs at the target database and generates appropriate commands for registering them. The set of XSR objects that is exported is controlled by the **-u**, **-z**, and **-a** parameters.

-xdir *dirname*
Exports XML-related files into the specified path. If you do not specify this parameter, all XML-related files are exported into the current directory.

-cor Generates DDL statements with the CREATE OR REPLACE clause, regardless of whether or not the statements originally contained that clause.

-noimplschema
Specifies that CREATE SCHEMA DDL statements for implicitly created schemas are not generated. If you specify this parameter, you must also specify the **-e** parameter.

Examples

The following examples show how to use the **db2look** command:

- Generate the DDL statements for objects created by user `walid` in database DEPARTMENT. The output is sent to the `db2look.sql` file.

  ```
  db2look -d department -u walid -e -o db2look.sql
  ```

- Generate the DDL statements for objects that have schema name `ianhe`, created by user `walid`, in database DEPARTMENT. The output is sent to the `db2look.sql` file.

  ```
  db2look -d department -u walid -z ianhe -e -o db2look.sql
  ```

- Generate the UPDATE statements to replicate the statistics for the database objects created by user `walid` in database DEPARTMENT. The output is sent to the `db2look.sql` file.

  ```
  db2look -d department -u walid -m -o db2look.sql
  ```

- Generate both the DDL statements for the objects created by user `walid` and the UPDATE statements to replicate the statistics on the database objects created by the same user. The output is sent to the `db2look.sql` file.

  ```
  db2look -d department -u walid -e -m -o db2look.sql
  ```

- Generate the DDL statements for objects created by all users in the database DEPARTMENT. The output is sent to the `db2look.sql` file.

  ```
  db2look -d department -a -e -o db2look.sql
  ```

- Generate the DDL statements for all user-defined database partition groups, buffer pools and table spaces. The output is sent to the `db2look.sql` file.

  ```
  db2look -d department -l -o db2look.sql
  ```

- Generate the UPDATE statements for optimizer-related database and database manager configuration parameters and the **db2set** commands for optimizer-related registry variables in database DEPARTMENT. The output is sent to the `db2look.sql` file.

  ```
  db2look -d department -f -o db2look.sql
  ```

- Generate the **db2set** commands for optimizer-related registry variables and the following statements for database DEPARTMENT:
  - The DDL statements for all database objects
  - The UPDATE statements to replicate the statistics on all tables and indexes
  - The GRANT authorization statements
  - The UPDATE statements for optimizer-related database and database manager configuration parameters
  - The **db2set** commands for optimizer-related registry variables
  - The DDL statements for all user-defined database partition groups, buffer pools, and table spaces

  The output is sent to the `db2look.sql` file.

  ```
  db2look -d department -a -e -m -l -x -f -o db2look.sql
  ```

- Generate all authorization DDL statements for all objects in database DEPARTMENT, including the objects that were created by the original creator. (In this case, the authorizations were granted by SYSIBM at object creation time.) The output is sent to the `db2look.sql` file.

  ```
  db2look -d department -xd -o db2look.sql
  ```

- Generate the DDL statements for objects created by all users in the database DEPARTMENT. The output is sent to the `db2look.sql` file.

  ```
  db2look -d department -a -e -td % -o db2look.sql
  ```

The output can then be read by the CLP:

```
db2 -td% -f db2look.sql
```

- Generate the DDL statements for objects in database DEPARTMENT, excluding the CREATE VIEW statements. The output is sent to the `db2look.sql` file.

  ```
  db2look -d department -e -noview -o db2look.sql
  ```

- Generate the DDL statements for objects in database DEPARTMENT related to specified tables. The output is sent to the `db2look.sql` file.

  ```
  db2look -d department -e -t tab1 "\"My TaBlE2\"" -o db2look.sql
  ```

- Generate the DDL statements for all objects (federated and non-federated) in the federated database FEDDEPART. For federated DDL statements, only those that apply to the specified wrapper, FEDWRAP, are generated. The output is sent to standard output.

  ```
  db2look -d feddepart -e -wrapper fedwrap
  ```

- Generate a script file that includes only non-federated DDL statements. The following system command can be run against a federated database FEDDEPART and yet only produce output like that found when run against a database which is not federated. The output is sent to the `out.sql` file.

  ```
  db2look -d feddepart -e -nofed -o out
  ```

- Generate the DDL statements for objects that have schema name `walid` in the database DEPARTMENT. The files required to register any included XML schemas and DTDs are exported to the current directory. The output is sent to the `db2look.sql` file.

  ```
  db2look -d department -z walid -e -xs -o db2look.sql
  ```

- Generate the DDL statements for objects that were created by all users in the database DEPARTMENT. The files that are required to register any included XML schemas and DTDs are exported to the `/home/ofer/ofer/` directory. The output is sent to standard output.

  ```
  db2look -d department -a -e -xs -xdir /home/ofer/ofer/
  ```

- Generate only WLM-specific DDL statements for database DEPARTMENT:

  ```
  db2look -d department -wlm
  ```

- Generate the DDL statements for all objects in database DEPARTMENT:

  ```
  db2look -d department -wlm -e -l
  ```

- Generate the DDL statements for both the parent table TAB1 in schema TABLES and the dependent view of TAB1 that is called VIEW1 in the VIEWS schema in database DB1. The output is sent to the `db2look.sql` file.

  ```
  db2look -d DB1 -t TABLES.TAB1 -e -o db2look.sql
  ```

- Generate the DDL statements and authorization DDL statements for the parent table TAB1 in schema TABLES and the dependent view of TAB1 that is called VIEW1 in the VIEWS schema in database DB1. The output is sent to the `db2look.sql` file.

  ```
  db2look -d DB1 -t TABLES.TAB1 -e -xdep -o db2look.sql
  ```

- Generate the RUNSTATS DDL statements on the TABLE1 table in mimic mode. The output is sent to the `db2look.sql` file. Not all available RUNSTATS clauses of the command are supported.

  ```
  db2look -d DB1 -t TABLE1 -m  -e -o db2look.sql
  ```

- Generate the **CREATE DATABASE** command that was used to create the database DEPARTMENT. The output is sent to the `db2look.sql` file.

  ```
  db2look -d department -createdb -o db2look.sql
  ```

- Generate the UPDATE DB CFG statements from the database DEPARTMENT. The output is sent to the `db2look.sql` file.

```
db2look -d department -printdbcfg -o db2look.sql
```

- Generate the **CREATE DATABASE** command that was used to create the database, the UPDATE DB CFG statements, and the DDL statements for objects created in the database DEPARTMENT. The output is sent to the `db2look.sql` file.

```
db2look -d department -createdb -printdbcfg -e  -o db2look.sql
```

Usage notes

On Windows operating systems, you must issue the **db2look** command from a Db2 command window.

By default, the instance owner has the EXECUTE privilege on **db2look** packages. For other users to run the **db2look** command, the instance owner has to grant the EXECUTE privilege on **db2look** packages. To determine the **db2look** package names, the **db2bfd** command can be used as follows:

```
cd .../sqllib/bnd
db2bfd -b db2look.bnd
db2bfd -b db2lkfun.bnd
db2bfd -b db2lksp.bnd
```

To create DDL statements for federated objects, you must enable the use of federated systems in the database manager configuration. After the **db2look** command generates the script file, you must set the **federated** configuration parameter to `YES` before running the script. The following **db2look** command parameters are supported in a federated environment:

-ap
Generates AUDIT USING statements.

-e Generates DDL statements for federated objects.

-f Extracts federated-related information from the database manager configuration.

-m Extracts statistics for nicknames.

-x Generates GRANT statements to grant privileges on federated objects.

-xd
Generates DDL statements to add system-granted privileges to federated objects.

-wlm
Generates WLM-specific DDL statements.

If the nickname column and the remote table column are of different data types, then the **db2look** command will generate an ALTER COLUMN statement for the nickname column.

You must modify the output script to add the remote passwords for the CREATE USER MAPPING statements.

You must modify the **db2look** command output script by adding AUTHORIZATION and PASSWORD to those CREATE SERVER statements that are used to define a Db2 family instance as a data source.

Usage of the **-tw** option is as follows:

- To both generate the DDL statements for objects in the DEPARTMENT database associated with tables that have names beginning with abc and send the output to the db2look.sql file:

  ```
  db2look -d department -e -tw abc% -o db2look.sql
  ```

- To generate the DDL statements for objects in the DEPARTMENT database associated with tables that have a d as the second character of the name and to send the output to the db2look.sql file:

  ```
  db2look -d department -e -tw _d% -o db2look.sql
  ```

- The **db2look** command uses the LIKE predicate when evaluating which table names match the pattern specified by the *Tname* argument. Because the LIKE predicate is used, if either the _ character or the % character is part of the table name, the backslash (\) escape character must be used immediately before the _ or the %. In this situation, neither the _ nor the % can be used as a wildcard character in *Tname*. For example, to generate the DDL statements for objects in the DEPARTMENT database associated with tables that have a percent sign in the neither the first nor the last position of the name:

  ```
  db2look -d department -e -tw string\%string
  ```

- Case-sensitive, DBCS, and multi-word table and view names must be enclosed by both a backslash and double quotation marks. For example:

  ```
  \"My TabLe\"
  ```

  If a multibyte character set (MBCS) or double-byte character set (DBCS) name is not enclosed by the backward slash and double quotation delimiter and if it contains the same byte as the lowercase character, it will be converted into uppercase and **db2look** will look for a database object with the converted name. As a result, the DDL statement will not be extracted.

- The **-tw** option can be used with the **-x** option (to generate GRANT privileges), the **-m** option (to return table and column statistics), and the **-l** option (to generate the DDL for user-defined table spaces, database partition groups, and buffer pools). If the **-t** option is specified with the **-tw** option, the **-t** option (and its associated *Tname* argument) is ignored.
- The **-tw** option cannot be used to generate the DDL for tables (and their associated objects) that reside on federated data sources, or on Db2 Universal Database on z/OS and OS/390®, Db2 for i , or Db2 Server for VSE & VM.
- The **-tw** option is only supported via the CLP.

If you try to generate DDL statements on systems with a partitioned database environment, a warning message is displayed in place of the DDL statements for table spaces that are on inactive database partitions. To ensure that correct DDL statements are produced for all table spaces, you must activate all database partitions.

You can issue the **db2look** command from a Db2 client to a database that is of the same or later release as the client, but you cannot issue this command from a client to a database that is of an earlier release than the client. For example, you can issue the **db2look** command from a Version 9.8 client to a Version 10.1 database, but you cannot issue the command from a Version 10.1 client to a Version 9.8 database.

When you invoke the db2look utility, the **db2look** command generates the DDL statements for any object created using an uncommitted transaction.

When you extract a DDL statement for a security label component of type array, the extracted statement might not generate a component whose internal representation (encoding of elements in that array) matches that of the component

in the database for which you issued the **db2look** command. This mismatch can happen if you altered the security label component by adding one or more elements to it. In such cases, data that you extract from one table and moved to another table that you created from **db2look** output will not have corresponding security label values, and the protection of the new table might be compromised.

In a partitioned database environment, if the database was created with table spaces managed by Database Managed Space (DMS) or System Managed Space (SMS) with specified container paths including those defined by $N expressions, the **db2look** `-createdb` generated **CREATE DATABASE** command will list all container paths on each database partition, not just the original specified path or the $N expression. Before you run the generated statement you must adjust the container setting. There is no restriction with the automatic storage option in a partitioned database environment.

In a pureScale environment, the **db2look** `-printdbcfg` command generates the UPDATE DATABASE CONFIGURATION values based on the values of the database member from where the **db2look** `-printdbcfg` command is run.

Related information:

Nickname column and index names

Changing applications for migration

Remote storage requirements

Remote storage aliases are supported by a select number of Db2 commands for accessing data on the IBM® SoftLayer® Object Storage or the Amazon Simple Storage Service (S3). In order to use the select Db2 commands, several configurations to the system are required to account for this feature.

Supported platforms and prerequisites

This feature is supported on SuSE and RHEL Linux. For IBM SoftLayer Object Storage, only the Swift type is currently supported.

The remote storage support in Db2 requires the following packages to be installed:

- libcURL, version 7.19.7 and up
- libxml2, version 2.7.6 and up

Local staging path

A local staging path is needed for holding temporary files in the following situations:

- Downloading an object from a remote storage server
- Uploading an object from a local file system to a remote storage server

The default staging path is in `<instance_directory>/sqllib/tmp/RemoteStorage.xxxx`, where `xxxx` refers to the database partition number.

The default staging path can be configured by the *DB2_OBJECT_STORAGE_LOCAL_STAGING_PATH* variable. The size of the file system for this staging path must be sufficient to hold the working files of the utility, such as the source files for Load, Ingest, and images for Backup and Restore.

Compressed input data

Users are able to load directly from compressed input data files stored in the supported remote storage.

The following common compression formats are supported:

- *.gz - created by gzip utility
- *.zip - created by zip utility

It is required that the compressed file name has the same name as the original file, with the additional .zip or .gz file extension. For example, if a file has the name `db2load.txt`, it is expected that the compressed file name is `db2load.txt.zip` or `db2load.txt.gz`.

Index

H

I

L

M

N

P

R

S

T

U

X

Printed in USA