Microsoft
SQL Server

Guide to Migrating from DB2 to SQL Server and Azure SQL DB

SQL Server Technical Article

Authors: Alexander Pavlov (DB Best Technologies), Andrey Khudyakov (DB Best Technologies), Oksana Eremenko (DB Best Technologies), Stanislav Sklyarov (DB Best Technologies), Alexander Vasyuk (DB Best Technologies)

Technical Reviewers: Dmitry Balin (DB Best Technologies)

Editor: Peter Skjøtt Larsen (DB Best Technologies)

Published: March 2015

Applies to: Microsoft® SQL Server® 2014 and Azure SQL DB®

Summary

In this migration guide you will learn the differences between the IBM DB2 and Microsoft SQL Server database platforms, and the steps necessary to convert a DB2 database to SQL Server and Azure SQL DB.

Created by: DB Best Technologies LLC

P.O. Box 7461, Bellevue, WA 980008

Tel.: (408) 202-4567

E-mail: info@dbbest.com

Web: www.dbbest.com

Copyright

Contents

1.0 Contents Introduction

This migration guide outlines the procedures, issues, and solutions for migrating from IBM DB2 version 10.5 … for Linux, UNIX, or Windows® to Microsoft® SQL Server® 2014 and Azure SQL DB database software. The solutions provided here can also be applied to DB2 UDB for z/OS versions 9.0 and 10.0.

2.0 DB2 to SQL Server Migration

In this section, first a high-level view of the steps for migrating a DB2 database to SQL Server 2014 is summarized. Then you are given an overview of what you must know about converting security items and database objects. The section concludes with a table of recommended type mappings for converting table columns.

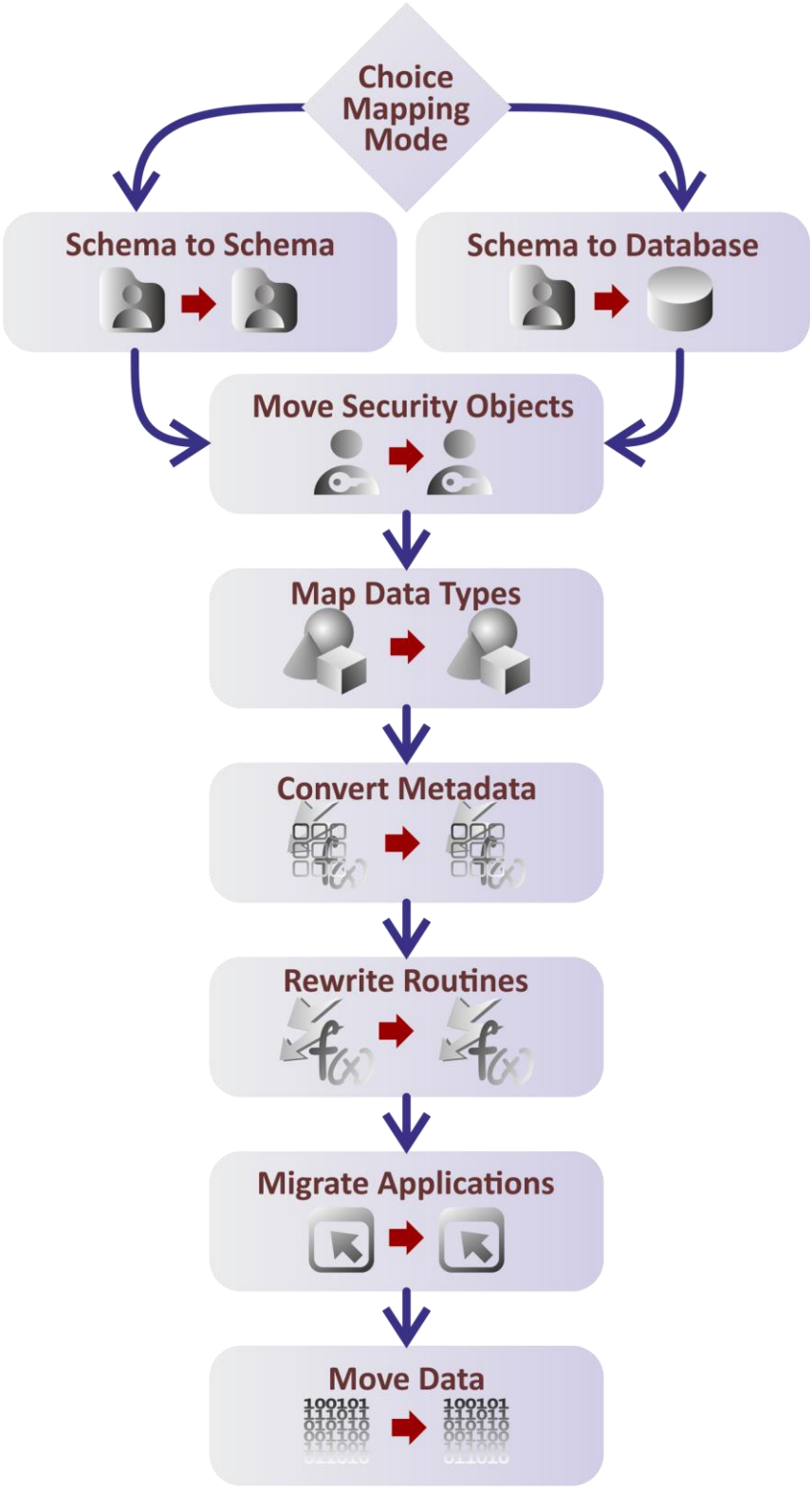
The free Microsoft SQL Server Migration Assistant (SSMA) for DB2 speeds up the migration process. SSMA converts DB2 database objects (including stored procedures) to SQL Server database objects, loads those objects into SQL Server, migrates data from DB2 to SQL Server.

2.1 Overview of Migration Steps

To migrate a DB2 database, perform the following steps, in order:

1. Decide how you will map DB2 databases to SQL Server 2014. You have two main options:
    - Map each DB2 database to a separate SQL Server database. For example, if one of your DB2 databases is named MyDB, you could map the MyDB database to one of your SQL Server databases.
    - Map each DB2 database to a separate schema within a single SQL Server database. For example, you could map your MyDB database to the schema MyDB of your SQL Server database.

      Note: In SQL Server, schemas are not necessarily linked to a specific user or login ID, and one server can contain multiple databases.
2. Migrate security items: the users and login IDs. (See section 2.2.)
3. Map data types from the DB2 data type to a corresponding SQL Server data type. (See section 2.3.)
4. Convert database objects: table columns, triggers, views, sequences, and routines. (See section 2.4.)
5. Rewrite your views, procedures, and functions according to SQL Server syntax. For a description of various issues related to this process, see section 3.0.
6. Change your applications as necessary so that they can work with SQL Server.
7. After a successful database conversion, migrate your data from the old DB2 database to the newly created SQL Server database. For this task you could use SQL Server Integration Services (SSIS), for example.



2.2 Migrating Security Items

There are three main mechanisms within DB2 that allow you to implement a database security plan: authentication, authorization, and privileges. This section first covers these security items, which differ from those used in SQL Server. A separate issue addressed in this section is the conversion of database users.

2.2.1 Authentication

In DB2 client-server applications, login and password checking can be performed on a server, a client, or an intermediate DB2 Connect gateway. Five different types of authentication are available for defining the location: SERVER, SERVER_ENCRYPT, CLIENT, KERBEROS, and KRB_SERVER_ENCRYPT.

Authentication in SQL Server is always performed on the server side only, and is implemented in a different way than in DB2. Because SQL Server offers two different modes: Windows Authentication and Mixed Authentication. Depending on the system architecture, authentication should be adjusted during the conversion.

To manage a trusted connection, DB2 uses the instance configuration parameters TRUST_ALLCLNTS and TRUST_CLNTAUTH. When a user connects to SQL Server using Windows Authentication (by means of a Windows login), SQL Server relies on ("trusts") the operating system to perform authentication checking, and checks only if the Windows user name corresponds to a login that is defined in this instance of SQL Server or if the user's login belongs to a Windows group with a login that is defined in SQL Server. Thus, it is sufficient to use Windows Authentication mode to implement a trusted connection in SQL Server.

2.2.2 Authorization

User authorization defines the list of the commands and objects that are available for a user. This list thereby controls user actions.

In DB2, there are predetermined groups of privileges for authorization, both at the instance level and at the level of a DB2 database.

- **Instance level:** The privileges at the instance level—SYSADM, SYSCTRL, and SYSMAINT—can be granted only to users and user groups of the operating system.
- **Database level:** DBADM and LOAD privileges are granted only on a particular database. Using the GRANT command, the DBADM and LOAD privileges can be granted to an existing user group of the operating system, to a DB2 user group, and to individual users of the operating system or database.

SQL Server has a similar authorization mechanism. Some DB2 privileges of the instance level or the database level can be replaced in SQL Server with predefined server roles, database roles, or a combination. Some privileges can also be replaced by creating new roles at the database level, which can be assigned to users and groups.

2.2.3 Privileges

DB2 privileges generally fall into two main categories: database-level privileges, which span all objects within the database, and object-level privileges, which are associated with a specific object. Converting privileges to SQL Server permissions is performed by means of permission sets, which can be granted to both users and groups. Depending on whether database-level or object-level privileges are being converted, access permissions to a definite database or its objects are granted to either groups or users in SQL Server. All existing DB2 privileges can be replaced with the equivalent SQL Server permissions or a combination of permissions.

2.2.4 Converting Users

In DB2, both users and groups can access an instance or a database. These users and groups can be registered in the operating system or can simply have been granted access to it. Also, access can be given to users and groups that belong to a domain in which the DB2 database server or the user workstation is included.

SQL Server also supports these possibilities. Thus, to move users or groups to SQL Server, first create those users or groups in the new operating system; next, create a login for each group and user in SQL Server and then create a user in the SQL Server target database for each login. To create a new login for the Login Name field in the master database, choose the user or group from a list of those registered in the operating system or domain. (You can also perform this step by means of SQL CREATE LOGIN command.) Finally, after creating server logins and database users, grant permissions corresponding to the privileges that these users and groups had in DB2.

2.3 Mapping Data Types

Most data types used in DB2 do not have exact equivalents in Microsoft SQL Server. They differ in scale, precision, length, and functionality. This specification explains the data type mapping for table columns and includes remarks about conversion issues.

Section 3.11, "User-Defined Types," covers migration of user-defined types.

2.3.1 The DB2 10 for z/OS Buil-in Data Types

Table 1. DB2 Built-in Data Types.

Area	DB2 Data Type			SQL Data Type
Datetime	Date			**date**
	Time			**time**
	Timestamp	Timestamp Without Timezone		**timestamp without timezone**
		Timestamp With Timezone		**timestamp with timezone**
String	Character	Fixed Length		**char**
		Varying Length		**varchar**
				clob
	Graphics	Fixed Length		**graphic**
		Varying Length		**vargraphic**
				dbclob
	Binaries	Fixed Length		**binary**
		Varying Length		**varbinary**
				blob
Signed Numeric	Exact	Binary Integer	16 Bit	**smallint**
			32 Bit	**integer**
			64 Bit	**bigint**
		Decimal	Packed	**decimal**
	Decimal Floating Point			**decfloat**
	Approximate Floating Point	Single Precision		**real**
		Double Precision		**double**
Row Identifier				**rowed**
Xml				**xml**

2.3.2 The DB2 Version 10.5.0 for Linux, UNIX, and Windows Built-in Data Types

Table 2. DB2 LUW Built in Data Types.

Area	DB2 Data Type			SQL Data Type
Datetime	Date			**date**
	Time			**time**
	Timestamp			**timestamp**
String	Character	Fixed Length		**char**
		Varying Length		**varchar**
				clob
	Graphics	Fixed Length		**graphic**
		Varying Length		**vargraphic**
				dbclob
	Binary	Varying Length		**blob**
Signed Numeric	Exact	Binary Integer	16 Bit	**smallint**
			32 Bit	**integer**
			64 Bit	**bigint**
		Decimal	Packed	**decimal**
	Decimal Floating Point			**decfloat**
	Approximate Floating Point	Single Precision		**real**
		Double Precision		**double**
Extensible markup language				**xml**

2.3.3 String data types

DB2® supports several types of string data: character strings, graphic strings, and binary strings.

Character strings contain text and can be either a fixed-length or a varying-length. Graphic strings contain graphic data, which can also be either a fixed-length or a varying-length. Binary strings contain strings of binary bytes and can be either a fixed-length or a varying-length. All of these types of string data can be represented as large objects.

All SQL Server character strings data types listed in the Table 3 applies to: SQL Server (SQL Server 2008 through current version), Windows Azure SQL Database (Initial release through current release).

All DB2 character strings data types listed in the Table 3 applies to: DB2 Version 10.5.0 for Linux, UNIX, and Windows, DB2 Version 10 for z/OS.

Table 3. Character Strings data types - lists the recommended type mappings for converting table columns.

DB2 v.10 for z/OS and DB2 v.10.5.0 for LUW data type	DB2 data type description	Recommended SQL Server 2012/2014 and Azure SQL DB Data Type	SQL Server 2012/2014 and Azure SQL DB Data Type description	Alternative SQL Server 2012/2014 and Azure SQL DB Data Types
CHARACTER(n)	Fixed-length character strings with a length of n bytes. n must be greater than 0 and not greater than 255. The default length is 1.	**char [(n)]**	Fixed-length, non-Unicode string data. n defines the string length and must be a value from 1 through 8,000. The storage size is n bytes. The ISO synonym for char is character .	**varchar(n)-** variable-length non-Unicode string data. n 1-8000 characters; **nchar(**n)- fixed-length Unicode string data. n 1-4000 characters; **nvarchar(n) -** variable-length Unicode string data. n 1-4000 characters;
DB2 **CHARACTER(n)** data type can be successfully mapped to **char [(n)]** data type.				
VARCHAR(n)	Varying-length character strings with a maximum length of n bytes. n must be greater than 0 and less than a number that depends on the page size of the table space. The maximum length is 32704.	**varchar [(max)]**	Variable-length, non-Unicode string data. n defines the string length and can be a value from 1 through 8,000. max indicates that the maximum storage size is 2^31-1 bytes (2 GB). The storage size is the actual length of the data entered + 2 bytes.	**text**-variable-length non-Unicode data. Length 2^31 - 1 (2,147,483,647) characters; **nvarchar(n|max)-** variable-length Unicode string data. n 1-4000 characters. max indicates size 2^31-1 bytes (2 GB); **ntext**-variable-length Unicode data. Length is 2^30 - 1 (1,073,741,823) characters; **varchar(n)-** variable-length non-Unicode string data. n 1-8000 characters; **char (n) -** non-Unicode string data. n 1-8000 characters**;**
DB2 **VARCHAR(n)** data type can be successfully mapped to SQL Server **varchar [(max)]** data type.				
CLOB(n)	Varying-length character strings with a maximum	**varchar [(max)]**	Variable-length, non-Unicode string data.	**nvarchar(max)-** variable-length Unicode string

DB2 v.10 for z/OS and DB2 v.10.5.0 for LUW data type	DB2 data type description	Recommended SQL Server 2012/2014 and Azure SQL DB Data Type	SQL Server 2012/2014 and Azure SQL DB Data Type description	Alternative SQL Server 2012/2014 and Azure SQL DB Data Types
	of n characters. n cannot exceed 2 147 483 647. The default length is 1M.		max indicates that the maximum storage size is 2^31-1 bytes (2 GB). The storage size is the actual length of the data entered + 2 bytes.	data. max indicates size 2^31-1 bytes (2 GB)**;** **text**-variable-length non-Unicode data. Length 2^31 - 1 (2,147,483,647) characters; **ntext**-variable-length Unicode data. Length is 2^30 - 1 (1,073,741,823) characters;
The best choice for migrating DB2 large object types (LOBs) such as **CLOB(n)** is SQL Server **varchar(max)** data type.				
GRAPHIC(n)	Fixed-length graphic strings that contain n double-byte characters. n must be greater than 0 and less than 128. The default length is 1.	**nchar [(n)]**	Fixed-length Unicode string data. n defines the string length and must be a value from 1 through 4,000. The storage size is two times n bytes. When the collation code page uses double-byte characters, the storage size is still n bytes. Depending on the string, the storage size of n bytes can be less than the value specified for n.	**nvarchar(n)-** variable-length Unicode string data. n 1-4000 characters; **ntext**-variable-length Unicode data. Length is 2^30 - 1 (1,073,741,823) characters;
DB2 **GRAPHIC(n)** data type can be successfully mapped to SQL Server **nchar [(n)]** data type.				
VARGRAPHIC(n)	Varying-length graphic strings that contain n double-byte. The maximum length, n, must be greater than 0 and less than a number that depends on the page size of the table space. The maximum length is 16352.	**nvarchar [(max)]**	Variable-length Unicode string data. max indicates that the maximum storage size is 2^31-1 bytes (2 GB). The storage size, in bytes, is two times the actual length of data entered + 2 bytes.	**ntext**-variable-length Unicode data. Length is 2^30 - 1 (1,073,741,823) characters; **nchar [(n)]-**fixed-length Unicode string data. n 1-4000;

DB2 v.10 for z/OS and DB2 v.10.5.0 for LUW data type	DB2 data type description	Recommended SQL Server 2012/2014 and Azure SQL DB Data Type	SQL Server 2012/2014 and Azure SQL DB Data Type description	Alternative SQL Server 2012/2014 and Azure SQL DB Data Types
DB2 **VARGRAPHIC(n)** data type can be successfully mapped to SQL Server **nvarchar [(max)]** data type.				
DBCLOB(n)	Varying-length strings of double-byte characters with a maximum of n double-byte characters. n cannot exceed 1 073 741 824. The default length is 1M.	**nvarchar [(max)]**	Variable-length Unicode string data. max indicates that the maximum storage size is 2^31-1 bytes (2 GB). The storage size, in bytes, is two times the actual length of data entered + 2 bytes.	**ntext**-variable-length Unicode data. Length is 2^30 - 1 (1,073,741,823) characters;
The best choice for migrating DB2 large object types (LOBs) such as **DBCLOB(n)** is SQL Server **nvarchar(max)** data type.				
BINARY(n) *	Fixed-length or varying-length binary strings with a length of n bytes. n must be greater than 0 and not greater than 255. The default length is 1.	**binary [(n)]**	Fixed-length binary data with a length of n bytes, where n is a value from 1 through 8,000. The storage size is n bytes.	**varbinary [(n)] -** variable-length binary data. n can be a value from 1 through 8000 bytes;
*** Restriction:** BINARY(n) data type can use as table column only in DB2 Version 10 for z/OS. DB2 Version 10 for z/OS **BINARY(n)** data type can be successfully mapped to SQL Server **binary [(n)]** data type.				
VARBINARY(n) *	Varying-length binary strings with a length of n bytes. The length of n must be greater than 0 and less than a number that depends on the page size of the table space. The maximum length is 32704.	**varbinary [(max)]**	Variable-length binary data. max indicates that the maximum storage size is 2^31-1 bytes. The storage size is the actual length of the data entered + 2 bytes. The data that is entered can be 0 bytes in length.	**varbinary [(n)] -** variable-length binary data. n can be a value from 1 through 8000 bytes; **binary [(n)]-**fixed-length binary data with a length of n bytes, where n 1-8000 bytes;
*** Restriction:** VARBINARY(n) data type can use as table column only in DB2 Version 10 for z/OS. DB2 Version 10 for z/OS **VARBINARY(n)** data type can be successfully mapped to SQL Server **varbinary [(max)]** data type.				
BLOB(n)	Varying-length binary strings with a length	**varbinary [(max)]**	Variable-length binary data.	**image** - binary data, size is 0 - 2^31 – 1

DB2 v.10 for z/OS and DB2 v.10.5.0 for LUW data type	DB2 data type description	Recommended SQL Server 2012/2014 and Azure SQL DB Data Type	SQL Server 2012/2014 and Azure SQL DB Data Type description	Alternative SQL Server 2012/2014 and Azure SQL DB Data Types
	of n bytes. n cannot exceed 2 147 483 647. The default length is 1M.		max indicates that the maximum storage size is 2^31-1 bytes. The storage size is the actual length of the data entered + 2 bytes. The data that is entered can be 0 bytes in length.	(2 147 483 647) bytes (2 GB); **binary**- fixed-length binary data with a length of n bytes, where n is a value from 1 through 8000 bytes;
The best choice for migrating DB2 large object types (LOBs) as **BLOB(n)** is SQL Server **varbinary(max)** data type.				
LONG VARCHAR		**varchar [(max)]**		**text;** **nvarchar(n|max);** **ntext;** **varchar(n);** **char (n);**
[LONG] VARCHAR (n) FOR BIT DATA		**varbinary [(max)]**		**varbinary [(n)];** **image;** **binary;**
LONG VARGRAPHIC		**nvarchar [(max)]**		**ntext**; **nchar [(n)]**;
CHAR (N) FOR BIT DATA		**binary [(n)]**		**varbinary [(n)];**

2.3.3.1 Binary strings

A binary string is a sequence of bytes. Unlike character strings, which usually contain text data, binary strings are used to hold non-traditional data such as pictures, voice, or mixed media. Character strings of the FOR BIT DATA subtype may be used for similar purposes, but the two data types are not compatible. The BLOB scalar function can be used to cast a FOR BIT DATA character string to a binary string.

Certain database columns can be declared FOR BIT DATA. These columns, which generally contain characters, are used to hold binary information. The CHAR(n), VARCHAR, LONG VARCHAR can contain binary data. Use these data types when working with columns with the FOR BIT DATA attribute.

2.3.3.2 Graphic strings

A graphic string is a sequence of bytes that represents double-byte character data. The length of the string is the number of double-byte characters in the sequence. If the length is zero, the value is called the empty string. This value should not be confused with the null value. Graphic strings are not supported in a database defined with a single-byte code page.

Graphic strings are not checked to ensure that their values contain only double-byte character code points. (The exception to this rule is an application precompiled with the WCHARTYPE CONVERT option. In this case, validation does occur.) Rather, the database manager assumes that double-byte character data is contained in graphic data fields. The database manager does check that a graphic string value is an even number of

bytes long. This data type cannot be created in a table. It can only be used to insert data into and retrieve data from the database.

2.3.3.3 Fixed-length graphic strings (GRAPHIC)

All values in a fixed-length graphic string column have the same length, which is determined by the length attribute of the column. The length attribute must be between 1 and 127, inclusive.

2.3.3.4 Varying-length graphic strings

There are two types of varying-length graphic string:

- A VARGRAPHIC value can be up to 16 336 double-byte characters long.
- A DBCLOB (double-byte character large object) value can be up to 1 073 741 823 double-byte characters long. A DBCLOB is used to store large DBCS character-based data (such as documents written with a single character set) and, therefore, has a DBCS code page associated with it.

Special restrictions apply to an expression that results in a varying-length graphic string whose maximum length is greater than 127 bytes. These restrictions are the same as those specified in Varying-length character strings.

2.3.3.5 LONG statement

Originally VARCHAR was limited to a length of 255, so LONG VARCHAR was needed, but obviously they both support about 32K now VARCHAR supports 28 bytes less. One big difference between VARCHAR and LONG VARCHAR is that LONG VARCHAR is stored in a separate area like a LOB (CLOB, BLOB, etc) and also like a LOB, does not use bufferpools, so every select, insert, update, or delete of a LONG VARCHAR requires direct disk I/O, just like LOBs.

2.3.4 Numeric data types

DB2® supports several types of numeric data types, each of which has its own characteristics.

For numeric data, use numeric columns rather than string columns. Numeric columns require less space than string columns, and DB2 verifies that the data has the assigned type.

All SQL Server numeric data type listed in the Table 4 applies to: SQL Server (SQL Server 2008 through current version), Windows Azure SQL Database (Initial release through current release).

All DB2 numeric data type listed in the Table 4 applies to: DB2 Version 10.5.0 for Linux, UNIX, and Windows, DB2 Version 10 for z/OS.

Table 4. Numeric data types. - lists the recommended type mappings for converting table columns.

DB2 v.10 for z/OS and DB2 v.10.5.0 for LUW data type	DB2 data type description	Recommended SQL Server 2012/2014 and Azure SQL DB Data Type	SQL Server 2012/2014 and Azure SQL DB Data Type description	Alternative SQL Server 2012/2014 and Azure SQL DB Data Types
SMALLINT	Small integers. A small integer is binary integer with a precision of 15 bits. The range is -32768 to +32767.	**smallint**	Range -2^15 (-32,768) to 2^15-1 (32,767) Storage 2 Bytes	**int-**range -2^15 to 2^15-1, storage 4 bytes; **bigint--**2^63 to 2^63-1, storage 8 bytes**;**

DB2 v.10 for z/OS and DB2 v.10.5.0 for LUW data type	DB2 data type description	Recommended SQL Server 2012/2014 and Azure SQL DB Data Type	SQL Server 2012/2014 and Azure SQL DB Data Type description	Alternative SQL Server 2012/2014 and Azure SQL DB Data Types
				tinyint- 0 to 255, storage 1 byte**;**
DB2 **SMALLINT** data type can be successfully mapped to SQL Server **smallint** data type.				
INTEGER or INT	Large integers. A *large integer* is binary integer with a precision of 31 bits. The range is -2147483648 to +2147483647.	**int**	Range -2^31 (-2,147,483,648) to 2^31-1 (2,147,483,647) Storage 4 Bytes	**bigint-**-2^63 to 2^63-1, storage 8 bytes**;** **smallint-** -2^15 to 2^15-1, storage 2 bytes; **tinyint-** 0 to 255, storage 1 byte**;**
DB2 **INTEGER** data type can be successfully mapped to SQL Server **int** data type.				
BIGINT	Big integers. A *big integer* is a binary integer with a precision of 63 bits. The range of big integers is -9223372036854775808 to +922337203685477 5807.	**bigint**	Range -2^63 (-9,223,372,036,854, 775,808) to 2^63-1 (9,223,372,036,854, 775,807) Storage 8 Bytes	**int-**range -2^15 to 2^15-1, storage 4 bytes; **smallint-** -2^15 to 2^15-1, storage 2 bytes; **tinyint-** 0 to 255, storage 1 byte**;**
DB2 **BIGINT** data type can be successfully mapped to SQL Server **bigint** data type.				
DECIMAL or NUMERIC	A *decimal* number is a packed decimal number with an implicit decimal point. The position of the decimal point is determined by the precision and the scale of the number. The scale, which is the number of digits in the fractional part of the number, cannot be negative or greater than the precision. The maximum precision is 31 digits. All values of a decimal column have the same precision and scale. The range of a decimal variable or the numbers in a	**decimal [(p[,s])] and numeric[(p[,s])]**	Fixed precision and scale numbers. When maximum precision is used, valid values are from - 10^38 +1 through 10^38 - 1. p (precision) - the maximum total number of decimal digits that will be stored, both to the left and to the right of the decimal point. The precision must be a value from 1 through the maximum precision of 38. The default precision is 18. s (scale) - the number of decimal digits that will be stored to the right of the decimal point.	

DB2 v.10 for z/OS and DB2 v.10.5.0 for LUW data type	DB2 data type description	Recommended SQL Server 2012/2014 and Azure SQL DB Data Type	SQL Server 2012/2014 and Azure SQL DB Data Type description	Alternative SQL Server 2012/2014 and Azure SQL DB Data Types
	decimal column is -*n* to +*n*, where *n* is the largest positive number that can be represented with the applicable precision and scale. The maximum range is 1 - 10^{31} to 10^{31} - 1.		This number is substracted from p to determine the maximum number of digits to the left of the decimal point. The maximum number of decimal digits that can be stored to the right of the decimal point. Scale must be a value from 0 through p. Scale can be specified only if precision is specified. The default scale is 0; therefore, 0 <= s <= p. Maximum storage sizes vary, based on the precision.	
DB2 **DECIMAL or NUMERIC** data types can be successfully mapped to SQL Server **decimal [(p[,s])] and numeric[(p[,s])]** data types respectively.				
DECFLOAT	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 maximum precision is 34 digits. The range of a decimal floating-point number is either 16 or 34 digits of precision; the exponent range is respectively 10-383 to 10+384 or 10-6143 to 10+6144.	**numeric[(p[,s])]**	Fixed precision and scale numbers. When maximum precision is used, valid values are from - 10^38 +1 through 10^38 - 1	
DB2 **DECFLOAT** data type can be successfully mapped to SQL Server **numeric[(p[,s])]** data type.				
REAL	A *single-precision floating-*	**real**	Range - 3.40E + 38 to -1.18E - 38, 0	

DB2 v.10 for z/OS and DB2 v.10.5.0 for LUW data type	DB2 data type description	Recommended SQL Server 2012/2014 and Azure SQL DB Data Type	SQL Server 2012/2014 and Azure SQL DB Data Type description	Alternative SQL Server 2012/2014 and Azure SQL DB Data Types
	point number is a short floating-point number of 32 bits. The range of single-precision floating-point numbers is approximately -7.2E+75 to 7.2E+75. In this range, the largest negative value is about -5.4E-79, and the smallest positive value is about 5.4E-079.		and 1.18E - 38 to 3.40E + 38 storage 4 bytes Approximate-number data types for use with floating point numeric data. Floating point data is approximate; therefore, not all values in the data type range can be represented exactly.	
DB2 **REAL** data type can be successfully mapped to SQL Server **real** data type.				
DOUBLE	A double-precision floating-point number is a long floating-point number of 64-bits. The range of double-precision floating-point numbers is approximately -7.2E+75 to 7.2E+75. In this range, the largest negative value is about -5.4E-79, and the smallest positive value is about 5.4E-079.	**float [(n|53)]**	Range - 1.79E+308 to -2.23E-308, 0 and 2.23E-308 to 1.79E+308 Approximate-number data types for use with floating point numeric data. Floating point data is approximate; therefore, not all values in the data type range can be represented exactly. n is the number of bits that are used to store the mantissa of the float number in scientific notation and, therefore, dictates the precision and storage size. If n is specified, it must be a value between 1 and 53. The default value of n is 53.	
DB2 **DOUBLE** data type can be successfully mapped to SQL Server **float [(n|53)]** data type.				

2.3.5 Date, time, and timestamp data types

The datetime data types are DATE, TIME, and TIMESTAMP.

All SQL Server date, time, and timestamp data types listed in the Table 5 applies to: SQL Server (SQL Server 2008 through current version), Windows Azure SQL Database (Initial release through current release).

All DB2 date, time, and timestamp data types listed in the Table 5 applies to: DB2 Version 10.5.0 for Linux, UNIX, and Windows, DB2 Version 10 for z/OS.

Table 5. Date, time, and timestamp data types.

DB2 v.10 for z/OS and DB2 v.10.5.0 for LUW data type	DB2 data type description	Recommended SQL Server 2012/2014 and Azure SQL DB Data Type	SQL Server 2012/2014 and Azure SQL DB Data Type description	Alternative SQL Server 2012/2014 and Azure SQL DB Data Types
DATE	A *date* is a three-part value representing a year, month, and day in the range of 0001-01-01 to 9999-12-31.	**date**	Default string literal format YYYY-MM-DD YYYY is four digits from 0001 to 9999 that represent a year. MM is two digits from 01 to 12 that represent a month in the specified year. DD is two digits from 01 to 31, depending on the month, that represent a day of the specified month.	**datetime2-** defines a date that is combined with a time of day that is based on 24-hour clock; **datetime-**defines a date that is combined with a time of day with fractional seconds that is based on a 24-hour clock; **datetimeoffset [(fractional seconds precision)] -** defines a date that is combined with a time of a day that has time zone awareness and is based on a 24-hour clock; **smalldatetime -** defines a date that is combined with a time of day. The time is based on a 24-hour day, with seconds always zero (:00) and without fractional seconds.
DB2 **DATE** data type can be successfully mapped to SQL Server **date** data type.				
TIME	A *time* is a three-part value representing a time of day in hours, minutes, and seconds, in the range of	**time [(fractional second precision)]**	**Default string literal format hh:mm:ss[.nnnnnnn] hh is two digits, ranging from 0 to**	**datetime2; datetime; datetimeoffset; smalldatetime;**

DB2 v.10 for z/OS and DB2 v.10.5.0 for LUW data type	DB2 data type description	Recommended SQL Server 2012/2014 and Azure SQL DB Data Type	SQL Server 2012/2014 and Azure SQL DB Data Type description	Alternative SQL Server 2012/2014 and Azure SQL DB Data Types
	00.00.00 to 24.00.00.		**23, that represent the hour.** **mm is two digits, ranging from 0 to 59, that represent the minute.** **ss is two digits, ranging from 0 to 59, that represent the second.** **n* is zero to seven digits, ranging from 0 to 9999999, that represent the fractional seconds.**	
DB2 **TIME** data type can be successfully mapped to SQL Server **time [(fractional second precision)]** data type. When a converted TIME column is retrieved in SQL Server 2014, it should be used together with the DATEPART function, which picks the time element out of the smalldatetime type.				
TIMESTAMP *	A timestamp is a seven-part value representing a date and time by year, month, day, hour, minute, second, and microsecond, in the range of 0001-01-01-00.00.00.00000 0000 to 9999-12-31-24.00.00.00000 0000 with nanosecond precision. Timestamps can also hold timezone information.	**datetimeoffset**	Defines a date that is combined with a time of a day that has time zone awareness and is based on a 24-hour clock. datetimeoffset [(fractional seconds precision)] Default string literal formats YYYY-MM-DD hh:mm:ss[.nnnnnnn] [{+|-}hh:mm] YYYY is four digits, ranging from 0001 through 9999, that represent a year. MM is two digits, ranging from 01 to 12, that represent a month in the specified year. DD is two digits, ranging from 01 to 31 depending on the month, that represent a day of the specified month.	**datetime2;** **datetime;** **smalldatetime;** **date;**

DB2 v.10 for z/OS and DB2 v.10.5.0 for LUW data type	DB2 data type description	Recommended SQL Server 2012/2014 and Azure SQL DB Data Type	SQL Server 2012/2014 and Azure SQL DB Data Type description	Alternative SQL Server 2012/2014 and Azure SQL DB Data Types
			hh is two digits, ranging from 00 to 23, that represent the hour. mm is two digits, ranging from 00 to 59, that represent the minute. ss is two digits, ranging from 00 to 59, that represent the second. n* is zero to seven digits, ranging from 0 to 9999999, that represent the fractional seconds. hh is two digits that range from -14 to +14. mm is two digits that range from 00 to 59.	
DB2 **TIMESTAMP** data type can be successfully mapped to SQL Server **datetimeoffset** data type. *** Restriction:** Timestamps can hold timezone information only for DB2 Version 10 for z/OS.				

2.3.6 XML, ROWID data types

XML, uniqueidentifier SQL Server data types describe in the Table 6 applies to: SQL Server (SQL Server 2008 through current version), Windows Azure SQL Database (Initial release through current release). XML, ROWID DB2 data types describe in the Table 6 applies to: DB2 Version 10.5.0 for Linux, UNIX, and Windows, DB2 Version 10 for z/OS.

Table 6. XML, ROWID data types.

Table 4. XML, ROWID data types				
DB2 v.10 for z/OS and DB2 v.10.5.0 for LUW data type	**DB2 data type description**	**Recommended SQL Server 2012/2014 and Azure SQL DB Data Type**	**SQL Server 2012/2014 and Azure SQL DB Data Type description**	**Alternative SQL Server 2012/2014 and Azure SQL DB Data Types**
XML	The XML data type is used to define columns of a table that store XML values. This pureXML® data type provides the	**XML**	xml([CONTENT|DOCUMENT] xml_schema_collection) **Is the data type that stores XML data. You can**	**varchar(n), nvarchar(n), varchar(max),**

Table 4. XML, ROWID data types				
DB2 v.10 for z/OS and DB2 v.10.5.0 for LUW data type	**DB2 data type description**	**Recommended SQL Server 2012/2014 and Azure SQL DB Data Type**	**SQL Server 2012/2014 and Azure SQL DB Data Type description**	**Alternative SQL Server 2012/2014 and Azure SQL DB Data Types**
	ability to store well-formed XML documents in a database.		**store xml instances in a column, or a variable of xml type.** CONTENT Restricts the xml instance to be a well-formed XML fragment. The XML data can contain multiple zero or more elements at the top level. Text nodes are also allowed at the top level. This is the default behavior. DOCUMENT Restricts the xml instance to be a well-formed XML document. The XML data must have one and only one root element. Text nodes are not allowed at the top level. xml_schema_collection Is the name of an XML schema collection. To create a typed xml column or variable, you can optionally specify the XML schema collection name. For more information about typed and untyped XML, see Compare Typed XML to Untyped XML.	**nvarchar(max)**
ROWID *	Use ROWID data type to uniquely and permanently identify rows in a DB2® subsystem. ROWID is externalized as a nn-byte value. For example, 63C6AB6415CED248260401D3701401000000000 00201, but stored as VARCHAR (17). Also ROWID could externalized in hex value. For example, 000E 63C6AB64 15CED248 260401D3	**uniqueidentifier**	The uniqueidentifier data type stores 16-byte binary values that operate as globally unique identifiers (GUIDs). **A uniqueidentifier value is not typically defined as a constant. You can specify a uniqueidentifier constant in the following ways:** **Character string format:** **'6F9619FF-8B86-D011-B42D-0C04FC964FF'** **Binary format:**	

Table 4. XML, ROWID data types				
DB2 v.10 for z/OS and DB2 v.10.5.0 for LUW data type	**DB2 data type description**	**Recommended SQL Server 2012/2014 and Azure SQL DB Data Type**	**SQL Server 2012/2014 and Azure SQL DB Data Type description**	**Alternative SQL Server 2012/2014 and Azure SQL DB Data Types**
	7014. The value '000E' declares the length of the ROWID column, which is currently 000E in hex and 14 in decimal. DB2 can generate a value for the column when a row is added, depending on the option that you choose (GENERATED ALWAYS or GENERATED BY DEFAULT) when you define the column. You can use a ROWID column in a table for several reasons. You can define a ROWID column to include LOB data in a table. You can use direct-row access so that DB2 accesses a row directly through the ROWID column. If an application selects a row from a table that contains a ROWID column, the row ID value implicitly contains the location of the row. If you use that row ID value in the search condition of subsequent SELECT statements, DB2 might be able to navigate directly to the row.		**0xff19966f868b11d0b42d00c04fc964ff** A column or local variable of uniqueidentifier data type can be initialized to a value in the following ways: By using the NEWID function. By converting from a string constant in the form xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx, in which each x is a hexadecimal digit in the range 0-9 or a-f. For example, 6F9619FF-8B86-D011-B42D-00C04FC964FF is a valid uniqueidentifier value. Comparison operators can be used with uniqueidentifier values . However, ordering is not implemented by comparing the bit patterns of the two values. The only operations that can be performed against a uniqueidentifier value are comparisons (=, <>, <, >, <=, >=) and checking for NULL (IS NULL and IS NOT NULL). No other arithmetic operators can be used. All column constraints and properties, except IDENTITY, can be used on the uniqueidentifier data type. Merge replication and transactional replication with updating subscriptions use uniqueidentifier columns to guarantee that rows are uniquely identified across multiple copies of the table.	

Table 4. XML, ROWID data types				
DB2 v.10 for z/OS and DB2 v.10.5.0 for LUW data type	**DB2 data type description**	**Recommended SQL Server 2012/2014 and Azure SQL DB Data Type**	**SQL Server 2012/2014 and Azure SQL DB Data Type description**	**Alternative SQL Server 2012/2014 and Azure SQL DB Data Types**
*** Restriction**: ROWID data type can use as table column only in DB2 Version 10 for z/OS. In DB2 Version 10.5.0 for LUW data type ROWID can use only as system information * **Restriction on mapping ROWID data type to uniqueidentifier:** ROWID data type uniquely identifies rows in a DB2® subsystem and represents physical location of the rows. ROWID data type could be mapped to uniqueidentifier, which is a GUID that could be generated for each row. Notice that the ROWID value for a particular row in a table might change over time due to a REORG of the table space. Also you must remember that ROWID the column which implicitly contains the location of the row it's not the same as uniqueidentifier which used to uniquely identify rows in table.				

2.3.6.1 Specifying direct row access by using row IDs

For some applications, you can use the value of a ROWID column to navigate directly to a row. When you select a ROWID column, the value implicitly contains the location of the retrieved row. If you use the value from the ROWID column in the search condition of a subsequent query, DB2® can choose to navigate directly to that row.

For DB2 to be able to use direct row access for the update operation, the SELECT from INSERT statement and the UPDATE statement must execute within the same unit of work. If these statements execute in different units of work, the ROWID value for the inserted row might change due to a REORG of the table space before the update operation. Alternatively, you can use a SELECT from MERGE statement. The MERGE statement performs INSERT and UPDATE operations as one coordinated statement.

2.3.6.2 ROWID columns as keys

If you define a column in a table to have the ROWID data type, DB2 provides a unique value for each row in the table only if you define the column as GENERATED ALWAYS. The purpose of the value in the ROWID column is to uniquely identify rows in the table.

You can use a ROWID column to write queries that navigate directly to a row, which can be useful in situations where high performance is a requirement. This direct navigation, without using an index or scanning the table space, is called direct row access. In addition, a ROWID column is a requirement for tables that contain LOB columns.

Requirement: To use direct row access, you must use a retrieved ROWID value before you commit. When your application commits, it releases its claim on the table space. After the commit, a REORG on your table space might execute and change the physical location of the rows.

Restriction: In general, you cannot use a ROWID column as a key that is to be used as a single column value across multiple tables. The ROWID value for a particular row in a table might change over time due to a REORG of the table space. In particular, you cannot use a ROWID column as part of a parent key or foreign key.

The value that you retrieve from a ROWID column is a varying-length character value that is not monotonically ascending or descending (the value is not always increasing or not always decreasing). Therefore, a ROWID column does not provide suitable values for many types of entity keys, such as order numbers or employee numbers.

2.3.6.3 ROWID columns

There are two different ways of defining a column to be a ROWID data type in a CREATE TABLE statement:

```
COLNAME ROWID GENERATED ALWAYS
COLNAME ROWID GENERATED BY DEFAULT
```

Using the GENERATED ALWAYS keyword, DB2 always generates a ROWID when inserting a row. Applications and users are not allowed to insert a ROWID.

If you use GENERATED BY DEFAULT, users and applications can supply a value for a ROWID column as long as the value was previously generated by DB2 and a unique, single column index that exists on the ROWID column. DB2 checks that the value you are going to insert is a valid ROWID. It is not sufficient to provide unique numbers yourself. You should only use this parameter when inserting data from another table for purposes of moving data. The recommended usage is GENERATED ALWAYS. As mentioned above, you have to create a unique index on the ROWID column when you specify GENERATED BY DEFAULT.

Make sure that there is no way to use the GENERATED ALWAYS clause before implementing GENERATED BY DEFAULT, because the additional index on a table may increase your response time for inserting and deleting transactions on the base table. The index is not affected by an UPDATE statement since the ROWID is not updateable. If you try to update a ROWID column, DB2 issues SQLCODE -151, because the catalog description indicates that this column cannot be updated.

Attention: When you specify GENERATED BY DEFAULT for a ROWID column, make sure that a single column unique index exists on your ROWID column. ROWID values can never contain null values, so the ROWID column has to be defined as NOT NULL.

Be aware that a ROWID column implies some restrictions, preventing the values in the column from being manipulated:

- Users are not allowed to update a ROWID column.
- Null values cannot be assigned to ROWID columns.
- EDITPROCs, FIELDPROCs and CHECK CONSTRAINTs are not provided for ROWIDs.
- It is not allowed to load a single partition or a range of partitions if a column of data type
- ROWID is part of the partitioning key.
- The ROWID column is stored like a VARCHAR (17) column. In DB2 V7 two different types of ROWIDs can be defined.

2.4 Converting Database Objects

This section briefly describes methods of conversion for miscellaneous database objects.

2.4.1 Tables, indexes

The data type of table columns should be converted using selected data type mapping. For information about recommended type mappings, see section 2.3, “Mapping Data Types.” See the issues related to table conversion in section 3.1.1, “Migrating Tables – CREATE TABLE Statement.”

To learn about the differences between index definitions, see the CREATE INDEX description in section 3.1.3, “Migrating Indexes – CREATE INDEX Statement.”

2.4.2 Triggers

Generally, DB2 triggers can be converted to SQL Server triggers. The type of a trigger may need to be changed; for example, you should convert the BEFORE trigger to INSTEAD OF. Also, some additional code may need to be added to a trigger, and references to new/old tables and rows should be replaced. For details, see section 3.2, “Triggers.”

2.4.3 Views

In most cases, views are compatible and problems appear only when converting an underlying SELECT statement.

2.4.4 Sequences

In most cases, SEQUENCEs are compatible and problems appear only when converting an PREVIOUS VALUE expression . For details, see section 3.3, “Sequences.”

2.4.5 Routines

Routines include stored procedures and user-defined functions. To read details about their conversion, see section 3.5.1, “Procedures” and section 3.5.2, “User-Defined Functions.”

Not all DB2 database objects have direct equivalents in SQL Server. In some cases, SSMA creates additional objects to provide the proper emulation.

3.0 DB2 Migration Issues

This section identifies problems that may occur when migrating from DB2 9.x to SQL Server, and suggests ways to handle those problems.

3.1 CREATE Statements

This section compares the CREATE TABLE, CREATE VIEW, and CREATE INDEX statements in DB2 and SQL Server.

3.1.1 Migrating Tables – CREATE TABLE Statement

DB2 databases store data in tables. In addition to tables used to store persistent data, there are also tables that are used for presenting results, summary tables and temporary tables; multidimensional clustering tables offer specific advantages in a warehouse environment, whereas partitioned tables let you spread data across more than one database partition.

3.1.1.1 Base tables

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

These types of tables hold persistent data. There are different kinds of base tables as outlined below.

3.1.1.2 Regular tables

Regular tables with indexes are the "general purpose" table choice.

3.1.1.3 Multidimensional clustering (MDC) tables

These types of tables are implemented as tables that are physically clustered on more than one key, or dimension, at the same time. MDC tables are used in data warehousing and large database environments. Clustering indexes on regular tables support single-dimensional clustering of data. MDC tables provide the benefits of data clustering across more than one dimension. MDC tables provide *guaranteed clustering* within the composite dimensions.

MDC introduces indexes that are block-based. "Block indexes" point to blocks or groups of records instead of to individual records. By physically organizing data in an MDC table into blocks according to clustering values, and then accessing these blocks using block indexes, MDC is able not only to address all of the drawbacks of clustering indexes, but to provide significant additional performance benefits.

MDC tables can coexist with partitioned tables and can themselves be partitioned tables.

3.1.1.4 Range-clustered tables (RCT)

These types of tables are implemented as sequential clusters of data that provide fast, direct access. Each record in the table has a predetermined record ID (RID) which is an internal identifier used to locate a record in a table. RCT tables are used where the data is tightly clustered across one or more columns in the table. The largest and smallest values in the columns define the range of possible values. You use these columns to access records in the table; this is the most optimal method of utilizing the predetermined record identifier (RID) aspect of RCT tables.

3.1.1.5 Partitioned tables

Partitioned tables use a data organization scheme in which table data is divided across multiple storage objects, called data partitions or ranges, according to values in one or more table partitioning key columns of the table.

3.1.1.6 Detached table

Just as you can add or attach new partitions to a partitioned table, you can also remove existing partitions. (Removed partitions become regular, stand-alone base tables.) Partitions can be removed by executing the ALTER TABLE statement with the DETACH PARTITION option specified.

3.1.1.7 Temporary tables

In DB2 you can globally declare temporary table to temporarily retain some rows for processing by subsequent SQL statements. A temporary table exists only until the thread is terminated (or sooner). It is not defined in the DB2 catalog, and neither its definition nor its contents are visible to other users. Multiple users can declare the same temporary table at the same time, with each independently working with their own copy.

The temporary table name can be any valid DB2 table name. The table qualifier, if provided, must be SESSION. If the qualifier is not provided, it is assumed to be SESSION. If the temporary table has been previously defined in this session, the WITH REPLACE clause can be used to override it. Alternatively, one can DROP the prior instance. An index can be defined on a global temporary table. The SESSION qualifier must be explicitly provided. Any column type can be used in the table, except for BLOB, CLOB, DBCLOB, LONG VARCHAR, LONG VARGRAPHIC, DATALINK, reference data types, and structured data types. You can choose to preserve or delete the rows in the table when a commit occurs (deletion is the default). Deleting the rows does not drop the table. Standard identity column definitions can be used if desired. Changes are not logged.

In SQL Server, temporary tables work differently. A temporary table can be either local or global. A local temporary table is visible only to the user who created that table, and is deleted after the user disconnects. If a local temporary table is created in a procedure, then it is automatically dropped after the process goes out of scope of the procedure. Global temporary tables are visible to all users and all sessions. Such tables are deleted after all users who are referencing them disconnect from the instance of SQL Server. Because global temporary tables are visible to all, you must use only local temporary tables.

When using local temporary tables in SQL Server, you must pay attention to the scope table, because the scope table can be used only at the level where it is created, or at deeper levels. In the case of a local table with a higher level, it will not be visible and an error message will result. On a deeper level, you can create a new local temporary table with the same name and apply the statements directly to this table, but in that case the previously created local temporary table becomes inaccessible.

3.1.1.7.1 Declaring a Global Temporary Table

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

Here is a DB2 example of declaring a global temporary table by listing the columns.

DB2 Example:

```
CREATE PROCEDURE DB2_TABLES.DECL_GLOBAL_TEMP_TAB
BEGIN
DECLARE GLOBAL TEMPORARY TABLE SESSION.TEMP_EMP
 (EMPNO CHAR(6) NOT NULL,
 SALARY DECIMAL(9, 2),
 BONUS DECIMAL(9, 2),
 COMM DECIMAL(9, 2))
ON COMMIT PRESERVE ROWS;
```

```
INSERT INTO SESSION.TEMP_EMP VALUES (1,1000,100,100);
END;
CALL DB2_TABLES.DECL_GLOBAL_TEMP_TAB;
```

Solution:

In SQL Server, generally you should not create a local temporary table in the same place where it is declared in DB2. You should create it as early as possible, preferably at the very beginning of the session. With this approach, the table scope will be broad enough to avoid the danger of destroying the table before some other code can reference it.

To convert from DB2, replace DECLARE with CREATE, and change the TEMPORARY keyword to a single pound sign (#) before the table name. Omit the database name.

SQL Server Example:

```
CREATE TABLE #TEMP_EMP
 (EMPNO CHAR(6) NOT NULL,
 SALARY DECIMAL(9, 2),
 BONUS DECIMAL(9, 2),
 COMM DECIMAL(9, 2))
```

DB2 Example:

```
CREATE PROCEDURE DB2_TABLES.DECL_TEMP_TAB_OPTIONS
BEGIN
DECLARE GLOBAL TEMPORARY TABLE TEMP_OPTIONS
(IDENTITY_2 INT NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 0,
INCREMENT BY 1, CACHE 7, MINVALUE 0, MAXVALUE 1000, CYCLE, ORDER),
 CLOB_V CLOB WITH DEFAULT EMPTY_CLOB(),
 DBCLOB_V DBCLOB WITH DEFAULT EMPTY_DBCLOB(),
 BLOB_V BLOB WITH DEFAULT EMPTY_BLOB(),
 NCLOB_V NCLOB WITH DEFAULT EMPTY_NCLOB(),
 USER_DEFAULT VARCHAR(100) DEFAULT 'ANNA',
 SYS_DEF_DATE DATE DEFAULT,
 SYS_DEF_VARCH VARCHAR (100) DEFAULT,
 SYS_DEF_INT INTEGER DEFAULT,
 CURRENT_DATE_V DATE WITH DEFAULT CURRENT_DATE);
END;
```

SQL Server Example:

```
CREATE PROCEDURE DBO.DECL_TEMP_TAB_OPTIONS AS
       CREATE TABLE #TEMP_OPTIONS
 (IDENTITY_2 INT NOT NULL IDENTITY (1,1),
       CLOB_V VARCHAR(MAX) DEFAULT NULL,
       DBCLOB_V NVARCHAR(MAX) DEFAULT NULL,
       BLOB_V VARBINARY(MAX) DEFAULT NULL,
       NCLOB_V NVARCHAR(MAX) DEFAULT NULL,
       USER_DEFAULT VARCHAR(100) DEFAULT 'ANNA',
 SYS_DEF_DATE DATE DEFAULT CURRENT_TIMESTAMP,
 SYS_DEF_VARCH VARCHAR (100) DEFAULT '',
 SYS_DEF_INT INTEGER DEFAULT 0,
       CURRENT_DATE_V DATE DEFAULT CURRENT_TIMESTAMP)
```

3.1.1.7.2 Creating a Temporary Table with Defined Columns

In DB2, in a temporary table, set LIKE to the name of a table, view, or nickname to specify that the columns of the temporary table have exactly the same names and descriptions as the columns of the identified table or view. The name specified after LIKE must identify a table, view, or nickname that exists in the catalog or a

declared temporary table. A typed table or typed view cannot be specified. EXCLUDING COLUMN DEFAULTS option specify that column defaults are not copied from the source result table definition.

DB2 Example:

```
DECLARE GLOBAL TEMPORARY TABLE SESSION.FRED
LIKE STAFF EXCLUDING COLUMN DEFAULTS
ON COMMIT PRESERVE ROWS;
```

Solution:

In SQL Server, you can emulate a temporary table with defined columns by using the statement SELECT * INTO.

SQL Server Example:

```
SELECT * INTO #FRED
FROM STAFF
WHERE 1 = 2
```

3.1.1.7.3 Creating a Temporary Table with Defined Columns and Defaults

In DB2, in a temporary table, set INCLUDING COLUMN DEFAULTS to specify that the column defaults for each updatable column of the source result table definition are copied. Columns that are not updatable will not have a default defined in the corresponding column of the created table.

DB2 Example:

```
DECLARE GLOBAL TEMPORARY TABLE SESSION.FRED
LIKE STAFF INCLUDING COLUMN DEFAULTS
ON COMMIT PRESERVE ROWS;
```

Solution:

In SQL Server, you can emulate a temporary table with defined columns by using the statement CREATE TABLE and defaults for the columns.

SQL Server Example:

```
CREATE TABLE #FRED
(DEPT SMALLINT NOT NULL DEFAULT 1
,AVG_SALARY DECIMAL(7,2) NOT NULL
,NUM_EMPS SMALLINT NOT NULL)
```

3.1.1.7.4 Creating a Temporary Table with Columns Returned by the SELECT statement

Here is a DB2 example that shows a temporary table defined to have a set of columns that are returned by a particular SELECT statement. The statement is not actually run at definition time, so any predicates provided are irrelevant.

DB2 Example:

```
DECLARE GLOBAL TEMPORARY TABLE SESSION.FRED AS
(SELECT DEPT
 ,MAX (ID) AS MAX_ID
 ,SUM (SALARY) AS SUM_SAL
FROM STAFF
```

```
WHERE NAME <> 'TOM'
GROUP BY DEPT)
DEFINITION ONLY;
```

Solution:

In SQL Server, you can emulate a temporary table with columns returned by using the statement SELECT … INTO.

SQL Server Example:

```
SELECT DEPT
 ,MAX (ID) AS MAX_ID
 ,SUM (SALARY) AS SUM_SAL
INTO #FRED
FROM STAFF
WHERE NAME <> 'TOM' AND 1=2
GROUP BY DEPT
```

3.1.1.7.5 Creating an Index for a Temporary Table

In DB2, an index can be added to a temporary table in order to improve performance and to enforce uniqueness. In this example, column defaults are copied, as in section 3.8.3.

DB2 Example:

```
DECLARE GLOBAL TEMPORARY TABLE SESSION.FRED
LIKE STAFF INCLUDING COLUMN DEFAULTS
ON COMMIT PRESERVE ROWS;

CREATE UNIQUE INDEX SESSION.FRED ON SESSION.FRED (ID);

INSERT INTO SESSION.FRED
SELECT *
FROM STAFF
WHERE ID < 200;

SELECT COUNT(*)
FROM SESSION.FRED;
```

Solution:

In SQL Server, emulation is identical to DB2 when you create an index (except for changing the TEMPORARY keyword to a pound sign before the table name, as noted above).

SQL Server Example:

```
CREATE TABLE #FRED
(DEPT SMALLINT NOT NULL DEFAULT 1
,AVG_SALARY DECIMAL(7,2)NOT NULL
,NUM_EMPS SMALLINT NOT NULL)

CREATE UNIQUE INDEX #FRED ON #FRED(ID)

INSERT INTO #FRED
SELECT *
FROM STAFF
WHERE ID < 200;
```

```
SELECT COUNT(*)
FROM #FRED;
```

3.1.1.7.6 Reusing a Temporary Table

In DB2, you must drop a temporary table to reuse the name of that table.

DB2 Example:

```
DECLARE GLOBAL TEMPORARY TABLE SESSION.FRED
(DEPT SMALLINT NOT NULL
,AVG_SALARY DEC (7,2) NOT NULL
,NUM_EMPS SMALLINT NOT NULL)
ON COMMIT PRESERVE ROWS;
/
INSERT INTO SESSION.FRED
SELECT DEPT
 ,AVG(SALARY)
 ,COUNT(*)
FROM STAFF
GROUP BY DEPT;
/
SELECT COUNT(*)
FROM SESSION.FRED;
/
DROP TABLE SESSION.FRED;
/
DECLARE GLOBAL TEMPORARY TABLE SESSION.FRED
(DEPT SMALLINT NOT NULL)
ON COMMIT DELETE ROWS;
/
SELECT COUNT(*)
FROM SESSION.FRED;
```

Solution:

In SQL Server, emulation is identical to all earlier examples in this section. Here, you must remove the local temporary table before you create a new local temporary table of the same name.

SQL Server Example:

```
CREATE TABLE #FRED
(DEPT SMALLINT NOT NULL
 ,AVG_SALARY DECIMAL(7,2) NOT NULL
 ,NUM_EMPS SMALLINT NOT NULL)

INSERT INTO #FRED
SELECT DEPT
 ,AVG(SALARY)
 ,COUNT(*)
FROM STAFF
GROUP BY DEPT

SELECT COUNT(*)
FROM #FRED;

DROP TABLE #FRED;
```

```
CREATE TABLE #FRED
(DEPT SMALLINT NOT NULL)

SELECT COUNT(*)
FROM #FRED
```

3.1.1.7.7 Declaring a Temporary Table

In DB2, in the case that a declared global temporary table already exists with the specified name, you can set WITH REPLACE to specify that the existing table is replaced with the temporary table defined by this statement (and that all rows of the existing table are deleted).

DB2 Example:

```
DECLARE GLOBAL TEMPORARY TABLE SESSION.FRED
(DEPT SMALLINT NOT NULL
,AVG_SALARY DEC(7,2) NOT NULL
,NUM_EMPS SMALLINT NOT NULL)
ON COMMIT PRESERVE ROWS WITH REPLACE;
```

Solution:

In SQL Server, you should first check whether this table already exists, and if it does, you should drop and re-create it as shown in the examples of sections 3.8.1 through 3.8.6.

SQL Server Example:

```
IF OBJECT_ID('TEMPDB..#FRED') IS NOT NULL
 DROP TABLE #FRED

CREATE TABLE #FRED
(DEPT SMALLINT NOT NULL
 ,AVG_SALARY DECIMAL(7,2) NOT NULL
 ,NUM_EMPS SMALLINT NOT NULL)
```

3.1.1.7.8 Deleting Rows After Commit

In DB2, you can set ON COMMIT DELETE ROWS to specify that all rows of the table will be deleted if no WITH HOLD cursor is open on the table. ON COMMIT DELETE ROWS is default option of this statement.

DB2 Example:

```
DECLARE GLOBAL TEMPORARY TABLE SESSION.FRED
(DEPT SMALLINT NOT NULL
 ,AVG_SALARY DEC(7,2) NOT NULL
 ,NUM_EMPS SMALLINT NOT NULL)
ON COMMIT DELETE ROWS;
```

Solution:

In SQL Server, to emulate this option you must before every COMMIT statement write the following code:

SQL Server Example:

```
DELETE FROM #FRED
```

3.1.1.7.9 CREATE GLOBAL TEMPORARY TABLE statement

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

The CREATE GLOBAL TEMPORARY TABLE statement creates a description of a temporary table at the current server. Each session that selects from a created temporary table retrieves only rows that the same session has inserted. When the session terminates, the rows of the table associated with the session are deleted.

DB2 Example:

```
CREATE GLOBAL TEMPORARY TABLE DB2_TABLES.GLOBAL_TMP_TAB
 (TMPDEPTNO CHAR(3) NOT NULL,
 TMPDEPTNAME VARCHAR(36) NOT NULL,
 TMPMGRNO CHAR(6),
 TMPLOCATION CHAR(16));
```

Solution:

You could use a simple table in MSSQL Server.

SQL Server Example:

```
CREATE TABLE DBO.GLOBAL_TMP_TAB
 (TMPDEPTNO CHAR(3) NOT NULL,
 TMPDEPTNAME VARCHAR(36) NOT NULL,
 TMPMGRNO CHAR(6),
 TMPLOCATION CHAR(16));
```

3.1.1.8 Materialized query tables

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

Materialized Query Tables, or MQTs can be used to greatly improve the elegance and efficiency of DB2-based data warehouses. (Of course, MQTs are not solely for data warehousing, but they are most useful for static data.)

An MQT can be thought of as a view whose data is physically stored instead of virtually accessed when needed. Each MQT is defined as a SQL query, similar to a view. But the MQT pre-computes the query results and stores the data. Subsequent user queries that require the data can re-use the data from the MQT instead of re-computing it, which can save time and resources.

A summary table is a specialized type of materialized query table.

After the SELECT statement, there are several parameters that define the nature of the MQT. First of all, when you create an MQT there are several options available to specify how the data is to be populated and refreshed into the MQT. These parameters are:

DATA INITIALLY DEFERRED - Data is not inserted into the table as part of the CREATE TABLE statement. A REFRESH TABLE statement specifying the table-name is used to insert data into the table.

REFRESH - Indicates how the data in the table is maintained.

DEFERRED - The data in the table can be refreshed at any time using the REFRESH TABLE statement. The data in the table only reflects the result of the query as a snapshot at the time the REFRESH TABLE statement is processed. System-maintained materialized query tables defined with this attribute do not allow INSERT, UPDATE, or DELETE statements. User-maintained materialized query tables defined with this attribute do allow INSERT, UPDATE, or DELETE statements.

DB2 Example:

```
CREATE TABLE DB2_OBJECTS.A (A BIGINT, B VARCHAR(100));

CREATE TABLE DB2_OBJECTS.MQT_A AS (
```

```
 SELECT COUNT (*) AS CNT, B
       FROM DB2_OBJECTS.A
       GROUP BY B)
DATA INITIALLY DEFERRED REFRESH DEFERRED;

REFRESH TABLE DB2_MSSQL_UNITTEST.MQT_A;
```

IMMEDIATE - The changes made to the underlying tables as part of a DELETE, INSERT, or UPDATE are cascaded to the materialized query table. In this case, the content of the table, at any point-in-time, is the same as if the specified subselect is processed. Materialized query tables defined with this attribute do not allow INSERT, UPDATE, or DELETE.

DB2 Example:

```
CREATE TABLE DB2_OBJECTS.MQT_AI AS (
 SELECT COUNT (*) AS CNT, B
       FROM DB2_OBJECTS.A
       GROUP BY B)
DATA INITIALLY DEFERRED REFRESH IMMEDIATE;

REFRESH TABLE DB2_OBJECTS.MQT_AI;
```

Solution

You could use SQL Server Indexed views instead of Materialized Query Tables.

SQL Server Example:

```
CREATE TABLE DBO.A (A BIGINT, B VARCHAR(100));

CREATE VIEW DBO.MQT_A
WITH SCHEMABINDING AS (
 SELECT COUNT_BIG(*) AS CNT, B
       FROM DBO.A
       GROUP BY B);

CREATE UNIQUE CLUSTERED INDEX IDX_V1 ON MQT_A (B);
```

MAINTAINED BY SYSTEM: indicates that the MQT is maintained by the system. This option is the default and it means that the MQT does not allow LOAD, INSERT, UPDATE, or DELETE, or SELECT FOR UPDATE statements. The REFRESH TABLE statement is used to populate data in the MQT.

MAINTAINED BY USER - The data in the materialized query table is maintained by the user. The user is allowed to perform update, delete, or insert operations against user-maintained materialized query tables. The REFRESH TABLE statement, used for system-maintained materialized query tables, cannot be invoked against user-maintained materialized query tables. Only a REFRESH DEFERRED materialized query table can be defined as MAINTAINED BY USER.

DB2 Example:

```
CREATE TABLE DB2_OBJECTS.MQT_MU AS (
 SELECT SUM (A) AS SM, B
       FROM DB2_OBJECTS.A
                   GROUP BY B)
DATA INITIALLY DEFERRED REFRESH DEFERRED
ENABLE QUERY OPTIMIZATION
MAINTAINED BY USER;
SET INTEGRITY FOR DB2_OBJECTS.MQT_MU ALL IMMEDIATE UNCHECKED;

INSERT INTO DB2_OBJECTS.MQT_MU SELECT * FROM (
 SELECT SUM (A) AS SM, B
       FROM DB2_OBJECTS.A
 GROUP BY B) SQ;
```

3.1.1.8.1 Materialized query table restrictions

Links: DB2 for Linux UNIX and Windows 10.5.0

The “fullselect” statements that form part of the definition of materialized query tables (MQTs) is subject to the following restrictions.

- Every select element must have a name.
- A fullselect must not reference any of the following object types:
  - materialized query tables,
  - staging tables,
  - declared global temporary tables,
  - created global temporary tables,
  - typed tables,
  - system catalog tables,
  - views that violate any MQT restrictions,
  - protected tables,
  - nicknames that are created with the DISALLOW CACHING clause of the CREATE NICKNAME or ALTER NICKNAME statements,
  - views that directly or indirectly depend on protected tables.
- A fullselect must not contain any column references or expressions of the following data types:
  - LOB,
  - LONG,
  - DATALINK,
  - XML,
  - reference,
  - user defined structured type,
  - any distinct type that is based on these data types.
- A fullselect must not contain any column references or expressions or functions that:
  - depend on the physical characteristics of the data. For example, DBPARTITIONNUM, HASHEDVALUE, and RID_BIT, RID.
  - depend on changes to the data. For example, a row change expression or a row change timestamp column.
  - are defined as EXTERNAL ACTION.
  - are defined as LANGUAGE SQL, CONTAINS SQL, READS SQL DATA, or MODIFIES SQL DATA.
- A fullselect must not include a CONNECT BY clause.
- When MAINTAINED BY FEDERATED_TOOL is specified in the CREATE TABLE statement, the SELECT clause must not contain a reference to a base table.
- When REFRESH IMMEDIATE is specified:
  - the CREATE MQT statement must not contain duplicate grouping sets.
  - at least one unique key from each table that is referenced must be in the select list.
  - the fullselect must be a subselect. The exception is that UNION ALL is supported in the input table expression of a GROUP BY clause.
  - the input table expressions of a UNION ALL or a JOIN must not contain aggregate functions.
- When REFRESH IMMEDIATE is specified, the fullselect must not contain:
  - a reference to a nickname.
  - a SELECT DISTINCT statement.
  - a reference to a special register.
  - a built-in function that depends on the value of a special register.
  - a reference to a global variable.
  - functions that are not deterministic.
  - OLAP functions.
  - sampling functions.
  - text functions.

– any expressions that use the result of aggregate functions.
– an aggregate function without the fullselect also containing a GROUP BY clause.
– a recursive common table expression.
– subqueries.

- When REFRESH IMMEDIATE is specified, and the fullselect contains a GROUP BY clause:
  – the select list must contain COUNT() or COUNT_BIG().
  – for each nullable column C, if the select list contains SUM(C), then COUNT(C) is also required.
  – you must include the SUM(), or GROUPING() aggregate function. No other aggregate function can be included.
  – the HAVING clause must not be specified.
  – in a partitioned database environment, the GROUP BY columns must contain the partitioning key of the materialized query table.
  – nesting of aggregate functions is not allowed.
- When REFRESH IMMEDIATE is specified, and the FROM clause references more than one table, only an inner join, without using the explicit INNER JOIN syntax, is supported.
- When REPLICATED is specified:
  – aggregate functions and the GROUP BY clause are not allowed.
  – the MQT must reference only a single table. It cannot include a join, union, or subquery.
  – the PARTITIONING KEY clause must not be specified.
  – unique indexes are not allowed for system maintained MQTs.

3.1.1.9 3.1.1.9 Typed Table

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

Typed tables are used to store instances of objects whose characteristics are defined with the CREATE TYPE statement. You can create a typed table using a variant of the CREATE TABLE statement.

3.1.1.9.1 Naming the Object Identifier

Because typed tables contain objects that can be referenced by other objects, every typed table has an *object identifier* column as its first column. You can name the object identifier column using the REF IS … USER GENERATED clause. In this case, the column is named Oid. The USER GENERATED part of the REF IS clause indicates that you must provide the initial value for the object identifier column of each newly inserted row. It is common practice in object-oriented design to completely separate the data from the object identifier. For that reason, you cannot update the value of the object identifier after you insert the object identifier.

DB2 Example:

```
CREATE TYPE DB2_OBJECTS.PERSON_T AS (NAME VARCHAR(20), AGE INT, DOB DATE)
INSTANTIABLE REF USING VARCHAR(13) FOR BIT DATA MODE DB2SQL;

CREATE TABLE DB2_OBJECTS.PERSON OF DB2_OBJECTS.PERSON_T (REF IS OID USER
GENERATED)
```

SQL Server Example:

```
CREATE TYPE DBO.PERSON_T AS TABLE (NAME VARCHAR(20), AGE INT, DOB DATE);

CREATE TABLE TEST_FUNCTIONS.DBO.PERSON (OID INTEGER PRIMARY KEY, NAME
VARCHAR(20), AGE INT, DOB DATE);
```

3.1.1.9.3 Typed table hierarchy

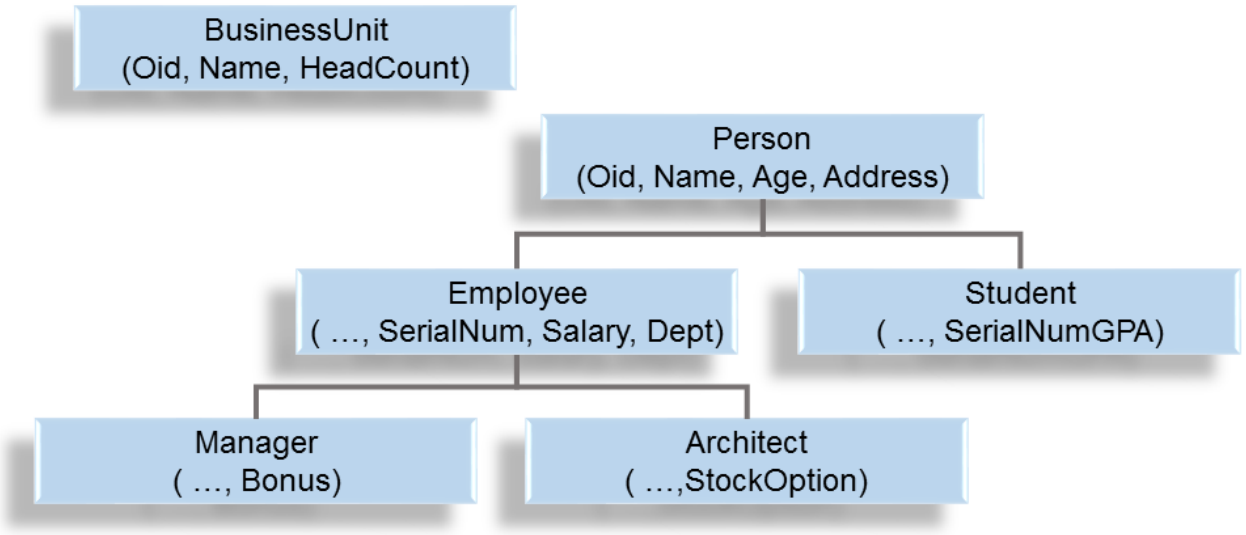
You can also create a hierarchy of typed tables that is based on a hierarchy of structured types. To store instances of subtypes in database tables, you must create a corresponding table hierarchy.

The Person typed table is of type Person_t. To store instances of the subtypes of employees and students, it is necessary to create the subtables of the Person table, Employee and Student. The two additional subtypes of Employee_t also require tables. Those subtables are named Manager and Architect. Just as a subtype inherits the attributes of its supertype, a subtable inherits the columns of its supertable, including the object identifier column. A subtable must reside in the same schema as its supertable.

Rows in the Employee subtable, therefore, will have a total of seven columns: Oid, Name, Age, Address, SerialNum, Salary, and Dept.

A SELECT, UPDATE, or DELETE statement that operates on a supertable automatically operates on all its subtables as well. For example, an UPDATE statement on the Employee table might affect rows in the Employee, Manager, and Architect tables, but an UPDATE statement on the Manager table can only affect Manager rows.

Typed table hierarchy



DB2 Example:

```
CREATE TABLE DB2_OBJECTS.BUSINESSUNIT OF DB2_OBJECTS.BUSINESSUNIT_T (REF
IS OID USER GENERATED);

CREATE TABLE DB2_OBJECTS.PERSON OF DB2_OBJECTS.PERSON_T (REF IS OID USER
GENERATED);

CREATE TABLE DB2_OBJECTS.EMPLOYEE OF DB2_OBJECTS.EMPLOYEE_T UNDER
DB2_OBJECTS.PERSON INHERIT SELECT PRIVILEGES
(SERIALNUM WITH OPTIONS NOT NULL, DEPT WITH OPTIONS SCOPE
DB2_OBJECTS.BUSINESSUNIT);

CREATE TABLE DB2_OBJECTS.STUDENT OF DB2_OBJECTS.STUDENT_T UNDER
DB2_OBJECTS.PERSON INHERIT SELECT PRIVILEGES;

CREATE TABLE DB2_OBJECTS.MANAGER OF DB2_OBJECTS.MANAGER_T UNDER
DB2_OBJECTS.EMPLOYEE INHERIT SELECT PRIVILEGES;

CREATE TABLE DB2_OBJECTS.ARCHITECT OF DB2_OBJECTS.ARCHITECT_T UNDER
DB2_OBJECTS.EMPLOYEE INHERIT SELECT PRIVILEGES;
```

This section describes the differences between DB2 and SQL Server in creating tables, and covers a few incompatible clauses of CREATE TABLE statement.

3.1.1.10 LIKE Clause

In DB2, the LIKE clause specifies that the columns of the created table have exactly the same name and description as the columns of the identified source table, view, or nickname. The name specified after LIKE must identify a table, view, nickname, or temporary table. The LIKE clause can also be used for defining default constraints and identity columns in the target table. This is controlled by the clauses INCLUDING/EXCLUDING COLUMN DEFAULTS and INCLUDING/EXCLUDING IDENTITY COLUMN ATTRIBUTES.

DB2 Example 1:

```
CREATE TABLE T1
( C1 INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY
 ( START WITH 1 INCREMENT BY 1),
 C2 VARCHAR(50) WITH DEFAULT 'A');

CREATE TABLE T2 LIKE T1
INCLUDING IDENTITY COLUMN ATTRIBUTES;
```

DB2 Example 2:

```
CREATE TABLE T1
(C1 INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY
 ( START WITH 1 INCREMENT BY 1),
 C2 VARCHAR(50) WITH DEFAULT 'A');

CREATE TABLE T2 LIKE T1
INCLUDING COLUMN DEFAULTS
INCLUDING IDENTITY COLUMN ATTRIBUTES
```

Solution:

SQL Server does not have similar functionality. Use the default CREATE TABLE or SELECT…INTO statement to create a table that will have columns with the same name and description as the columns of another table. The SQL Server statement SELECT…INTO allows you to create IDENTITY fields but does not allow you to add defaults and calculated fields. Thus you can convert SELECT…INTO statements, which do not contain the clause INCLUDING/EXCLUDING COLUMN DEFAULTS.

SQL Server Example 1:

```
CREATE TABLE T1
( C1 INT IDENTITY(1,1),
 C2 VARCHAR(50));

select * into t2 from t1;
```

SQL Server Example 2:

```
CREATE TABLE T1
(C1 INT IDENTITY(1,1),
 C2 VARCHAR(50) DEFAULT 'A');

CREATE TABLE T2
( C1 INT IDENTITY(1,1),
 C2 VARCHAR(50) DEFAULT 'A');
```

3.1.1.11 COMPRESS Clause

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

You could use **row compression only, value compression only, or both types of compression**.

Row compression will almost always yield benefits in terms of storage savings, as it attempts to replace data patterns that span multiple columns within a row with shorter symbol strings.

In DB2, the COMPRESS clause specifies whether data compression applies to the rows of the table.

DB2 Example:

```
CREATE TABLE DB2_OBJECTS.COMPRESS_TAB
(C1 INT, C2 VARCHAR(50) )
COMPRESS YES
```

Solution:

There is no similar clause in SQL Server. To compress data in a SQL Server table, use the WITH DATA_COMPRESSION = page statement.

SQL Server Example:

```
CREATE TABLE T1
(C1 INT, C2 NVARCHAR(50) )
WITH (DATA_COMPRESSION = PAGE);
```

3.1.1.12 VALUE COMPRESSION Clause

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

Value compression can offer savings when you have a many rows with columns that contain the same value, or when you have columns that contain the default value for the data type of the column.

In DB2, the VALUE COMPRESSION clause determines the row format that is to be used. Each data type has a different byte count, depending on the row format that is used. The NULL value is stored using three bytes. Whether a column is defined as nullable has no effect on the row size calculation. The zero-length data values for columns whose data type is VARCHAR, VARGRAPHIC, LONG VARCHAR, LONG VARGRAPHIC, CLOB, DBCLOB, BLOB, or XML are to be stored using two bytes only, which is less than the storage required when VALUE COMPRESSION is not active.

DB2 Example:

```
CREATE TABLE DB2_OBJECTS.VALUE_COMPRESS
(C1 INT, C2 VARCHAR(50) )
VALUE COMPRESSION
```

Solution:

In SQL Server, you can emulate this clause by using the WITH DATA_COMPRESSION = row statement.

SQL Server Example:

```
CREATE TABLE T2
(C1 INT, C2 NVARCHAR(50) )
WITH (DATA_COMPRESSION = ROW);
```

When value compression is enabled, you can also specify that columns that assume the system default value for their data types can be further compressed with the COMPRESS SYSTEM DEFAULT option.

DB2 Example:

```
CREATE TABLE DB2_OBJECTS.VAL_SYS_COMPRESS
(DEPTNO CHAR(3) NOT NULL,
 DEPTNAME VARCHAR(36) NOT NULL,
 EMPNO CHAR(6) NOT NULL,
 SALARY DECIMAL(9,2) NOT NULL WITH DEFAULT COMPRESS SYSTEM DEFAULT)
 VALUE COMPRESSION;
```

Solution:

In SQL Server, you can emulate this clause by using the WITH DATA_COMPRESSION = row statement.

SQL Server Example:

```
CREATE TABLE VAL_SYS_COMPRESS
(DEPTNO CHAR(3) NOT NULL,
 DEPTNAME VARCHAR(36) NOT NULL,
 EMPNO CHAR(6) NOT NULL,
 SALARY DECIMAL(9,2) NOT NULL DEFAULT 0)
 WITH (DATA_COMPRESSION = ROW);
```

Both value and row compression could be used.

DB2 Example:

```
CREATE TABLE DB2_OBJECTS.BOTH_COMPRESS
(DEPTNO CHAR(3) NOT NULL,
 DEPTNAME VARCHAR(36) NOT NULL)
 VALUE COMPRESSION COMPRESS YES;
```

Solution:

In SQL Server, you can emulate this clause by using the WITH DATA_COMPRESSION = page statement.

SQL Server Example:

```
CREATE TABLE BOTH_COMPRESS
(DEPTNO CHAR(3) NOT NULL,
 DEPTNAME VARCHAR(36) NOT NULL)
 WITH (DATA_COMPRESSION = PAGE);
```

3.1.1.13 CCSID [ASCII / UNICODE] Clauses

In DB2, the CCSID clause specifies the encoding scheme for string data stored in the table. If the CCSID clause is not specified, the default is CCSID UNICODE for Unicode databases, and CCSID ASCII for all other databases.

ASCII specifies that string data is encoded in the database code page. If the database is a Unicode database, CCSID ASCII cannot be specified.

UNICODE specifies that string data is encoded in Unicode. If the database is a Unicode database, character data is in UTF-8, and graphic data is in UCS-2. If the database is not a Unicode database, character data is in UTF-8.

DB2 Example 1:

```
CREATE TABLE DB2_TABLES.CCSID_ASCII (C1 INT, C2 VARCHAR(50) ) CCSID ASCII;
```

DB2 Example 2:

```
CREATE TABLE DB2_TABLES.CCSID_UNICODE(C1 INT, C2 VARCHAR(50)) CCSID
UNICODE;
```

Solution:

SQL Server supports NCHAR, NVARCHAR, and NTEXT data types for storing Unicode character data.

SQL Server Example 1:

```
CREATE TABLE T1
(C1 INT, C2 VARCHAR(50) );
```

SQL Server Example 2:

```
CREATE TABLE T1
(C1 INT, C2 NVARCHAR(50) );
```

3.1.1.14 Generated-clause

GENERATED

Specifies that DB2 generates values for the column. GENERATED must be specified if the column is to be considered an identity column or a row change timestamp column, row-begin column, row-end column, transaction-start-ID column, or generated expression column.

ALWAYS

Specifies that a value will always be generated for the column when a row is inserted into the table, or whenever the result value of the generation-expression changes. The result of the expression is stored in the table. GENERATED ALWAYS is the recommended value unless data propagation or unload and reload operations are being done. GENERATED ALWAYS is the required value for generated columns.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE TABLE DB2_TABLES.GEN_ALW_STMT (ID INT, GENERATED_EXPRESSION
VARCHAR(100) GENERATED ALWAYS AS (CASE WHEN 1=1 THEN 'WORLD' ELSE
'NOTHING' END));
```

Solution:

You could replace GENERATED ALWAYS AS clause with SQL Server Computed Columns.

.

SQL Server Example :

```
CREATE TABLE GEN_ALW_ST (ID INT, GENERATED_EXPRESSION AS
(CASE WHEN 1=1 THEN 'WORLD' ELSE 'NOTHING' END))
```

System default - special register value as the default for this column when no other value is provided.
Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014

DB2 Example:

```
CREATE TABLE DB2_TABLES.SYSTEM_DEFAULT (ID INT, SYS_DEF_DATE DATE DEFAULT,
SYS_DEF_VARCH VARCHAR(100) DEFAULT, SYS_DEF_INT INTEGER DEFAULT);
```

Solution:
You could replace DEFAULT clause with SQL Server DEFAULT clause plus value by default depending on column data type: DATE equal to GETDATE(),VARCHAR equal to '', INTEGER equal to 0.

SQL Server Example:

```
CREATE TABLE SYSTEM_DEFAULT
(ID INT, SYS_DEF_DATE DATE DEFAULT GETDATE(), SYS_DEF_VARCH VARCHAR(100)
DEFAULT '', SYS_DEF_INT INTEGER DEFAULT 0)
```

BY DEFAULT

Specifies that DB2 will generate a value for the column when a row is inserted, or updated specifying the DEFAULT clause, unless an explicit value is specified. BY DEFAULT is the recommended value when using data propagation or performing an unload and reload operation.
Although not explicitly required, to ensure uniqueness of the values, define a unique single-column index on generated IDENTITY columns.

DB2 Example:

```
CREATE TABLE DB2_TABLES.GEN_ALW_ST (ID INT, GENERATED_EXPRESSION INT
GENERATED BY DEFAULT AS IDENTITY (START WITH 0, INCREMENT BY 1));
```

Solution:
You could replace GENERATED BY DEFAULT AS IDENTITY clause with SQL Server IDENTITY clause.

SQL Server Example:

```
CREATE SEQUENCE SEQ_IDENTITY
 AS INT
 START WITH 0
 INCREMENT BY 1
 MINVALUE 0
 MAXVALUE 99999999999999999999999999
 /
 CREATE TABLE SEQ_IDEN
(ID INT, SEQ_IDENTITY INT DEFAULT NEXT VALUE FOR SEQ_IDENTITY) AS IDENTITY
```

Specifies that the column is to be the identity column for this table. A table can only have a single identity column (SQLSTATE 428C1). The IDENTITY keyword can only be specified if the data type associated with the column is an exact numeric type with a scale of zero, or a user-defined distinct type for which the source type is an exact numeric type with a scale of zero (SQLSTATE 42815). SMALLINT, INTEGER, BIGINT, or DECIMAL with a scale of zero, or a distinct type based on one of these types, are considered exact numeric types. By contrast, single- and double-precision floating points are considered approximate numeric data types. Reference types, even if represented by an exact numeric type, cannot be defined as identity columns.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE TABLE DB2_TABLES.IDENT (ID INT, IDENT INTEGER NOT NULL GENERATED
ALWAYS AS IDENTITY (START WITH 0, INCREMENT BY 1, NO CACHE, NO MINVALUE,
MAXVALUE 0, NO CYCLE, NO ORDER));
```

Solution:
You could replace GENERATED ALWAYS AS IDENTITY clause with SQL Server IDENTITY clause.

SQL Server Example:

```
CREATE TABLE IDENT (ID INT, IDENT INTEGER NOT NULL IDENTITY (1,1))
```

FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP

Specifies that the column is a timestamp column for the table. A value is generated for the column in each row that is inserted, and for any row in which any column is updated. The value that is generated for a ROW CHANGE TIMESTAMP column is a timestamp that corresponds to the insert or update time for that row. If multiple rows are inserted or updated with a single statement, the value of the ROW CHANGE TIMESTAMP column might be different for each row.
A table can only have one ROW CHANGE TIMESTAMP column. A ROW CHANGE TIMESTAMP column cannot have a DEFAULT clause. NOT NULL must be specified for a ROW CHANGE TIMESTAMP column.

DB2 Example:

```
CREATE TABLE DB2_TABLES.ROW_CHANGE_TIMESTAMP (ID INT, GEN_DEF TIMESTAMP
NOT NULL GENERATED BY DEFAULT FOR EACH ROW ON UPDATE AS ROW CHANGE
TIMESTAMP);
```

Solution:

For INSERT statement in MS SQL Server you could use DEFAULT CURRENT_TIMESTAMP for column.
For UPDATE statement in MS SQL Server you could use additional column in SET statement and set appropriate column with CURRENT_TIMESTAMP function.

SQL Server Example:

```
CREATE TABLE DBO.ROW_CHANGE_TIMESTAMP (ID INT, CT DATETIME DEFAULT
CURRENT_TIMESTAMP);

UPDATE DBO.ROW_CHANGE_TIMESTAMP_ SET ID=2, CT = CURRENT_TIMESTAMP;
```

Note: Azure SQL DB doesn't support sequence objects

3.1.1.15 Constraints

3.1.1.15.1 PRIMARY KEY

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

Both DB2 and SQL Server support column constraints on tables.

DB2 Example:

```
CREATE TABLE DB2_TABLES.ON_DEL_NOACT_FK (ID_FK INT NOT NULL PRIMARY KEY,
VAL VARCHAR (100));
```

Solution:

In SQL Server, the only difference in primary key constraint syntax is that SQL Server does not require the NOT NULL keyword to be added.

SQL Server Example:

```
CREATE TABLE EMPLOYEE
(EMPNO SMALLINT IDENTITY PRIMARY KEY,
 NAME CHAR(30),
 SALARY DECIMAL(5,2))
```

3.1.1.15.2 ON DELETE, ON UPDATE RULE-CLAUSE

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

Specifies what action to take on dependent tables:

- ON DELETE;

Specifies what action is to take place on the dependent tables when a row of the parent table is deleted. There are four possible actions:

  - NO ACTION (default);
  - RESTRICT;
  - CASCADE;
  - SET NULL.

- ON UPDATE.

Specifies what action is to take place on the dependent tables when a row of the parent table is updated. There are two possible actions:

  - NO ACTION (default);
  - RESTRICT.

The delete rule applies when a row of T2 is the object of a DELETE or propagated delete operation and that row has dependents in T1. Let p denote such a row of T2.

- If C or NO ACTION is specified, an error occurs and no rows are deleted.
- If CASCADE is specified, the delete operation is propagated to the dependents of p in T1.
- If SET NULL is specified, each nullable column of the foreign key of each dependent of p in T1 is set to null.

The use of NO ACTION or RESTRICT as delete or update rules for referential constraints determines when the constraint is enforced. A delete or update rule of RESTRICT is enforced *before* all other constraints, including those referential constraints with modifying rules such as CASCADE or SET NULL. A delete or update rule of NO ACTION is enforced after other referential constraints. One example where different behavior is evident involves the deletion of rows from a view that is defined as a UNION ALL of related tables.

Solution:

In SQL Server you could use ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } statement which specifies what action happens to rows in the table created, if those rows have a referential relationship and the referenced row is deleted from the parent table. The default is NO ACTION.

DB2 Example:

```
ON DELETE; NO ACTION (DEFAULT);

CREATE TABLE DB2_TABLES.ON_DEL_NOACT_PK (ID INT, ID_FK INT, VAL VARCHAR
(100), CONSTRAINT FK_NOACT FOREIGN KEY (ID_FK) REFERENCES
DB2_TABLES.ON_DEL_NOACT_FK ON DELETE NO ACTION);
CREATE TABLE DB2_TABLES.ON_DEL_NOACT_FK (ID_FK INT NOT NULL PRIMARY KEY,
VAL VARCHAR (100));
```

DB2 Example:

```
ON DELETE; RESTRICT;

CREATE TABLE DB2_TABLES.ON_DEL_RESTRICT_PK (ID INT, ID_FK INT, VAL VARCHAR
(100), CONSTRAINT FK_RESTR FOREIGN KEY (ID_FK) REFERENCES
DB2_TABLES.ON_DEL_RESTRICT_FK ON DELETE RESTRICT);
CREATE TABLE DB2_TABLES.ON_DEL_RESTRICT_FK (ID_FK INT NOT NULL PRIMARY
KEY, VAL VARCHAR (100));
```

Solution:

NO ACTION

The Database Engine raises an error and the delete action on the row in the parent table is rolled back.

SQL Server Example:

```
CREATE TABLE DBO.ON_DEL_NOACT_PK (ID INT, ID_FK INT, VAL VARCHAR (100),
CONSTRAINT FK_NOACT FOREIGN KEY (ID_FK) REFERENCES DBO.ON_DEL_NOACT_FK ON
DELETE NO ACTION);
CREATE TABLE DBO.ON_DEL_NOACT_FK (ID_FK INT NOT NULL PRIMARY KEY, VAL
VARCHAR (100));
```

DB2 Example:

```
ON DELETE; CASCADE;

CREATE TABLE DB2_TABLES.ON_DEL_CASCADE_PK (ID INT, ID_FK INT, VAL VARCHAR
(100), CONSTRAINT FK_CASC FOREIGN KEY (ID_FK) REFERENCES
DB2_TABLES.ON_DEL_CASCADE_FK ON DELETE CASCADE);
CREATE TABLE DB2_TABLES.ON_DEL_CASCADE_FK (ID_FK INT NOT NULL PRIMARY KEY,
VAL VARCHAR (100));
```

Solution:

CASCADE

Corresponding rows are deleted from the referencing table if that row is deleted from the parent table.

SQL Server Example:

```
CREATE TABLE DBO.ON_DEL_CASCADE_PK (ID INT, ID_FK INT, VAL VARCHAR (100),
CONSTRAINT FK_CASC FOREIGN KEY (ID_FK) REFERENCES DBO.ON_DEL_CASCADE_FK ON
DELETE CASCADE);
CREATE TABLE DBO.ON_DEL_CASCADE_FK (ID_FK INT NOT NULL PRIMARY KEY, VAL
VARCHAR (100));
```

DB2 Example:

```
ON DELETE; SET NULL;

CREATE TABLE DB2_TABLES.ON_DEL_SETNULL_PK (ID INT, ID_FK INT, VAL VARCHAR
(100), CONSTRAINT FK_NULL FOREIGN KEY (ID_FK) REFERENCES
DB2_TABLES.ON_DEL_SETNULL_FK ON DELETE SET NULL);
CREATE TABLE DB2_TABLES.ON_DEL_SETNULL_FK (ID_FK INT NOT NULL PRIMARY KEY,
VAL VARCHAR (100));
```

Solution:

SET NULL

All the values that make up the foreign key are set to NULL if the corresponding row in the parent table is deleted. For this constraint to execute, the foreign key columns must be nullable.

SQL Server Example:

```
CREATE TABLE DBO.ON_DEL_SETNULL_PK (ID INT, ID_FK INT, VAL VARCHAR (100),
CONSTRAINT FK_NULL FOREIGN KEY (ID_FK) REFERENCES DBO.ON_DEL_SETNULL_FK ON
DELETE SET NULL);
CREATE TABLE DBO.ON_DEL_SETNULL_FK (ID_FK INT NOT NULL PRIMARY KEY, VAL
VARCHAR (100));
```

In DB2 ON UPDATE - Specifies what action is to take place on the dependent tables when a row of the parent table is updated. The clause is optional. ON UPDATE NO ACTION is the default and ON UPDATE RESTRICT is the only alternative. In MS SQL Server you could use ON UPDATE { NO ACTION } – statement.

3.1.2 Migrating Views – CREATE VIEW Statement

This section describes the options of DB2 views that are incompatible with SQL Server views.

3.1.2.1 WITH LOCAL CHECK OPTION Clause

In DB2, the WITH LOCAL CHECK OPTION constraint on a view means that the search condition of the view is applied as a constraint for an insert or update of the view, or for any other view that is dependent on this view. In contrast to a view with the cascaded check option, a view with the local check option doesn't inherit the search conditions as constraints from any updatable view on which it is dependent.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE VIEW V2 AS SELECT ID, V FROM V1 WHERE ID < 5 WITH LOCAL CHECK
OPTION;
```

Solution:

SQL Server supports only views with the cascaded check option. Create the view without the check option, and INSTEAD OF, INSERT, UPDATE, or DELETE triggers on it. Then implement the check in the triggers by raising errors when the condition is false.

3.1.3 Migrating Indexes – CREATE INDEX Statement

This section describes the differences between DB2 and SQL Server in the syntax of the CREATE INDEX statement.
Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

3.1.3.1 CLUSTER Keyword

DB2 uses the CLUSTER keyword to create a clustered index, which is incompatible with SQL Server.

DB2 Example:

```
CREATE TABLE DB2_TABLES.EMPLOYEES_INDX (
 ID   DECIMAL(5,0),
 NAME VARCHAR(50),
 LAST_NAME  VARCHAR(50),
 DOB  DATE,
 DEPT VARCHAR(10),
 SAL  DECIMAL(5,0),
 JOB  VARCHAR(10));

CREATE INDEX DB2_TABLES.ID_CLUSTER ON DB2_TABLES.EMPLOYEES_INDX (ID)
CLUSTER;
```

Solution:

Replace the CLUSTER keyword with CLUSTERED. There is also a slight difference in the syntax of creating clustered indexes.

SQL Server Example:

```
CREATE TABLE DBO.EMPLOYEES_INDX (
 ID   DECIMAL(5,0),
 NAME VARCHAR(50),
 LAST_NAME  VARCHAR(50),
 DOB  DATE,
 DEPT VARCHAR(10),
 SAL  DECIMAL(5,0),
 JOB  VARCHAR(10));
```

```
CREATE CLUSTERED INDEX ID_CLUSTER ON DBO.EMPLOYEES_INDX (ID);
```

3.1.3.2 Filling Index Pages

In DB2, you can leave a percentage of each index page as free space when building the index.

DB2 Example:

```
CREATE TABLE DB2_TABLES.EMP_INDX_PSTFREE_REVERS (ID DECIMAL, NAME
VARCHAR(50), LAST_NAME VARCHAR(50), DOB DATE, DEPT VARCHAR (10), SAL
DECIMAL);

CREATE INDEX DB2_TABLES.IDX_PSTFREE_REV ON
DB2_TABLES.EMP_INDX_PSTFREE_REVERS (ID ASC) CLUSTER PCTFREE 30;
```

Solution:

In SQL Server, you can emulate this feature by using the FILLFACTOR keyword and a value of the filled percentage of each index page (the reverse percentage of the PCTFREE clause in DB2).

SQL Server Example:

```
CREATE TABLE DBO.EMP_INDX_PSTFREE_REVERS (ID DECIMAL, NAME VARCHAR(50),
LAST_NAME VARCHAR(50), DOB DATE, DEPT VARCHAR (10), SAL DECIMAL);

CREATE CLUSTERED INDEX IDX_PSTFREE_REV ON DBO.EMP_INDX_PSTFREE_REVERS (ID)
WITH FILLFACTOR=70;
```

3.1.3.3 Collecting Statistics

DB2 includes a set of COLLECT STATISTICS clauses that specify which basic index statistics are to be collected during index creation.

COLLECT DETAILED STATISTICS specifies that extended index statistics are also to be collected during index creation. COLLECT SAMPLED DETAILED STATISTICS specifies that sampling can be used when compiling extended index statistics.

DB2 Example 1:

```
CREATE TABLE DB2_TABLES.EMP_INDX_COLLECT_STAT (ID DECIMAL, NAME
VARCHAR(50), LAST_NAME VARCHAR(50), DOB DATE, DEPT VARCHAR (10), SAL
DECIMAL);

CREATE INDEX DB2_TABLES.IDX1 ON DB2_TABLES.EMP_INDX_COLLECT_STAT (ID)
COLLECT STATISTICS;
```

DB2 Example 2:

```
CREATE INDEX DB2_TABLES.IDX2 ON DB2_TABLES.EMP_INDX_COLLECT_STAT (ID)
COLLECT DETAILED STATISTICS;
```

DB2 Example 3:

```
CREATE INDEX DB2_TABLES.IDX3 ON DB2_TABLES.EMP_INDX_COLLECT_STAT (ID)
COLLECT SAMPLED DETAILED STATISTICS;
```

Solution:

This clause can be partially emulated in SQL Server using the WITH STATISTICS_NORECOMPUTE statement.

SQL Server Example 1:

```
CREATE TABLE DBO.EMP_INDX_COLLECT_STAT (ID DECIMAL, NAME VARCHAR(50),
LAST_NAME VARCHAR(50), DOB DATE, DEPT VARCHAR (10), SAL DECIMAL);
```

```
CREATE INDEX IDX1 ON DBO.EMP_INDX_COLLECT_STAT (ID) WITH
(STATISTICS_NORECOMPUTE = OFF);
```

3.1.3.4 Unique XML Indexes

In DB2, for indexes on XML data, the UNIQUE keyword enforces uniqueness within a single XML column across all documents whose nodes are qualified by the XML pattern.

DB2 Example:

```
CREATE TABLE DB2_TABLES.COMPANY_XML_IND (ID INT NOT NULL, XML_COLUMN XML
NOT NULL);
CREATE UNIQUE INDEX DB2_TABLES.XML_IND ON
DB2_TABLES.COMPANY_XML_IND(XML_COLUMN) GENERATE KEY USING XMLPATTERN
'/COMPANY/EMP/NAME/LAST' AS SQL VARCHAR(100);

INSERT INTO DB2_TABLES.COMPANY_XML_IND VALUES (1,
 XMLPARSE
 (DOCUMENT
'<COMPANY NAME="COMPANY1">
 <EMP ID="31201" SALARY="60000" GENDER="FEMALE">
 <NAME>
 <FIRST>LAURA</FIRST>
 <LAST>BROWN</LAST>
 </NAME>
 <DEPT ID="M25">
 FINANCE
 </DEPT>
 </EMP>
</COMPANY>'))
```

Solution:

SQL Server table needs to have a clustered primary key to create a primary XML index on it. The SQL Server statement PRIMARY XML INDEX contains the primary key of the parent table, so the unique requirement is always achieved automatically.

SQL Server Example:

```
CREATE TABLE DBO.COMPANY_XML_IND (ID INT NOT NULL PRIMARY KEY, XML_COLUMN
XML NOT NULL);

CREATE PRIMARY XML INDEX COMPINDEX ON DBO.COMPANY_XML_IND(XML_COLUMN);
```

3.1.3.5 Indexing of XML Data

Indexing of XML data depends on which pattern expression the data has. The XML pattern affects what exactly will be indexed: the paths or the nodes of the XML document. To index on an XML pattern, you provide the index specification clause GENERATE KEY USING XMLPATTERN during index creation.

DB2 Example 1:

@ - Specifies attributes of the context node. This is the abbreviated syntax for attribute::.

```
CREATE INDEX DB2_TABLES.EMPINDEX ON DB2_TABLES.COMPANY_XML_IND(XML_COLUMN)
GENERATE KEY USING XMLPATTERN '/COMPANY/EMP/@ID' AS SQL DOUBLE;
```

DB2 Example 2:

child:: - Specifies children of the context node. This is the default, if no other forward axis is specified.
attribute:: - Specifies attributes of the context node.

```
CREATE INDEX DB2_TABLES.CHILDINDEX ON
DB2_TABLES.COMPANY_XML_IND(XML_COLUMN) GENERATE KEY USING XMLPATTERN
'/CHILD::COMPANY/CHILD::EMP/ATTRIBUTE::ID' AS SQL DOUBLE;
```

DB2 Example 3:

// (double forward slash) - this is the abbreviated syntax for /descendant-or-self::node()/.

```
CREATE INDEX DB2_TABLES.SLASHINDEX ON
DB2_TABLES.COMPANY_XML_IND(XML_COLUMN) GENERATE KEY USING XMLPATTERN
'//@ID' AS SQL DOUBLE;
```

DB2 Example 4:

descendant-or-self:: - specifies the context node and the descendants of the context node.
node() - matches any node. You cannot use node() if you also specify UNIQUE.

```
CREATE INDEX DB2_TABLES.DESCINDEX ON
DB2_TABLES.COMPANY_XML_IND(XML_COLUMN) GENERATE KEY USING XMLPATTERN
'/DESCENDANT-OR-SELF::NODE()/ATTRIBUTE::ID' AS SQL DOUBLE;
```

DB2 Example 5:

text() - matches any text node.

```
CREATE INDEX DB2_TABLES.TEXTINDEX ON
DB2_TABLES.COMPANY_XML_IND(XML_COLUMN) GENERATE KEY USING XMLPATTERN
'/COMPANY/EMP/NAME/LAST/TEXT()' AS SQL VARCHAR(25);
```

Solution:

In SQL Server, create a secondary XML index using the FOR { VALUE | PATH | PROPERTY } clause.

XML indexes fall into the following categories:

Primary XML index

Secondary XML index

The first index on the xml type column must be the primary XML index. Using the primary XML index, the following types of secondary indexes are supported: PATH, VALUE, and PROPERTY.

PATH Secondary XML Index - if your queries generally specify path expressions on xml type columns, a PATH secondary index may be able to speed up the search.

SQL Server Example 1:

```
CREATE TABLE DBO.COMP_XML_IND (ID INT NOT NULL PRIMARY KEY, XML_COLUMN XML
NOT NULL);
-- CREATE PRIMARY INDEX.
CREATE PRIMARY XML INDEX PR_EMPINDEX ON DBO.COMP_XML_IND(XML_COLUMN)
-- CREATE SECONDARY INDEX.
CREATE XML INDEX EMPINDEX_PATH ON DBO.COMP_XML_IND(XML_COLUMN) USING XML
INDEX PR_EMPINDEX FOR PATH
```

SQL Server Example 2:

```
CREATE PRIMARY XML INDEX PR_EMPINDEX ON DBO.COMP_XML_IND(XML_COLUMN)
-- CREATE SECONDARY INDEX.
CREATE XML INDEX EMPINDEX_PATH ON DBO.COMP_XML_IND(XML_COLUMN) USING XML
INDEX PR_EMPINDEX FOR PATH
```

SQL Server Example 3:

```
CREATE PRIMARY XML INDEX EMPINDEX ON DBO.COMP_XML_IND(XML_COLUMN);
-- CREATE SECONDARY INDEX.
CREATE XML INDEX EMPINDEX_VALUE ON DBO.COMP_XML_IND(XML_COLUMN) USING XML
INDEX EMPINDEX FOR VALUE
```

SQL Server Example 4:

```
CREATE PRIMARY XML INDEX EMPINDEX ON DBO.COMP_XML_IND(XML_COLUMN);
-- CREATE SECONDARY INDEX.
CREATE XML INDEX EMPINDEX_VALUE ON DBO.COMP_XML_IND(XML_COLUMN) USING XML
INDEX EMPINDEX FOR VALUE
```

SQL Server Example 5:

```
CREATE PRIMARY XML INDEX PR_EMPINDEX ON DBO.COMP_XML_IND(XML_COLUMN)
-- CREATE SECONDARY INDEX.
CREATE XML INDEX EMPINDEX_PATH ON DBO.COMP_XML_IND(XML_COLUMN) USING XML
INDEX PR_EMPINDEX FOR PATH
```

3.1.3.6 COMPRESS clause

COMPRESS - specifies whether index compression is enabled. By default, index compression will be enabled if data row compression is enabled; index compression will be disabled if data row compression is disabled. This option can be used to override the default behavior.

YES - specifies that index compression is enabled. Insert and update operations on the index will be subject to compression.

NO - specifies that index compression is disabled.

DB2 Example :

```
CREATE TABLE DB2_TABLES.EMP_INDX_COMPRESS (ID INT, FIELD VARCHAR(50));

CREATE INDEX DB2_TABLES.COMPRESS_IND_YES ON
DB2_TABLES.EMP_INDX_COMPRESS (ID) COMPRESS YES;
```

Solution:

DATA_COMPRESSION - specifies the data compression option for the specified index, partition number, or range of partitions. The options are as follows:

You could use PAGE compression - index or specified partitions are compressed by using page compression.

SQL Server Example:

```
CREATE TABLE DBO.EMP_INDX_COMPRESS (ID INT, FIELD VARCHAR(50));

CREATE INDEX COMPRESS_IND_YES ON DBO.EMP_INDX_COMPRESS (ID) WITH
(DATA_COMPRESSION = PAGE)
```

3.2 Triggers

This section explains how to convert DB2 triggers to SQL Server triggers.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

3.2.1.1 Create trigger restriction:

1. OLD and NEW can only be specified once each.
2. OLD or NEW correlation names cannot be defined in a FOR EACH STATEMENT trigger.
3. OLD TABLE and NEW TABLE can only be specified once each, and only for AFTER triggers or INSTEAD OF triggers.
4. FOR EACH STATEMENT may not be specified for BEFORE triggers or INSTEAD OF triggers.
5. A trigger event must not be specified more than once for the same operation. For example, INSERT OR DELETE is allowed, but INSERT OR INSERT is not allowed.
6. WHEN condition may not be specified for INSTEAD OF triggers.
7. A compound SQL (compiled) statement cannot be specified if the trigger definition includes a REFERENCING OLD TABLE clause or a REFERENCING NEW TABLE clause. A compound SQL (compiled) statement also cannot be specified for a trigger definition in a partitioned database environment.
8. Transition tables cannot be modified.

The following rules apply to the REFERENCING clause:

- If the triggered-action includes a compound SQL (compiled) statement:

– OLD TABLE or NEW TABLE identifiers cannot be defined.

– If the operation is a DELETE operation, OLD correlation-name captures the value of the deleted row. If it is an UPDATE operation, it captures the value of the row before the UPDATE operation. For an insert operation, OLD correlation-name captures null values for each column of a row.

– For an insert operation or an update operation, the value of NEW captures the new state of the row as provided by the original operation and as modified by any BEFORE trigger that has executed to this point. For a delete operation, NEW correlation-name captures null values for each column of a row. In a BEFORE DELETE trigger, any non-null values assigned to the new transition variables persist only within the trigger where the assignment occurred.

- If the triggered-action does not include a compound SQL (compiled) statement:

– The OLD correlation-name and the OLD TABLE identifier can only be used if the trigger event is either a DELETE operation or an UPDATE operation. If the operation is a DELETE operation, OLD correlation-name captures the value of the deleted row. If it is an UPDATE operation, it captures the value of the row before the UPDATE operation. The same applies to the OLD TABLE identifier and the set of affected rows.

– The NEW correlation-name and the NEW TABLE identifier can only be used if the trigger event is either an INSERT operation or an UPDATE operation. In both operations, the value of NEW captures the new state of the row as provided by the original operation and as modified by any BEFORE trigger that has executed to this point. The same applies to the NEW TABLE identifier and the set of affected rows.

Transition variables can be defined depending on the kind of trigger event:

UPDATE

An UPDATE trigger can refer to both OLD and NEW transition variables.

INSERT

An INSERT trigger can only refer to a NEW transition variable because before the activation of the INSERT operation, the affected row does not exist in the database. That is, there is no original state of the row that would define old values before the triggered action is applied to the database.

DELETE

A DELETE trigger can only refer to an OLD transition variable because there are no new values specified in the delete operation.

Transition variables can only be specified for FOR EACH ROW triggers. In a FOR EACH STATEMENT trigger, a reference to a transition variable is not sufficient to specify to which of the several rows in the set of affected rows the transition variable is referring. Instead, refer to the set of new and old rows by using the OLD TABLE and NEW TABLE clauses of the CREATE TRIGGER statement.

3.2.2 FOR EACH ROW Triggers

DB2 supports the FOR EACH ROW triggers, which are not supported in SQL Server. These triggers are fired as many times as there are rows changed by the triggering statement.

Inside a DB2 trigger, you can refer to columns in the subject table (the table associated with the trigger) by using the OLD and NEW aliases that are specified in the REFERENCING clause. The "OLD AS O" O.*col_name* refers to a column in an existing row before it is updated or deleted. The "NEW AS N" N.*col_name* refers to the column of a new row to be inserted or an existing row after it is updated.

DB2 Example:

```
CREATE TRIGGER DB2_TABLES.FOR_EACH_ROW AFTER UPDATE ON
DB2_TABLES.AFTER_UPD
REFERENCING NEW AS N OLD AS O
FOR EACH ROW
BEGIN ATOMIC
 INSERT INTO DB2_TABLES.AFTER_UPD_INS VALUES (N.ID, O.V || '' || N.V);
END;
```

Solution:

The functionality offered by the FOR EACH ROW trigger can be emulated by using a SQL Server cursor in a trigger.

For this solution, you need to add a new column to the table: This column will be used to uniquely identify the row being updated, so we will name it rowid and assign SQL Server type uniqueidentifier to it. Thus this column is used to synchronize old and new values of each row.

```
ALTER TABLE DBO.TEST ADD ROWID UNIQUEIDENTIFIER DEFAULT NEWID()
```

SQL Server Example:

```
CREATE TRIGGER DBO.FOR_EACH_ROW ON DBO.TEST AFTER UPDATE AS
 BEGIN SET NOCOUNT ON
 /* COLUMN VARIABLES DECLARATION*/
 DECLARE
 @NEW$0 UNIQUEIDENTIFIER,
 @OLD$0 UNIQUEIDENTIFIER,
 @NEW$ID NUMERIC(38, 0),
 @OLD$ID NUMERIC(38, 0),
 @NEW$V VARCHAR(100),
 @OLD$V VARCHAR(100)
 DECLARE
FOREACHINSERTEDROWTRIGGERCURSOR CURSOR LOCAL FORWARD_ONLY READ_ONLY FOR
 SELECT ROWID, ID, V FROM INSERTED
 OPEN FOREACHINSERTEDROWTRIGGERCURSOR
 FETCH FOREACHINSERTEDROWTRIGGERCURSOR
 INTO @NEW$0, @NEW$ID, @NEW$V
 WHILE @@FETCH_STATUS = 0
 BEGIN
 SELECT @OLD$0 = ROWID, @OLD$ID = ID, @OLD$V = V
 FROM DELETED
 WHERE ROWID = @NEW$0
 /* TRIGGER IMPLEMENTATION: BEGIN */
 BEGIN
 INSERT DBO.DEBUG(ID, V)
 VALUES (@NEW$ID, ISNULL(@OLD$V, '') + ISNULL(@NEW$V, ''))
 END
 /* TRIGGER IMPLEMENTATION: END */
 FETCH FOREACHINSERTEDROWTRIGGERCURSOR
 INTO @NEW$0, @NEW$ID, @NEW$V
```

```
END
CLOSE FOREACHINSERTEDROWTRIGGERCURSOR
DEALLOCATE FOREACHINSERTEDROWTRIGGERCURSOR
END
```

3.2.3 FOR EACH STATEMENT Triggers

FOR EACH STATEMENT can only be specified for AFTER triggers.

DB2 supports the FOR EACH STATEMENT trigger, which indicates that the trigger is invoked once after the execution of the triggering statement. This kind of trigger is also supported in SQL Server. But there is one small difference.

Inside a DB2 FOR EACH STATEMENT trigger, you can refer to *special tables* (called *inserted* and *deleted* tables in SQL Server) by using the aliases OLD_TABLE and NEW_TABLE that are specified in the REFERENCING clause. OLD_TABLE refers to rows before they are updated or deleted. NEW_TABLE refers to new rows that were inserted, or existing rows after they were updated.

DB2 Example:

```
CREATE TRIGGER DB2_TABLES.FOR_EACH_ST AFTER UPDATE ON DB2_TABLES.AFTER_UPD
REFERENCING NEW_TABLE AS N OLD_TABLE AS O
FOR EACH STATEMENT
BEGIN ATOMIC
 INSERT INTO DB2_TABLES.AFTER_UPD_INS
       SELECT N.ID, O.V || ' ' || N.V
       FROM N JOIN O ON N.ID=O.ID;
END;
```

Solution:

The functionality offered by the FOR EACH STATEMENT triggers can be emulated by using INSERTED and DELETED aliases instead of NEW_TABLE and OLD_TABLE aliases respectively.

SQL Server Example:

```
CREATE TRIGGER AFTER_UPD_ST ON TEST AFTER UPDATE AS SET NOCOUNT ON
BEGIN
 INSERT INTO DEBUG
 SELECT N.ID, O.V + N.V
 FROM INSERTED N JOIN DELETED O ON N.ID=O.ID;
END
```

3.2.4 BEFORE Triggers

DB2 supports the BEFORE Triggers, which are not supported in SQL Server. BEFORE triggers can be emulated by using a SQL Server INSTEAD OF triggers.

BEFORE triggers must have a granularity of FOR EACH ROW.

DB2 supports BEFORE triggers only for each row. In these DB2 triggers, the BEFORE keyword indicates that the trigger is invoked before the execution of the triggering statement.

DB2 Example:

```
CREATE TRIGGER DB2_TABLES.BEFORE_TRIGGER BEFORE UPDATE ON
DB2_TABLES.BEFORE_UPD
REFERENCING NEW AS N OLD AS O
FOR EACH ROW
BEGIN ATOMIC
 SET N.V=O.V || N.V;
END;
```

Solution:

BEFORE can be emulated by using a SQL Server INSTEAD OF trigger.

Note that for this solution, you need to add a new column to the subject table. This column will be used to uniquely identify the row being updated, so we will name it rowid and assign SQL Server type uniqueidentifier to it. Thus this column lets us synchronize old and new values of each row. For the INSTEAD OF trigger that is based on a view, the view should be modified in such a way that a rowid column is present there.

SQL Server Example:

```
CREATE TRIGGER DBO.BEFORE_TRIGGER ON BEFORE_UPD INSTEAD OF UPDATE AS
 BEGIN SET NOCOUNT ON
 /* COLUMN VARIABLES DECLARATION*/
 DECLARE
 @NEW$0 UNIQUEIDENTIFIER,
 @NEW$ID NUMERIC(38, 0),
 @OLD$ID NUMERIC(38, 0),
 @NEW$V VARCHAR(100),
 @OLD$V VARCHAR(100)
 DECLARE
 FOREACHINSERTEDROWTRIGGERCURSOR CURSOR LOCAL FORWARD_ONLY READ_ONLY FOR
 SELECT ROWID, ID, V FROM INSERTED
 OPEN FOREACHINSERTEDROWTRIGGERCURSOR
                  FETCH FOREACHINSERTEDROWTRIGGERCURSOR
 INTO @NEW$0, @NEW$ID, @NEW$V
 WHILE @@FETCH_STATUS = 0
 BEGIN
 SELECT @OLD$ID = ID, @OLD$V = V
 FROM DELETED
 WHERE ROWID = @NEW$0
 /* ROW-LEVEL TRIGGERS IMPLEMENTATION: BEGIN */
 BEGIN
 SET @NEW$V = ISNULL(@OLD$V, '') + ISNULL(@NEW$V, '')
 END
 /* ROW-LEVEL TRIGGERS IMPLEMENTATION: END */
 /* DML-OPERATION EMULATION */
 UPDATE BEFORE_UPD
 SET ID = @NEW$ID, V = @NEW$V
 WHERE ROWID = @NEW$0
 FETCH FOREACHINSERTEDROWTRIGGERCURSOR
 INTO @NEW$0, @NEW$ID, @NEW$V
 END
 CLOSE FOREACHINSERTEDROWTRIGGERCURSOR
 DEALLOCATE FOREACHINSERTEDROWTRIGGERCURSOR
 END
```

DB2 Example:

```
CREATE TRIGGER DB2_TABLES.TTR$TR1 BEFORE INSERT ON DB2_TABLES.TTR
REFERENCING NEW AS N OLD AS O FOR EACH ROW
BEGIN
SET N.A= N.A||O.A;
END;
```

SQL Server Example:

```
CREATE TRIGGER DBO.TTR$TR1 ON DBO.TTR INSTEAD OF INSERT AS
 BEGIN SET NOCOUNT ON
 /* COLUMN VARIABLES DECLARATION */
 DECLARE
 @NEW$0 UNIQUEIDENTIFIER,
 @NEW$A NUMERIC(38, 0),
 @OLD$A NUMERIC(38, 0)
 DECLARE
 FOREACHINSERTEDROWTRIGGERCURSOR CURSOR LOCAL FORWARD_ONLY READ_ONLY FOR
```

```
SELECT ROWID, A FROM INSERTED
OPEN FOREACHINSERTEDROWTRIGGERCURSOR
      FETCH FOREACHINSERTEDROWTRIGGERCURSOR
INTO @NEW$0, @NEW$A
WHILE @@FETCH_STATUS = 0
BEGIN
SELECT @OLD$A = A
FROM DELETED
WHERE ROWID = @NEW$0
/* ROW-LEVEL TRIGGERS IMPLEMENTATION: BEGIN */
BEGIN
SET @NEW$A = @NEW$A + @OLD$A
END
/* ROW-LEVEL TRIGGERS IMPLEMENTATION: END */
/* DML-OPERATION EMULATION */
INSERT DBO.TTR (ROWID, A) VALUES (@NEW$0, @NEW$A)
FETCH FOREACHINSERTEDROWTRIGGERCURSOR
INTO @NEW$0, @NEW$A
END
CLOSE FOREACHINSERTEDROWTRIGGERCURSOR
DEALLOCATE FOREACHINSERTEDROWTRIGGERCURSOR
END
```

DB2 Example:

```
CREATE TRIGGER DB2_TABLES.BEF_DEL BEFORE DELETE ON DB2_TABLES.BEF_INS
REFERENCING OLD AS O
FOR EACH ROW
BEGIN
SIGNAL SQLSTATE '75000' SET MESSAGE_TEXT = 'YOU ARE INTEND TO DELETE
ROWS';
END;
```

SQL Server Example:

```
CREATE TRIGGER DBO.BEF_DEL ON DBO.TTR INSTEAD OF DELETE AS
 BEGIN SET NOCOUNT ON
 /* COLUMN VARIABLES DECLARATION */
 DECLARE
 @OLD$0 UNIQUEIDENTIFIER,
 @NEW$A NUMERIC(38, 0),
 @OLD$A NUMERIC(38, 0)
 DECLARE
 FOREACHINSERTEDROWTRIGGERCURSOR CURSOR LOCAL FORWARD_ONLY READ_ONLY FOR
 SELECT ROWID, A FROM DELETED
 OPEN FOREACHINSERTEDROWTRIGGERCURSOR
                  FETCH FOREACHINSERTEDROWTRIGGERCURSOR
 INTO @OLD$0, @OLD$A
 WHILE @@FETCH_STATUS = 0
 /* ROW-LEVEL TRIGGERS IMPLEMENTATION: BEGIN */
 BEGIN
 IF 1=1 THROW 75000, 'YOU ARE INTEND TO DELETE ROWS', 1
 END
 /* ROW-LEVEL TRIGGERS IMPLEMENTATION: END */
 /* DML-OPERATION EMULATION */
 DELETE DBO.TTR
 WHERE ROWID = @OLD$0
 FETCH FOREACHINSERTEDROWTRIGGERCURSOR
 INTO @OLD$0, @OLD$A
 CLOSE FOREACHINSERTEDROWTRIGGERCURSOR
 DEALLOCATE FOREACHINSERTEDROWTRIGGERCURSOR
```

```
END
```

3.2.5 Trigger event predicates

A trigger event predicate is used in a triggered action to test the event that activated the trigger.
DELETING - True if the trigger was activated by a delete operation. False otherwise.
INSERTING - True if the trigger was activated by an insert operation. False otherwise.
UPDATING - True if the trigger was activated by an update operation. False otherwise.

DB2 Example:

```
CREATE TRIGGER DB2_TABLES.IF_INS_DEL_UPD
 AFTER INSERT OR DELETE OR UPDATE ON DB2_TABLES.TTR
 REFERENCING NEW AS N OLD AS O FOR EACH ROW
 BEGIN
 IF INSERTING
 THEN UPDATE DB2_TABLES.TTR2 SET A = A + 1;
 END IF;

 IF DELETING
 THEN UPDATE DB2_TABLES.TTR2 SET A = A - 1;
 END IF;

 IF UPDATING
 THEN UPDATE DB2_TABLES.TTR2 SET A = A - 10;
 END IF;
 END;
```

Solution:

DB2 allow any combination of the events can be specified, but each event (INSERT, DELETE, and UPDATE) can only be specified once. In SQL Server more than one trigger event is not supported, you could create a separate trigger for every trigger event.

3.2.6 WHEN clause

WHEN condition may not be specified for INSTEAD OF triggers.
The triggered action condition or WHEN is an optional clause of the triggered action which specifies a search condition that must evaluate to true to run statements within the triggered action. If the WHEN clause is omitted, then the statements within the triggered action are always executed.

The triggered action condition is evaluated once for each row if the trigger is a FOR EACH ROW trigger, and once for the statement if the trigger is a FOR EACH STATEMENT trigger.

DB2 Example1:

```
CREATE TRIGGER DB2_TABLES.BEF_UPD BEFORE UPDATE ON DB2_TABLES.BEFORE_UPD
REFERENCING NEW AS N OLD AS O
FOR EACH ROW WHEN (N.ID=5 AND O.ID=1)
BEGIN ATOMIC
 SET N.V=O.V || N.V;
END;
```

Solution:

In SQL Server, WHEN clause can be emulated by using IF construct.

SQL Server Example1:

```
CREATE TRIGGER BEF_UPD ON BEFORE_UPD INSTEAD OF UPDATE AS
 BEGIN SET NOCOUNT ON
```

```
/* COLUMN VARIABLES DECLARATION*/
DECLARE
@NEW$0 UNIQUEIDENTIFIER,
@NEW$ID NUMERIC(38, 0),
@OLD$ID NUMERIC(38, 0),
@NEW$V VARCHAR(100),
@OLD$V VARCHAR(100)
DECLARE
FOREACHINSERTEDROWTRIGGERCURSOR CURSOR LOCAL FORWARD_ONLY READ_ONLY FOR
SELECT ROWID, ID, V FROM INSERTED
OPEN FOREACHINSERTEDROWTRIGGERCURSOR
                FETCH FOREACHINSERTEDROWTRIGGERCURSOR
INTO @NEW$0, @NEW$ID, @NEW$V
WHILE @@FETCH_STATUS = 0
BEGIN
SELECT @OLD$ID = ID, @OLD$V = V
FROM DELETED
WHERE ROWID = @NEW$0
/* ROW-LEVEL TRIGGERS IMPLEMENTATION: BEGIN*/
BEGIN
IF (@NEW$ID = 5 AND @OLD$ID = 1)
BEGIN
SET @NEW$V = ISNULL(@OLD$V, '') + ISNULL(@NEW$V, '')
END
END
/* ROW-LEVEL TRIGGERS IMPLEMENTATION: END*/
/* DML-OPERATION EMULATION*/
UPDATE DBO.CONDITION
SET ID = @NEW$ID, V = @NEW$V
WHERE ROWID = @NEW$0
FETCH FOREACHINSERTEDROWTRIGGERCURSOR
INTO @NEW$0, @NEW$ID, @NEW$V
END
CLOSE FOREACHINSERTEDROWTRIGGERCURSOR
DEALLOCATE FOREACHINSERTEDROWTRIGGERCURSOR
END
```

DB2 Example2:

```
CREATE TRIGGER DB2_OBJECTS.AFTER_UPD_ST AFTER UPDATE ON
DB2_OBJECTS.AFTER_UPD
REFERENCING NEW_TABLE AS N OLD_TABLE AS O
FOR EACH STATEMENT WHEN ((SELECT COUNT(*) FROM N)>1)
BEGIN ATOMIC
 INSERT INTO DB2_OBJECTS.AFTER_UPD_INS
       SELECT N.ID, O.V || N.V FROM N JOIN O ON N.ID=O.ID;
END
```

SQL Server Example2:

```
CREATE TRIGGER AFTER_UPD_ST ON TEST AFTER UPDATE AS
IF ((SELECT COUNT(*) FROM INSERTED)>1)
BEGIN
 INSERT INTO DEBUG
 SELECT N.ID, O.V + N.V
 FROM INSERTED N JOIN DELETED O ON N.ID=O.ID;
END
```

3.3 Sequences

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

A DB2 SEQUENCE is a user-defined object that generates a series of numeric values based on the specification with which the SEQUENCE was created. The most common purpose of a SEQUENCE is to provide unique values for the primary key column of a table. DB2 SEQUENCEs are not associated with tables. Applications refer to a SEQUENCE object to get the current or next value of that SEQUENCE. DB2 keeps the set of generated values of a SEQUENCE in a cache, and a unique set of cached values is created for each session.

In DB2, the NEXTVAL expression generates and returns the next value for the specified SEQUENCE. The DB2 PREVVAL expression returns the most recently generated value of the previous NEXTVAL expression for the same SEQUENCE within the current application process. In DB2, the value of the PREVVAL expression persists until the next value is generated for the SEQUENCE, the SEQUENCE is dropped, or the application session ends.

Solution:

SQL Server 2014 support objects with functionality similar to that of a DB2 SEQUENCE. In many cases if you use SEQUENCE only for getting NEXTVAL you can convert it to SQL Server SEQUENCE.

DB2 Example:

```
CREATE SEQUENCE CUSTOMER_NO AS INTEGER
INSERT INTO CUSTOMERS VALUES
 (NEXT VALUE FOR CUSTOMER_NO, 'COMMENT', ...)
```

SQL Server Example:

```
CREATE SEQUENCE DBO.CUSTOMER_NO AS INTEGER
INSERT INTO DBO.CUSTOMERS VALUES
 (NEXT VALUE FOR DBO.CUSTOMER_NO, 'COMMENT', ...)
```

However, some features of DB2 SEQUENCEs (e.g. PREVVAL) are not supported in SQL Server. Two distinct scenarios of DB2 SEQUENCE PREVVAL usage exist: a variable that saves SEQUENCE value, and an auxiliary table that represents a DB2 SEQUENCE.

SQL Server Scenario 1: Converting a DB2 table with automatically generated primary key

In the first scenario, a SEQUENCE is used to generate single unique value which is used for a few tables. This is fully compatible with SQL Server usage, and in this case you should modify code like as in example:

DB2 Example:

```
CREATE SEQUENCE SEQ1 AS INTEGER
...
INSERT INTO T1 (ID, NAME)
 VALUES (NEXT VALUE FOR SEQ1, 'NAME');
INSERT INTO T2 (ID, NAME)
 VALUES (PREVIOUS VALUE FOR SEQ1, 'NAME');
...
```

SQL Server Example:

```
CREATE SEQUENCE SEQ1 AS INTEGER
...
DECLARE @NEWID BIGINT;
SELECT @NEWID = NEXT VALUE FOR SEQ1;
INSERT INTO T1 (ID, NAME)
 VALUES (@NEWID, 'NAME');
INSERT INTO T2 (ID, NAME)
 VALUES (@NEWID, 'NAME');
...
```

In this case, we don't need any emulation for PREVVAL.

SQL Server Scenario 2: Converting an auxiliary table representing a DB2 SEQUENCE

In the second scenario, a DB2 SEQUENCE is used in a way that is incompatible with SQL Server SEQUENCE. For example, NEXTVAL and PREVVAL of SEQUENCE can use in difference procedures or application modules.

In this case, you can create an auxiliary table to represent the DB2 SEQUENCE object. This table contains a single column declared as IDENTITY. When you need to get a new SEQUENCE value, you insert a row in this auxiliary table and then retrieve the automatically assigned value from the new row.

```
CREATE TABLE MY_SEQUENCE (
 ID INT IDENTITY(1 /* SEED */, 1 /* INCREMENT*/ )
)
GO
```

To maintain such emulation of NEXTVAL, you must clean up the added rows to avoid unrestricted growth of the auxiliary table. The fastest way to do this in SQL Server is to use a transactional approach:

```
DECLARE @TRAN BIT,
 @NEXTVAL INT
SET @TRAN = 0
IF @@TRANCOUNT > 0
 BEGIN
 SAVE TRANSACTION SEQ
 SET @TRAN = 1
 END
ELSE BEGIN TRANSACTION
INSERT INTO MY_SEQUENCE DEFAULT VALUES
SET @NEXTVAL = SCOPE_IDENTITY()
IF @TRAN=1
 ROLLBACK TRANSACTION SEQ
ELSE ROLLBACK
```

In SQL Server, IDENTITY is generated in a transaction-independent way and, as in DB2, rolling back the transaction does not affect the current IDENTITY value. In this scenario, we can emulate PREVVAL by using SQL Server @@IDENTITY or SCOPE_IDENTITY() functions. @@IDENTITY returns the value for the last INSERT statement in the session, and SCOPE_IDENTITY() gets the last IDENTITY value assigned within the scope of current Transact-SQL module. Note that the values returned by these two functions can be overwritten by next INSERT statement in the current session, so we highly recommend that you save the value in an intermediate variable, if PREVVAL is used afterwards in the source code. Both @@IDENTITY and SCOPE_IDENTITY() are limited to the current session scope, which means that as in DB2, the identities generated by concurrent processes are not visible.

Note: Azure SQL DB doesn't support sequence objects.

3.4 Data Manipulation Statements

This section explains how to migrate commonly used DML statements from DB2 to SQL Server.

3.4.1 SELECT Statement

3.4.1.1 FETCH FIRST Clause

In DB2, the FETCH FIRST clause sets a maximum number of rows that can be retrieved.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
SELECT ID, VAL
```

```
FROM (
 SELECT ID, VAL
 FROM TABLE_FF
 ORDER BY ID DESC
 FETCH FIRST 10 ROWS ONLY
) A
FETCH FIRST 5 ROW ONLY
```

Solution:

In SQL Server, convert the FETCH FIRST clause by using the optional <offset_fetch> clause of ORDER BY in SELECT statement. OFFSET clause is required, but you should set rows count equal to 0. If ORDER BY clause is missed in DB2, in SQL Server you should generate ORDER BY clause with (SELECT <constant>) as an order_by_expression.

SQL Server Example:

```
SELECT ID, VAL
FROM (
 SELECT ID, VAL
 FROM TABLE_FF
 ORDER BY ID DESC
 OFFSET 0 ROWS
 FETCH FIRST 10 ROWS ONLY
) A
ORDER BY (SELECT 1)
OFFSET 0 ROW
FETCH FIRST 5 ROW ONLY
```

3.4.1.2 OPTIMIZE FOR Clause

In DB2, the OPTIMIZE FOR clause requests special processing of the *select statement*. If the OPTIMIZE FOR clause is specified, it is assumed that the number of rows retrieved will probably not exceed *n*, where *n* is the value of integer.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
SELECT ID, VAL
FROM TABLE_FF
OPTIMIZE FOR 5 ROWS
```

Solution:

In SQL Server, emulate an OPTIMIZE FOR clause by using the FAST query hint.

SQL Server Example:

```
SELECT ID, VAL
FROM TABLE_FF
OPTION (FAST 5);
```

3.4.1.3 EXCEPT ALL and INTERSECT ALL Operators

DB2 supports two set operators that are incompatible with SQL Server: EXCEPT ALL and INTERSECT ALL.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

EXCEPT or **EXCEPT ALL**

Derives a result table by combining two other result tables (R1 and R2). If EXCEPT ALL is specified, the result consists of all rows that do not have a corresponding row in R2, where duplicate rows are significant. If EXCEPT is specified without the ALL option, the result consists of all rows that are only in R1, with duplicate rows in the result of this operation eliminated.

For compatibility with other SQL implementations, MINUS can be specified as a synonym for EXCEPT.

INTERSECT or **INTERSECT ALL**

Derives a result table by combining two other result tables (R1 and R2). If INTERSECT ALL is specified, the result consists of all rows that are in both R1 and R2. If INTERSECT is specified without the ALL option, the result consists of all rows that are in both R1 and R2, with the duplicate rows eliminated.

The ALL versions of the EXCEPT and INTERSECT operators are not supported in SQL Server.

DB2 Example:

```
SELECT X, Y, Z FROM (VALUES (1, 1, 1), (1, 1, 1), (1, 1, 2), (2, 1, 2),
 (2, 1, 2), (2, 1, 1), (3, 1, 1), (4, 1, 1), (4, 1, 1), (5, 1, 1)
) R1 (X, Y, Z)
EXCEPT ALL
SELECT X, Y, Z FROM (VALUES (1, 1, 1), (1, 1, 1), (2, 1, 2), (3, 1, 1),
 (3, 1, 2), (3, 2, 1), (3, 1, 3), (4, 2, 1)
) R2 (X, Y, Z)
ORDER BY X, Y, Z

SELECT X, Y, Z FROM (VALUES (1, 1, 1), (1, 1, 1), (1, 1, 2), (2, 1, 2),
 (2, 1, 2), (2, 1, 1), (3, 1, 1), (4, 1, 1), (4, 1, 1), (5, 1, 1)
) R1 (X, Y, Z)
EXCEPT
SELECT X, Y, Z FROM (VALUES (1, 1, 1), (1, 1, 1), (2, 1, 2), (3, 1, 1),
 (3, 1, 2), (3, 2, 1), (3, 1, 3), (4, 2, 1)
) R2 (X, Y, Z)
ORDER BY X, Y, Z

SELECT X, Y, Z FROM (VALUES (1, 1, 1), (1, 1, 1), (1, 1, 2), (2, 1, 2),
 (2, 1, 2), (2, 1, 1), (3, 1, 1), (4, 1, 1), (4, 1, 1), (5, 1, 1)
) R1 (X, Y, Z)
INTERSECT ALL
SELECT X, Y, Z FROM (VALUES (1, 1, 1), (1, 1, 1), (2, 1, 2), (3, 1, 1),
 (3, 1, 2), (3, 2, 1), (3, 1, 3), (4, 2, 1)
) R2 (X, Y, Z)
ORDER BY X, Y, Z

SELECT X, Y, Z FROM (VALUES (1, 1, 1), (1, 1, 1), (1, 1, 2), (2, 1, 2),
 (2, 1, 2), (2, 1, 1), (3, 1, 1), (4, 1, 1), (4, 1, 1), (5, 1, 1)
) R1 (X, Y, Z)
INTERSECT
SELECT X, Y, Z FROM (VALUES (1, 1, 1), (1, 1, 1), (2, 1, 2), (3, 1, 1),
 (3, 1, 2), (3, 2, 1), (3, 1, 3), (4, 2, 1)
) R2 (X, Y, Z)
ORDER BY X, Y, Z
```

Solution:

In SQL Server, emulate INTERSECT ALL and EXCEPT ALL by using an additional numeric column (Tmp$Num) with INTERSECT and EXCEPT as shown:

```
SELECT <SELECT_COLUMNS_OR_ALIAS> FROM (
 SELECT <FIRST_SELECT_COLUMNS_WITH_ALIAS>,
 ROW_NUMBER() OVER(PARTITION BY
 <FIRST_SELECT_COLUMNS_WITHOUT_ALIAS> ORDER BY (SELECT 1)
 ) AS TMP$NUM
 FROM ...
 { INTERSECT | EXCEPT }
 SELECT <SECOND_SELECT_COLUMNS_WITH_ALIAS>,
 ROW_NUMBER() OVER(PARTITION BY
 <SECOND_SELECT_COLUMNS_WITHOUT_ALIAS> ORDER BY (SELECT 1)
 ) AS TMP$NUM
 FROM ...
) <SUB QUERY TABLE NAME>
```

All duplicate rows are numbered in both SELECT statements in the new column Tmp$Num. This set no longer contains duplicates, so you can now use INTERSECT or EXCEPT. Then you can select the result without the Tmp$Num column.

SQL Server Example:

```
SELECT X, Y, Z FROM (
 SELECT X, Y, Z,
 ROW_NUMBER() OVER(PARTITION BY X, Y, Z ORDER BY (SELECT 1)
 ) AS TMP$NUM
 FROM (VALUES (1, 1, 1), (1, 1, 1), (1, 1, 2), (2, 1, 2), (2, 1, 2),
 (2, 1, 1), (3, 1, 1), (4, 1, 1), (4, 1, 1), (5, 1, 1)
 ) R1 (X, Y, Z)
 EXCEPT
 SELECT X, Y, Z,
 ROW_NUMBER() OVER(PARTITION BY X, Y, Z ORDER BY (SELECT 1)
 ) AS TMP$NUM
 FROM (VALUES (1, 1, 1), (1, 1, 1), (2, 1, 2), (3, 1, 1), (3, 1, 2),
 (3, 2, 1), (3, 1, 3), (4, 2, 1)) R2 (X, Y, Z)
) R3
ORDER BY X, Y, Z

SELECT X, Y, Z FROM (VALUES (1, 1, 1), (1, 1, 1), (1, 1, 2), (2, 1, 2),
 (2, 1, 2), (2, 1, 1), (3, 1, 1), (4, 1, 1), (4, 1, 1), (5, 1, 1)
) R1 (X, Y, Z)
EXCEPT
SELECT X, Y, Z FROM (VALUES (1, 1, 1), (1, 1, 1), (2, 1, 2), (3, 1, 1),
 (3, 1, 2), (3, 2, 1), (3, 1, 3), (4, 2, 1)
) R2 (X, Y, Z)
ORDER BY X, Y, Z

SELECT X, Y, Z FROM (
 SELECT X, Y, Z,
 ROW_NUMBER() OVER(PARTITION BY X, Y, Z ORDER BY (SELECT 1)
 ) AS TMP$NUM
 FROM (VALUES (1, 1, 1), (1, 1, 1), (1, 1, 2), (2, 1, 2), (2, 1, 2),
 (2, 1, 1), (3, 1, 1), (4, 1, 1), (4, 1, 1), (5, 1, 1)
 ) R1 (X, Y, Z)
 INTERSECT
 SELECT X, Y, Z,
 ROW_NUMBER() OVER(PARTITION BY X, Y, Z ORDER BY (SELECT 1)
```

```
 ) AS TMP$NUM
 FROM (VALUES (1, 1, 1), (1, 1, 1), (2, 1, 2), (3, 1, 1), (3, 1, 2),
 (3, 2, 1), (3, 1, 3), (4, 2, 1)
 ) R2 (X, Y, Z)
) R3
ORDER BY X, Y, Z

SELECT X, Y, Z FROM (VALUES (1, 1, 1), (1, 1, 1), (1, 1, 2), (2, 1, 2),
 (2, 1, 2), (2, 1, 1), (3, 1, 1), (4, 1, 1), (4, 1, 1), (5, 1, 1)
) R1 (X, Y, Z)
INTERSECT
SELECT X, Y, Z FROM (VALUES (1, 1, 1), (1, 1, 1), (2, 1, 2), (3, 1, 1),
 (3, 1, 2), (3, 2, 1), (3, 1, 3), (4, 2, 1)
) R2 (X, Y, Z)
ORDER BY X, Y, Z
```

3.4.1.4 ORDER BY in a Subquery and ORDER OF

In DB2, The ORDER BY clause specifies an ordering of the rows of the result table. The ORDER OF *table-designator* clause specifies that the same ordering used in *table-designator* applies to the result table of the subselect. The ordering that is applied is the same as if the columns of the ORDER BY clause in the nested subselect (or fullselect) were included in the outer subselect (or fullselect), and these columns were specified in place of the ORDER OF clause.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
SELECT ID, VAL
FROM (
 SELECT ID, VAL
 FROM TABLE_FF
 ORDER BY ID DESC
) A
ORDER BY VAL, ORDER OF A
```

Solution:

In SQL Server, the ORDER BY clause is invalid in views, inline functions, derived tables, subqueries, and common table expressions, unless TOP, OFFSET or FOR XML is also specified. Also functionality of ORDER OF is not supported in SQL Server.

Replace the ORDER OF clause with a list of fields or aliases from the subquery sort specification. Then either remove the ORDER BY clause from the subquery (see example 1) or add the TOP clause to the subquery (see example 2) or add the OFFSET clause to the subquery (see example 3).

SQL Server Example 1:

```
SELECT ID, VAL
FROM (
 SELECT ID, VAL
 FROM TABLE_FF
) A
ORDER BY VAL, ID DESC
```

SQL Server Example 2:

```
SELECT ID, VAL
```

```
FROM (
 SELECT TOP 100 PERCENT ID, VAL
 FROM TABLE_FF
 ORDER BY ID DESC
) A
ORDER BY VAL, ID DESC
```

SQL Server Example 3:

```
SELECT ID, VAL
FROM (
 SELECT ID, VAL
 FROM TABLE_FF
 ORDER BY ID DESC
 OFFSET 0 ROW
) A
ORDER BY VAL, ID DESC
```

3.4.1.5 Column Names in a Correlation Clause

In DB2, when a correlation name is specified, column names can also be specified to give names to the columns of the table name, view name, nickname, function name reference, or nested table expression.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example 1:

```
SELECT A, B
FROM (SELECT ID, VAL FROM TABLE_FF) AS C (A,B)
ORDER BY A;
```

DB2 Example 2:

```
SELECT A, B
FROM TABLE_C AS C (A,B)
ORDER BY A;
```

Solution:

In SQL Server, column aliases can be specified only after derived table (like nested table in DB2). Replace the column aliases with the column names with aliases (see example 2).

SQL Server Example 1:

```
SELECT A, B
FROM (SELECT ID, VAL FROM TABLE_FF) AS C (A,B)
ORDER BY A;
```

SQL Server Example 2:

```
SELECT ID AS A, VAL AS B
FROM TABLE_FF AS C
ORDER BY A;
```

3.4.1.6 Alias for a Nested Table Expression

In DB2, you do not need to specify an alias for a nested table expression.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
SELECT VAL
FROM (SELECT ID, VAL FROM TABLE_FF)
ORDER BY ID;
```

Solution:

In SQL Server, you must include an alias for a derived table (subquery). Add an alias for the nested table expression.

SQL Server Example:

```
SELECT VAL
FROM (SELECT ID, VAL FROM TABLE_FF) AS C
ORDER BY ID;
```

3.4.1.7 Table-Valued Function Call in a FROM Clause

In general in DB2, a table function, together with its argument values, can be referenced in the FROM clause of a SELECT in exactly the same way as a table or view.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE FUNCTION FUNC_TABLE_LIST
 (PAR_SCHEMA VARCHAR(30), PAR_NAME CHAR(1))
RETURNS TABLE (NAME VARCHAR(128), CTIME TIMESTAMP)
RETURN
 SELECT NAME, CTIME
 FROM SYSIBM.SYSTABLES
 WHERE (UPPER(CREATOR) = UPPER(PAR_SCHEMA))
 AND (UPPER(SUBSTR(NAME, 1, 1)) = UPPER(PAR_NAME));

SELECT *
FROM TABLE(FUNC_TABLE_LIST('DB2_OBJECTS', 'A')) AS T;
```

Solution:

In SQL Server, the syntax of a table-valued function call in a FROM clause is different from that in DB2. In SQL Server, remove the TABLE keyword and the parentheses around the function name in the FROM clause.

SQL Server Example:

```
CREATE FUNCTION FUNC_TABLE_LIST
 (@PAR_SCHEMA VARCHAR(30), @PAR_NAME CHAR(1))
RETURNS TABLE
RETURN
 SELECT T.NAME, T.CREATE_DATE CTIME
 FROM SYS.TABLES T
 INNER JOIN SYS.SCHEMAS S ON T.SCHEMA_ID = S.SCHEMA_ID
 WHERE (UPPER(S.NAME) = UPPER(@PAR_SCHEMA))
 AND (UPPER(SUBSTRING(T.NAME, 1, 1)) = UPPER(@PAR_NAME));
```

```
SELECT *
FROM FUNC_TABLE_LIST('DBO', 'A') AS T;
```

3.4.1.8 Data–change-table-reference Clause

In DB2, a *data-change-table-reference* clause specifies an intermediate result table. This table is based on the rows that are directly changed by the searched UPDATE, searched DELETE, or INSERT statement that is included in the clause. A data-change-table-reference can be specified as the only table-reference in the FROM clause of the outer fullselect that is used in a select-statement, a SELECT INTO statement, or a common table expression.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

In DB2, the table types for the data-change-table-reference are:

OLD: Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement as they existed before the application of the data change statement.

NEW: Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement before the application of referential constraints and AFTER triggers. Data in the target table at the completion of the statement might not match the data in the intermediate result table because of additional processing for referential constraints and AFTER triggers.

FINAL: Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement as they exist at the completion of the data change statement.

The content of the intermediate result table for a data-change-table-reference is determined when the cursor opens. The intermediate result table contains all manipulated rows, including all the columns in the specified target table or view. All the columns of the target table or view for an SQL data change statement are accessible using the column names from the target table or view. If an INCLUDE clause was specified within a data change statement, the intermediate result table will contain these additional columns.

DB2 Example 1:

```
SELECT ID, VAL, MODIFY, COMMENT
FROM NEW TABLE (
 INSERT INTO TABLE_FF (ID, VAL) INCLUDE (COMMENT VARCHAR(128))
       VALUES (1, 'ABCDEFG', 'INSERT_NEW')
);

SELECT ID, VAL, MODIFY, COMMENT
FROM FINAL TABLE (
 INSERT INTO TABLE_FF (ID, VAL) INCLUDE (COMMENT VARCHAR(128))
       VALUES (1, 'ABCDEFG', 'INSERT_FINAL')
);
```

DB2 Example 2:

```
SELECT ID, VAL, MODIFY, COMMENT
FROM OLD TABLE (
 UPDATE TABLE_FF INCLUDE (COMMENT VARCHAR(128)) SET
 VAL = 'ABCDEFG',
 MODIFY = CURRENT TIMESTAMP,
 COMMENT = 'UPDATE OLD'
 WHERE ID = 10
);
```

```
SELECT ID, VAL, MODIFY, COMMENT
FROM NEW TABLE (
 UPDATE TABLE_FF INCLUDE (COMMENT VARCHAR(128)) SET
 VAL = 'ABCDEFG',
 MODIFY = CURRENT TIMESTAMP,
 COMMENT = 'UPDATE NEW'
 WHERE ID = 10
);

SELECT ID, VAL, MODIFY, COMMENT
FROM FINAL TABLE (
 UPDATE TABLE_FF INCLUDE (COMMENT VARCHAR(128)) SET
 VAL = 'ABCDEFG',
 MODIFY = CURRENT TIMESTAMP,
 COMMENT = 'UPDATE FINAL'
 WHERE ID = 10
);
```

DB2 Example 3:

```
SELECT ID, VAL, MODIFY, COMMENT
FROM OLD TABLE (
 DELETE FROM TABLE_FF INCLUDE (COMMENT VARCHAR(128))
 SET COMMENT = 'DELETE OLD'
 WHERE ID = 10
);
```

Solution:

In SQL Server, rewrite the SELECT statement with the data-change-table-reference clause in SQL Server syntax, using the OUTPUT clause in the nested INSERT, UPDATE, and DELETE statements. For table types, make the following changes:

- To emulate the DB2 OLD table, use the DELETED column prefix.
- To emulate the DB2 NEW table, use the INSERTED column prefix.
- No possibility to emulate the DB2 FINAL table.

You can emulate the included columns by adding constants or expressions to the output select list.

SQL Server Example 1:

```
INSERT INTO TABLE_FF (ID, VAL)
OUTPUT INSERTED.ID, INSERTED.VAL, INSERTED.MODIFY, 'INSERT NEW' COMMENT
VALUES (1, 'ABCDEFG');
```

SQL Server Example 2:

```
UPDATE TABLE_FF SET
 VAL = 'ABCDEFG',
 MODIFY = GETDATE()
OUTPUT INSERTED.ID, INSERTED.VAL, INSERTED.MODIFY, 'UPDATE NEW' COMMENT
WHERE ID = 10;

UPDATE TABLE_FF SET
 VAL = 'ABCDEFG',
 MODIFY = GETDATE()
OUTPUT DELETED.ID, DELETED.VAL, DELETED.MODIFY, 'UPDATE OLD' COMMENT
WHERE ID = 10;
```

SQL Server Example 3:

```
DELETE FROM TABLE_FF
OUTPUT DELETED.ID, DELETED.VAL, DELETED.MODIFY, 'DELETE OLD' COMMENT
WHERE ID = 10;
```

3.4.1.9 Outer join operator

In DB2, when you set the DB2_COMPATIBILITY_VECTOR registry variable to support the outer join operator (+), queries can use this operator as alternative syntax within predicates of the WHERE clauseLinks: DB2 for Linux UNIX and Windows 10.5.0, SQL Server 2014.

DB2 Example 1:

```
SELECT * FROM TABLE_FF A, TABLE_F B
WHERE A.ID = B.ID (+);
```

DB2 Example 2:

```
SELECT * FROM TABLE_FF A, TABLE_F B
WHERE A.ID (+) = B.ID;
```

Solution:

In SQL Server, rewrite the joins to ANSI format. If the operator (+) is specified with right operand, the joins are converted to LEFT OUTER JOIN. If the operator (+) is specified with left operand, the joins are converted to RIGHT OUTER JOIN.

SQL Server Example 1:

```
SELECT * FROM TABLE_FF A
LEFT OUTER JOIN TABLE_F B
ON A.ID = B.ID;
```

SQL Server Example 2:

```
SELECT * FROM TABLE_FF A
RIGHT OUTER JOIN TABLE_F B
ON A.ID = B.ID;
```

3.4.1.10 Hierarchical queries

A hierarchical query is a form of recursive query that retrieves a hierarchy from relational data by using a CONNECT BY clause. You can then use CONNECT BY syntax, including pseudocolumns, unary operators, and the SYS_CONNECT_BY_PATH scalar functionLinks: DB2 for Linux UNIX and Windows 10.5.0, SQL Server 2014.

DB2 Example:

```
SELECT
 NAME,
 LEVEL,
 SALARY,
 CONNECT_BY_ROOT NAME AS ROOT,
 SYS_CONNECT_BY_PATH(NAME, ':') AS CHAIN
FROM MY_EMP
START WITH NAME = 'GOYAL'
CONNECT BY PRIOR EMPID = MGRID
```

Solution:

In SQL Server, emulate hierarchical queries by using WITH common_table_expression.

SQL Server Example:

```
WITH EMP_REPORT(EMPID, NAME, LEVEL, SALARY, ROOT, CHAIN, SORT) AS (
 SELECT
 EMPID,
 NAME,
 1 LEVEL,
 SALARY,
 NAME ROOT,
 CAST(':' + NAME AS VARCHAR(MAX)) CHAIN,
 CAST('.' +
 CAST(EMPID AS VARCHAR(10)) AS VARCHAR(MAX)) SORT
 FROM MY_EMP
 WHERE NAME = 'GOYAL'

 UNION ALL

 SELECT
 E.EMPID,
 E.NAME,
 D.LEVEL + 1 LEVEL,
 E.SALARY,
 D.ROOT,
 CAST(D.CHAIN + ':' + E.NAME AS VARCHAR(MAX)) CHAIN,
 CAST(D.SORT + '.' +
 CAST(E.EMPID AS VARCHAR(10)) AS VARCHAR(MAX)) SORT
 FROM MY_EMP E
 JOIN EMP_REPORT AS D ON E.MGRID = D.EMPID
)
SELECT NAME, LEVEL, SALARY, ROOT, CHAIN
FROM EMP_REPORT
ORDER BY SORT;
```

3.4.1.11 ROWNUM pseudocolumn

In DB2, ROWNUM numbers the records in a result set. The first record that meets the WHERE clause criteria in a SELECT statement is given a row number of 1, and every subsequent record meeting that same criteria increases the row number. Note that ROWNUM is affected by the ORDER BY clauseLinks: DB2 for Linux UNIX and Windows 10.5.0, SQL Server 2014.

DB2 Example:

```
SELECT T.*
FROM TABLE_FF T
WHERE ROWNUM BETWEEN 2 AND 5
ORDER BY ID DESC
```

Solution:

In SQL Server, emulate ROWNUM pseudocolumn by using ROW_NUMBER Ranking Function. This function can only appear in the SELECT or ORDER BY clauses.

SQL Server Example:

```
SELECT T.ID, T.VAL, T.MODIFY FROM (
 SELECT T.*, ROW_NUMBER() OVER (ORDER BY ID DESC) ROWNUM
 FROM TABLE_FF T
) T
WHERE T.ROWNUM BETWEEN 2 AND 5
```

3.4.1.12 common-table-expression

A *common table expression* permits defining a result table with a table-name that can be specified as a table name in any FROM clause of the fullselect that follows.
Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
WITH PLUS AS (
SELECT NAME, DEPT
FROM DB2_TABLES.EMPLOYEES
WHERE DEPT=20)
SELECT NAME, DEPT FROM PLUS;
```

Solution:
In MS SQL Server you could use the same WITH common_table_expression – which specifies a temporary named result set.

SQL Server Example:

```
WITH PLUS AS (
SELECT NAME, DEPT
FROM [DBO].[TEST_FUNC_EMPLOYEES]
WHERE DEPT=30)
SELECT NAME, DEPT FROM PLUS;
```

DB2 Example:

```
CREATE TABLE DB2_TABLES.MYEMPLOYEE (EMPLOYEEID INT, MANAGERID INT)

INSERT INTO DB2_TABLES.MYEMPLOYEE (EMPLOYEEID, MANAGERID) VALUES ( 2, 10);
INSERT INTO DB2_TABLES.MYEMPLOYEE (EMPLOYEEID, MANAGERID) VALUES ( 3, 10);
INSERT INTO DB2_TABLES.MYEMPLOYEE (EMPLOYEEID, MANAGERID) VALUES ( 4, 10);
INSERT INTO DB2_TABLES.MYEMPLOYEE (EMPLOYEEID, MANAGERID) VALUES ( 5, 11);
INSERT INTO DB2_TABLES.MYEMPLOYEE (EMPLOYEEID, MANAGERID) VALUES ( 6, 11);
INSERT INTO DB2_TABLES.MYEMPLOYEE (EMPLOYEEID, MANAGERID) VALUES ( 7, 12);
INSERT INTO DB2_TABLES.MYEMPLOYEE (EMPLOYEEID, MANAGERID) VALUES ( 8, 12);
INSERT INTO DB2_TABLES.MYEMPLOYEE (EMPLOYEEID, MANAGERID) VALUES ( 9, 12);
INSERT INTO DB2_TABLES.MYEMPLOYEE (EMPLOYEEID, MANAGERID) VALUES (10, 15);
INSERT INTO DB2_TABLES.MYEMPLOYEE (EMPLOYEEID, MANAGERID) VALUES (11, 16);
INSERT INTO DB2_TABLES.MYEMPLOYEE (EMPLOYEEID, MANAGERID) VALUES (12, 16);
INSERT INTO DB2_TABLES.MYEMPLOYEE (EMPLOYEEID, MANAGERID) VALUES (13, 15);
INSERT INTO DB2_TABLES.MYEMPLOYEE (EMPLOYEEID, MANAGERID) VALUES (14, 16);
INSERT INTO DB2_TABLES.MYEMPLOYEE (EMPLOYEEID, MANAGERID) VALUES (15, 17);
INSERT INTO DB2_TABLES.MYEMPLOYEE (EMPLOYEEID, MANAGERID) VALUES (16, 17);
INSERT INTO DB2_TABLES.MYEMPLOYEE (EMPLOYEEID, MANAGERID) VALUES (17,
NULL);

WITH DIRECTREPORTS(MANAGERID, EMPLOYEEID, EMPLOYEELEVEL) AS
(
SELECT MANAGERID, EMPLOYEEID, 0 AS EMPLOYEELEVEL
FROM DB2_TABLES.MYEMPLOYEE
WHERE MANAGERID IS NULL
 UNION ALL
```

```
SELECT E.MANAGERID, E.EMPLOYEEID, EMPLOYEELEVEL + 1
FROM DB2_TABLES.MYEMPLOYEE AS E,DIRECTREPORTS AS D
WHERE E.MANAGERID = D.EMPLOYEEID
)
SELECT MANAGERID, EMPLOYEEID, EMPLOYEELEVEL
FROM DIRECTREPORTS
ORDER BY MANAGERID;
```

Solution:
In MS SQL Server you could use the same WITH common_table_expression – which specifies a temporary named result set.

SQL Server Example:

```
WITH DIRECTREPORTS(MANAGERID, EMPLOYEEID, EMPLOYEELEVEL) AS
(
SELECT MANAGERID, EMPLOYEEID, 0 AS EMPLOYEELEVEL
FROM DBO.MYEMPLOYEE
WHERE MANAGERID IS NULL
 UNION ALL
SELECT E.MANAGERID, E.EMPLOYEEID, EMPLOYEELEVEL + 1
FROM DBO.MYEMPLOYEE AS E, DIRECTREPORTS AS D
WHERE E.MANAGERID = D.EMPLOYEEID
)
SELECT MANAGERID, EMPLOYEEID, EMPLOYEELEVEL
FROM DIRECTREPORTS
ORDER BY MANAGERID;
```

DB2 Example:

```
WITH PLUS (AA,BB) AS (
SELECT NAME, DEPT
FROM DB2_TABLES.EMPLOYEES
WHERE DEPT=20)
SELECT AA, BB FROM PLUS;
```

SQL Server Example:

```
WITH PLUS (AA,BB) AS (
SELECT NAME, DEPT
FROM [DBO].[TEST_FUNC_EMPLOYEES]
WHERE DEPT=30)
SELECT AA, AA FROM PLUS
```

3.4.2 VALUES Statement

In DB2, the VALUES statement is a form of query. The VALUES statement derives a result table by specifying the actual values, using expressions or row expressions, for each column of a row in the result table. Multiple rows may be specified.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
VALUES (1), (2), (3); -- 3 ROWS OF 1 COLUMN
VALUES 1, (2), 3; -- 3 ROWS OF 1 COLUMN
VALUES (1, 2, 3); -- 1 ROW OF 3 COLUMNS
VALUES (1, 21), (2, 22), (3, 23); -- 3 ROWS OF 2 COLUMNS
```

Solution:

In SQL Server, the VALUES statement is not supported. For emulation use the table value constructor to specify multiple values in the FROM clause of a SELECT statement. The values list must be always enclosed in parentheses. Aliases for sets and columns must be specified.

SQL Server Example:

```
SELECT * FROM(VALUES (1), (2), (3)) T(A); -- 3 ROWS OF 1 COLUMN
SELECT * FROM(VALUES (1), (2), (3)) T(A); -- 3 ROWS OF 1 COLUMN
SELECT * FROM(VALUES (1, 2, 3)) T(A, B, C); -- 1 ROW OF 3 COLUMNS
SELECT * FROM(VALUES (1, 21), (2, 22), (3, 23)) T(A, B);
 -- 3 ROWS OF 2 COLUMNS
```

3.4.3 INSERT Statement

3.4.3.1 Values Without Parentheses

In DB2, you can write single values in a VALUES clause with or without parentheses. In SQL Server, the values list must be always enclosed in parentheses.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
INSERT INTO TABLE_INS (ID) VALUES 1;
INSERT INTO TABLE_INS (ID) VALUES 1,(2),3;
```

Solution:

In SQL Server, add parentheses to all single-row values.

SQL Server Example:

```
INSERT INTO TABLE_INS (ID) VALUES (1);
INSERT INTO TABLE_INS (ID) VALUES (1),(2),(3);
```

3.4.3.2 Subquery (fullselect) as Object of the INSERT Operation

In DB2, you can use a subquery (fullselect, insertable view) as the object of the INSERT operation.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
INSERT INTO (SELECT * FROM TABLE_INS) VALUES (1,'ABC');
```

Solution:

In SQL Server, use the common table expression (CTE) to emulate this functionality.

SQL Server Example:

```
WITH PLUS AS (SELECT * FROM TABLE_INS)
INSERT INTO PLUS VALUES (1,'ABC');
```

3.4.3.3 Common Table Expression (CTE) in INSERT Statement

DB2 and SQL Server have different syntaxes for the common table expression (CTE) in an INSERT statement. In DB2, a CTE can be used only with the SELECT part of an INSERT SELECT statement. In SQL Server, you can specify the CTE within the scope of the INSERT statement.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
INSERT INTO TABLE_INS (ID, VAL)
WITH PLUS AS (
SELECT TD.ID, TD.VAL, TI.ID AS NEW
FROM TABLE_DEL TD LEFT OUTER JOIN TABLE_INS TI ON TD.ID=TI.ID
)
SELECT ID, VAL FROM PLUS WHERE NEW IS NULL;
```

Solution:

In SQL Server, place the CTE before the INSERT statement.

SQL Server Example:

```
with plus as (
select td.id, td.val, ti.id as new
from table_del td left outer join table_ins ti on td.id=ti.id
)
insert into table_ins (id, val)
select id, val from plus where new is null;
```

3.4.3.4 Inserting Rows that Contain Structured Type Values

Links: DB2 for Linux UNIX and Windows 10.5.0

When you create a structured type, **DB2 automatically generates a constructor function for the type, and generates mutator and observer methods for the attributes of the type**. You can use these methods to create instances of structured types, and insert these instances into a column of a table.

When you create a structured type, DB2 creates a function of the same name as the type is created. This function has no parameters and returns an instance of the type with all of its attributes set to null.

The function that is created for structured type DB2_OBJECTS.MAN_T, for example, has the following format:

DB2 Example:

```
CREATE FUNCTION MAN_T() RETURNS MAN_T
```

To construct an instance of a type to insert into a column, use the constructor function with the mutator methods. A mutator method exists for each attribute of an object. So, for type MAN_T, DB2 for Linux, UNIX, and Windows creates mutator methods for each of the following attributes: (Name, Age, DOB).

The mutator method DB2 creates for attribute *Age*, for example, has the following format:

DB2 Example:

```
ALTER TYPE MAN_T ADD METHOD AGE(INT) RETURNS MAN_T
```

An observer method exists for each attribute of an object. If the method for an attribute receives an object of the expected type or subtype, the method returns the value of the attribute for that object.

The observer method DB2 creates for the attribute Age of the type MAN_T, for example, has the following format:

DB2 Example:

```
ALTER TYPE MAN_T ADD METHOD AGE() RETURNS INTEGER;
```

Assume that you want to add a new row to the typed table, and that you want that row to contain person information (PERSON_T). Just as with built-in data types, you can add this row using INSERT with the VALUES clause. However, when you specify the value to insert into the PERSON_T, you must invoke the system-provided constructor function (DB2_OBJECTS.PERSON_T()) and observer methods (..Name()..Age ()..DOB) to create the values. To invoke a method on a structured type, use the method invocation operator: '..'.

DB2 Example:

```
CREATE TABLE DB2_OBJECTS.WOMAN (EYES_COLOUR VARCHAR (20), HAIR_COLOUR
VARCHAR(20), PERSON DB2_OBJECTS.PERSON_T)
/
INSERT INTO DB2_OBJECTS.WOMAN (EYES_COLOUR, HAIR_COLOUR, PERSON)
 VALUES('GREEN', 'RED',
             DB2_OBJECTS.PERSON_T()
                  ..NAME ('ANGELINA')
 ..AGE (26)
             ..DOB (DATE('25.12.1988')));
/
SELECT EYES_COLOUR, HAIR_COLOUR, PERSON..NAME, PERSON..AGE, PERSON..DOB
FROM DB2_OBJECTS.WOMAN
```

To avoid having to explicitly call the mutator methods for each attribute of a structured type every time you create an instance of the type, consider defining your own SQL-bodied constructor function that initializes all of the attributes. The following example contains the declaration for an SQL-bodied constructor function for the US_addr_t type:

```
CREATE FUNCTION DB2_OBJECTS.UDF_PERSON_T
(NAME VARCHAR(20), AGE INT, DOB DATE)
 RETURNS DB2_OBJECTS.PERSON_T
 LANGUAGE SQL
 RETURN DB2_OBJECTS.PERSON_T()..NAME(NAME)..AGE(AGE)..DOB(DOB);
/
INSERT INTO DB2_OBJECTS.WOMAN(EYES_COLOUR, HAIR_COLOUR, PERSON)
VALUES('BROWN', 'BLACK', DB2_OBJECTS.UDF_PERSON_T('ANGEL', '20',
'01.01.1995'));
```

There are two common approaches of generating unique values, both of which can be applied to object identifiers:

- with SEQUENCEs
- with the GENERATE_UNIQUE function

Links: DB2 for Linux UNIX and Windows 10.5.0, SQL Server 2014.

3.4.3.5 with sequences

If you need to use numeric values as object identifiers, you can use a SEQUENCE. To begin, use the REF USING clause to specify that the base type of the object reference is to be a numeric type, in the following case, an INT:

DB2 Example:

```
CREATE TYPE DB2_OBJECTS.MAN_T AS (NAME VARCHAR(20), AGE INT, DOB DATE)
```

```
REF USING INT MODE DB2SQL;
/
CREATE TABLE DB2_OBJECTS.MAN OF DB2_OBJECTS.MAN_T (REF IS OID USER
GENERATED)
/
CREATE SEQUENCE DB2_OBJECTS.MANOID AS REF(DB2_OBJECTS.MAN_T)
START WITH 1 INCREMENT BY 1
/
INSERT INTO DB2_OBJECTS.MAN (OID, NAME, AGE, DOB)
VALUES (NEXT VALUE FOR DB2_OBJECTS.MANOID,'ALEX', 26, DATE('25.12.1988'))
```

SQL Server Example:

```
CREATE TABLE MAN (OID INT, NAME VARCHAR(20), AGE INT, DOB DATE
CONSTRAINT UC_OID UNIQUE (OID))
/
CREATE SEQUENCE MANOID START WITH 1 INCREMENT BY 1;
/
INSERT INTO MAN (OID, NAME, AGE, DOB)
VALUES (NEXT VALUE FOR MANOID, 'ALEX', 26, GETDATE())
```

Note: Azure SQL DB doesn't support SEQUENCE objects.

3.4.3.6 with the GENERATE_UNIQUE function

As an alternative to using SEQUENCEs to generate object identifiers, you can use the GENERATE_UNIQUE function. Because GENERATE_UNIQUE returns a CHAR (13) FOR BIT DATA value, ensure that the REF USING clause on the CREATE TYPE statement can accommodate a value of that type. The default of VARCHAR (16) FOR BIT DATA is suitable for this purpose.

```
CREATE TYPE DB2_OBJECTS.WOMAN_T AS (NAME VARCHAR(20), AGE INT, DOB DATE)
MODE DB2SQL;
/
CREATE TABLE DB2_OBJECTS.WOMEN OF DB2_OBJECTS.WOMAN_T (REF IS OID USER
GENERATED)
/
```

DB2 Example:

```
INSERT INTO DB2_OBJECTS.WOMEN (OID, NAME, AGE, DOB)
VALUES(DB2_OBJECTS.WOMAN_T (GENERATE_UNIQUE ()), 'ALEX', 26,
DATE('25.12.1988'));
/
```

DB2 Example:

```
INSERT INTO DB2_OBJECTS.WOMEN (OID, NAME, AGE, DOB)
VALUES(DB2_OBJECTS.WOMAN_T('A'), 'ALEX', 26, DATE('25.12.1988'));
```

SQL Server Example:

```
CREATE TABLE WOMAN (OID VARBINARY(8000), NAME VARCHAR(20), AGE INT, DOB
DATE
CONSTRAINT U_OID UNIQUE (OID))
/
CREATE SEQUENCE MANOID START WITH 1 INCREMENT BY 1;
/
INSERT INTO WOMAN (OID, NAME, AGE, DOB)
VALUES (CAST (NEXT VALUE FOR MANOID AS VARBINARY),'ALEX', 26, GETDATE())
```

Note: Azure SQL DB doesn't support SEQUENCE objects.

3.4.4 UPDATE Statement

3.4.4.1 Issue: Subquery (fullselect) as Object of the UPDATE Operation

In DB2, you can use a subquery (fullselect, updatable view) as the object of the UPDATE operation.

Links:

DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014

DB2 Example:

```
UPDATE (SELECT * FROM A)
SET
      B = B||C
WHERE A=3
```

Solution:

In SQL Server, use the CTE to emulate this functionality by moving the updatable subquery to the CTE.

SQL Server Example:

```
WITH A$SSSMA AS (SELECT * FROM A)
UPDATE A$SSSMA
SET
      B = B + C
WHERE
      A = 3;
```

3.4.4.2 Column Groups in a SET Clause

In DB2, you can update a few columns from a single subquery result.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014

DB2 Example:

```
UPDATE
      EMPLOYEES E
SET
      (ID, LAST_NAME) = (SELECT ID, PNAME FROM EMP_PATRONYMIC P WHERE
P.ID=E.ID),
      (NAME, SAL) = (NAME||' '||LAST_NAME, SAL/10);
```

Solution:

In SQL Server, remove from all subqueries the conditions that reference the up-level objects, and move the subqueries from the SET clause to the CTE.

Generate a single assignment for each column from the column group. Use the expression from the appropriate CTE for the assignment.

To generate join conditions, use the conditions that were removed from the original subqueries.

SQL Server Example:

```
UPDATE E SET
      ID          = (SELECT ID FROM EMP_PATRONYMIC P WHERE P.ID=E.ID),
      LAST_NAME   = (SELECT PNAME FROM EMP_PATRONYMIC P WHERE P.ID=E.ID),
      NAME = NAME+' '+LAST_NAME,
      SAL = SAL/10
FROM
      EMPLOYEES E
```

3.4.5 MERGE Statement

3.4.5.1 Single vs Multiple Occurrences of Each Clause

SQL Server doesn't support multiple uses of each clause in a MERGE statement..

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014

DB2 Example:

```
MERGE INTO DB2_DML_PROC.MERG_IN_EMP E
USING DB2_DML_PROC.USING_EMP E1
ON (E.ID = E1.ID)
WHEN MATCHED AND E.ID < 4 THEN
      UPDATE SET
            E.SAL = E1.SAL
WHEN MATCHED AND E.ID > 4 THEN
      UPDATE SET
            E.SAL = 111
WHEN MATCHED AND E.ID = 4 THEN
      UPDATE SET
            E.SAL = 444
WHEN NOT MATCHED THEN
      INSERT VALUES(E1.ID, E1.NAME, E1.LAST_NAME, E1.SAL);
```

Solution:

In SQL Server, you can use the CASE statement to emulate multiple uses. You can use the condition (1=1) to emulate the Searched WHEN clause.

> **Note:** In the CASE statement you can't use the DEFAULT value. You must write the default value manually.

SQL Server Example:

```
MERGE INTO
      EMPLOYEES E
USING
      EMPL_1 E1
ON
      (E.ID = E1.ID)
WHEN MATCHED AND 1=1 THEN
      UPDATE SET
            E.SAL =
            CASE
                  WHEN E.ID < 4 THEN E1.SAL
                  WHEN E.ID > 4 THEN 111
                  WHEN E.ID = 4 THEN 444
```

```
            END
WHEN NOT MATCHED THEN
      INSERT VALUES(E1.ID, E1.NAME, E1.LAST_NAME, E1.DOB, E1.DEPT, E1.SAL,
E1.JOB);
```

3.4.5.2 SIGNAL Clause in MERGE

In DB2, you can use the SIGNAL exception to make the MERGE statement fail.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014

DB2 Example:

```
MERGE INTO DB2_DML_PROC.MERG_IN_EMP E
USING DB2_DML_PROC.USING_EMP E1
ON    (E.ID = E1.ID)
WHEN MATCHED AND E.ID < 4 THEN
      UPDATE SET
            E.SAL = E1.SAL
WHEN MATCHED AND E.ID > 4 THEN
      UPDATE SET
            E.SAL = 111
WHEN MATCHED AND E.ID = 4 THEN
 SIGNAL SQLSTATE '70102' SET MESSAGE_TEXT = 'ERROR SALARY'
WHEN NOT MATCHED THEN
      INSERT VALUES(E1.ID, E1.NAME, E1.LAST_NAME, E1.SAL);
```

Solution:

In SQL Server, you can generate a conversion exception in UPDATE or INSERT, so that the MERGE statement will fail.

SQL Server Example:

```
IF EXISTS (SELECT * FROM EMPLOYEES E JOIN EMPL_1 E1 ON E.ID = E1.ID
 WHERE E.ID = 4)
THROW 70102, 'ERROR MY LEBEN', 1

MERGE INTO EMPLOYEES E
USING EMPL_1 E1
ON (E.ID = E1.ID)
WHEN MATCHED AND 1=1 THEN
      UPDATE SET
            E.SAL = CASE
                  WHEN E.ID < 2 THEN E1.SAL
                  WHEN E.ID > 2 THEN 77777 END
WHEN NOT MATCHED THEN
      INSERT VALUES(E1.ID, E1.NAME, E1.SAL, E1.JOB);
```

3.4.5.3 Mixed UPDATE and DELETE Clauses in a MERGE Statement

In DB2, a MERGE statement can have mixed UPDATE and DELETE clauses with intersects conditions. This cannot be emulated in SQL Server automaticaly. It requires manual emulation.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014

DB2 Example:

```
MERGE INTO DB2_DML_PROC.MERGE_INTO_TAB D
USING (SELECT ID, VAL FROM DB2_DML_PROC.MERGE_INTO_TAB UNION VALUES (4,
'D') UNION VALUES (5, 'E') UNION VALUES (6, 'F')) AS B (ID, VAL)
ON (D.ID=B.ID)
WHEN MATCHED AND D.ID=1 THEN
UPDATE SET VAL='X'
WHEN MATCHED AND D.ID<3 THEN
DELETE
WHEN MATCHED THEN
UPDATE SET VAL='Y'
WHEN NOT MATCHED THEN
INSERT VALUES (B.ID,B.VAL);
```

Solution:

SQL Server does not have similar functionality; there is no migration solution for this DB2 capability yet.

3.4.5.4 Subquery (fullselect) as Object of a MERGE Operation

In DB2, you can use a subquery (fullselect) as an object of a MERGE operation.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014

DB2 Example:

```
MERGE INTO  (SELECT * FROM DB2_DML_PROC.MERG_IN_EMP) E
USING DB2_DML_PROC.USING_EMP E1
ON    (E.ID = E1.ID)
WHEN MATCHED AND E.ID < 4 THEN
      UPDATE SET
            E.SAL = E1.SAL
WHEN NOT MATCHED THEN
      INSERT VALUES(E1.ID, E1.NAME, E1.LAST_NAME, E1.SAL);
```

Solution:

In SQL Server, you can use a common table expression (CTE) to emulate this functionality.

SQL Server Example:

```
WITH E AS (SELECT * FROM EMPLOYEES)
MERGE INTO
      E
USING
      EMPL_1 E1
ON
      (E.ID = E1.ID)
WHEN MATCHED AND E.ID < 4 THEN
      UPDATE SET
            E.SAL = E1.SAL
WHEN NOT MATCHED THEN
      INSERT VALUES(E1.ID, E1.NAME, E1.LAST_NAME, E1.DOB, E1.DEPT, E1.SAL,
E1.JOB);
```

3.4.5.5 Column Groups in an UPDATE SET Clause

In DB2, in a MERGE statement you can use the UPDATE SET clause to update several columns from a single subquery result.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014

DB2 Example:

```
MERGE INTO DB2_DML_PROC.MERG_IN_EMP E
USING DB2_DML_PROC.USING_EMP E1
ON (E.ID = E1.ID)
WHEN MATCHED AND E.ID < 4 THEN
      UPDATE SET
            (E.NAME, E.SAL) = (SELECT 'DBBEST - '||NAME, SAL*2 FROM
DB2_DML_PROC.MERG_IN_EMP EM WHERE EM.ID = E.ID)
WHEN NOT MATCHED THEN
      INSERT VALUES(E1.ID, E1.NAME, E1.LAST_NAME, E1.SAL);
```

Solution:

In SQL Server, you can generate a single assignment and a duplicate subquery for each column from a column group.

SQL Server Example:

```
MERGE INTO
      EMPLOYEES E
USING
      EMPL_1 E1
ON
      (E.ID = E1.ID)
WHEN MATCHED AND E.ID < 4 THEN
      UPDATE SET
            E.NAME = (SELECT 'DBBEST - '+NAME FROM EMPLOYEES EM WHERE
EM.ID = E.ID),
            E.SAL = (SELECT SAL*2 FROM EMPLOYEES EM WHERE EM.ID = E.ID)
WHEN NOT MATCHED THEN
      INSERT VALUES(E1.ID, E1.NAME, E1.LAST_NAME, E1.DOB, E1.DEPT, E1.SAL,
E1.JOB);
```

DB2 Example:

```
MERGE INTO DB2_DML_PROC.MERGE_INTO_TAB MI
USING (SELECT ID, VAL FROM DB2_DML_PROC.USING_TAB) US
ON (MI.ID = US.ID)
WHEN NOT MATCHED AND US.ID = 3 THEN
INSERT (MI.ID, MI.VAL)
VALUES (US.ID, US.VAL)
WHEN NOT MATCHED AND US.ID = 4 THEN
INSERT (MI.ID, MI.VAL)
VALUES (5, 'BLA-BLA');
```

SQL Server Example:

```
MERGE INTO DBO.MERGE_INTO_TAB MI
USING (SELECT ID, VAL FROM DBO.USING_TAB) US
ON (MI.ID = US.ID)
      WHEN NOT MATCHED THEN
      INSERT VALUES (CASE WHEN US.ID = 3 THEN ID
```

```
            WHEN US.ID = 4 THEN 5 END,
                        CASE WHEN US.ID = 3 THEN VAL
                              WHEN US.ID = 4 THEN 'BLA-BLA' END);
```

DB2 Example:

```
MERGE INTO DB2_DML_PROC.MERG_IN_EMP E
USING DB2_DML_PROC.USING_EMP E1
ON    (E.ID = E1.ID)
WHEN MATCHED AND E.ID < 4 THEN
      UPDATE SET E.SAL = 77777
WHEN MATCHED AND E.ID = 4 THEN
      UPDATE SET E.SAL = 11111
WHEN MATCHED THEN
 UPDATE SET E.SAL = 22222
WHEN NOT MATCHED THEN
      INSERT VALUES(E1.ID, E1.NAME, E1.LAST_NAME, E1.SAL);
```

SQL Server Example:

```
MERGE INTO DBO. MERG_IN_EMP E
USING DBO. USING_EMP E1
ON (E.ID = E1.ID)
WHEN MATCHED THEN
       UPDATE SET E.SAL = CASE WHEN E.ID < 4 THEN 77777
       WHEN E.ID = 4 THEN 11111
                                 WHEN 1=1 THEN 22222 END
WHEN NOT MATCHED THEN
      INSERT VALUES(E1.ID, E1.NAME, E1.LAST_NAME, E1.SAL);
```

DB2 Example:

```
MERGE INTO DB2_DML_PROC.MERG_IN_EMP E
USING DB2_DML_PROC.USING_EMP E1
ON    (E.ID = E1.ID)
WHEN MATCHED AND E.ID < 4 THEN
      UPDATE SET E.SAL = 77777
WHEN MATCHED AND E.ID = 4 THEN
      UPDATE SET E.SAL = 22222
WHEN MATCHED AND E.ID > 4 THEN
      DELETE
WHEN NOT MATCHED THEN
      INSERT VALUES(E1.ID, E1.NAME, E1.LAST_NAME, E1.SAL);
```

SQL Server Example:

```
MERGE INTO DBO.MERG_IN_EMP E
USING DBO.USING_TAB E1
ON (E.ID = E1.ID)
WHEN MATCHED AND 1=1 THEN
       UPDATE SET E.SAL = CASE WHEN E.ID < 4 THEN 77777
 WHEN E.ID = 4 THEN 22222 END
WHEN MATCHED AND E.ID > 4 THEN
DELETE
WHEN NOT MATCHED THEN
      INSERT VALUES(E1.ID, E1.NAME, E1.LAST_NAME, E1.SAL);
```

DB2 Example:

```
MERGE INTO DB2_DML_PROC.MERGE_INTO_TAB D
USING DB2_DML_PROC.USING_TAB AS B
ON (D.ID=B.ID)
WHEN MATCHED AND D.ID=1 THEN
```

```
DELETE
WHEN MATCHED THEN
DELETE
WHEN MATCHED AND D.ID=2 THEN
UPDATE SET VAL='Y'
WHEN NOT MATCHED THEN
INSERT VALUES (B.ID,B.VAL);
```

SQL Server Example:

```
MERGE INTO DBO.MERGE_INTO_TAB D
USING DBO.USING_TAB AS B
ON (D.ID=B.ID)
WHEN MATCHED AND D.ID=1 OR 1=1 THEN
DELETE
WHEN MATCHED AND D.ID=2 THEN
UPDATE SET VAL='Y'
WHEN NOT MATCHED THEN
INSERT VALUES (B.ID,B.VAL);
```

3.4.6 DELETE Statement

3.4.6.1 Correlation Clause in DELETE Statement

In DB2, a correlation clause can be used in a DELETE statement to designate a table, view, nickname, fullselect, or column names.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
DELETE FROM TABLE_DEL AS D (A,B,C,D) WHERE D.A BETWEEN 100 AND 109;
```

Solution:

In SQL Server, replace the column aliases with the column names in a search condition, and remove the correlation clause from the DELETE statement.

SQL Server Example:

```
DELETE FROM TABLE_DEL WHERE ID BETWEEN 100 AND 109;
```

3.4.6.2 Subquery (fullselect) as Object of DELETE

In DB2, the syntax of using a subquery as the object of the DELETE operation is different from that in SQL Server.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
-- DELETING DUPLICATES
DELETE FROM
 (SELECT ROWNUMBER() OVER (PARTITION BY ID ORDER BY IDENT)
 FROM DB2_OBJECTS.IDENT) AS E (RN)
 WHERE RN > 1
```

Solution:

In SQL Server, convert such queries by assigning an alias to the subquery and then adding a FROM clause with this alias.

SQL Server Example:

```
-- DELETING DUPLICATES
DELETE E FROM
(SELECT ROW_NUMBER() OVER (PARTITION BY ID ORDER BY IDENT)
FROM IDENT) AS E (RN)
WHERE RN > 1;
```

3.4.7 Isolation Level and Lock Type

In DB2, the optional isolation-clause (WITH {RR|RS|CS|UR}) specifies the isolation level at which the subselect or fullselect is run, and whether a specific type of lock is to be acquired. The lock-request-clause (USE AND KEEP) applies only to queries and to positioning read operations within an insert, update, or delete operation. The insert, update, and delete operations themselves will run using locking determined by the database manager.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

Isolation levels are enforced by locks, and the type of lock that is used limits or prevents access to the data by concurrent application processes. Declared temporary tables and their rows cannot be locked because they are only accessible to the application that declared them.

The database manager supports three general categories of locks:

Share (S)

Under an S lock, concurrent application processes are limited to read-only operations on the data.

Update (U)

Under a U lock, concurrent application processes are limited to read-only operations on the data, if these processes have not declared that they might update a row. The database manager assumes that the process currently looking at a row might update it.

Exclusive (X)

Under an X lock, concurrent application processes are prevented from accessing the data in any way. This does not apply to application processes with an isolation level of uncommitted read (UR), which can read but not modify the data.

Regardless of the isolation level, the database manager places exclusive locks on every row that is inserted, updated, or deleted. Thus, all isolation levels ensure that any row that is changed by an application process during a unit of work is not changed by any other application process until the unit of work is complete.

The database manager supports four isolation levels.

- Repeatable read (RR)
- Read stability (RS)
- Cursor stability (CS)
- Uncommitted read (UR)

DB2 Example 1:

```
SELECT ID, VAL
FROM TABLE_FF
WHERE ID BETWEEN 65 AND 90
WITH UR
```

DB2 Example 2:

```
SELECT ID, VAL
FROM TABLE_FF
WHERE ID BETWEEN 65 AND 90
WITH RS USE AND KEEP EXCLUSIVE LOCKS
```

Solution:

In SQL Server, use table hints to set the isolation level and lock type. Correspondence between the DB2 and SQL Server isolation levels and lock types is presented in Table 2.

Table 2: Isolation Level and Lock Type Differences Between DB2 and SQL Server

DB2 Isolation Level	Description	SQL Server Table Hint
RR	Repeatable Read	SERIALIZABLE
RS	Read Stability	REPEATABLEREAD
CS	Cursor Stability	READCOMMITTED
UR	Uncommitted Read	READUNCOMMITTED
DB2 Lock Type	**Description**	**SQL Server Table Hint**
SHARE	Concurrent processes can acquire SHARE or UPDATE locks on the data.	TABLOCK
UPDATE	Concurrent processes can acquire SHARE locks on the data, but no concurrent process can acquire an UPDATE or EXCLUSIVE lock.	UPDLOCK
EXCLUSIVE	Concurrent processes cannot acquire a lock on the data.	TABLOCKX

Unlike DB2, SQL Server does not lock the current cursor row in READCOMMITTED isolation level.

SQL Server Example 1:

```
SELECT ID, VAL
FROM TABLE_FF WITH (READUNCOMMITTED)
WHERE ID BETWEEN 65 AND 90
```

SQL Server Example 2:

```
SELECT ID, VAL
FROM TABLE_FF WITH (REPEATABLEREAD, TABLOCKX)
WHERE ID BETWEEN 65 AND 90
```

3.5 Routines

This section describes migration issues for DB2 stored procedures and user-defined functions.

The syntax of DB2's routines language is significantly different from the syntax of SQL Server's procedural language, Transact-SQL. This makes converting code from stored procedures, functions, or triggers a challenge. SSMA, however, can resolve most of the problems related to these conversions.

3.5.1 Procedures

3.5.1.1 Overloaded Procedures

In DB2, two procedures can exist in one schema with the same name but different parameter types or a different number of parameters. SQL Server 2014 – SQL Server does not support this.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0 – DB2 zOS does not support overloaded procedures

DB2 Example 1:

```
-- PROCEDURE
CREATE PROCEDURE PROC_NAME (
      IN A INTEGER,
      IN B INTEGER
)
LANGUAGE SQL
BEGIN
      SQL STATEMENT
…
END;

--OVERLOAD PROCEDURE
CREATE PROCEDURE PROC_NAME ( -- NAME OF PROCEDURE IS LIKE IN THE STATEMENT
ABOVE
      IN A INTEGER,
      IN B INTEGER,
      IN C INTEGER
)
LANGUAGE SQL
BEGIN
      SQL STATEMENT
…
END;
```

Solution:

In SQL Server, you must choose distinct names for procedures that have the same DB2 name but different parameter signatures.

SQL Server Example 1:

```
--PROCEDURE 1
CREATE PROCEDURE PROC_NAME$OVL1 -- WE ADD SUFFIX $OVL1 IN A NAME OF THE
FIRST PROCEDURE
      @A INT,
      @B INT
AS
BEGIN
      SQL STATEMENT
      …
END;

--PROCEDURE2
CREATE PROCEDURE PROC_NAME$OVL2 -- WE ADD SUFFIX $OVL2 IN A NAME OF THE
SECOND PROCEDURE
      @A INT,
      @B INT,
      @C INT
AS
BEGIN
      SQL STATEMENT
      …
END;
```

3.5.1.2 OUT and INOUT Parameters

In DB2, if a procedure has an OUT parameter, it always returns a value in a call variable. In SQL Server this construct cannot return any value SQL Server 2014.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0

If argument has an IN then its value can't change into a body of procedure. If argument has an OUT or an IN OUT then its value can change into body. Also, we can't set a constant value for OUT or (IN OUT)-arguments in external call. For IN-arguments it's should be variable. Using of OUT-argument cannot send value into a body of procedure. Need use IN or IN OUT for this.

DB2 Example:

```
CREATE PROCEDURE PROC_IN_OUT(
      IN    P_IN1       INTEGER DEFAULT 6,
      INOUT P_INOUT     INTEGER,
      OUT   P_OUT             INTEGER
)
LANGUAGE SQL
BEGIN
      SET P_OUT   = P_IN1 * P_INOUT;
      SET P_INOUT = P_IN1 + P_INOUT;
END;


BEGIN ATOMIC
      DECLARE RET1      INTEGER;
      DECLARE RET2      INTEGER;
      SET RET2 = 4;
      CALL PROC_IN_OUT(5, RET2, RET1);
      CALL DBMS_OUTPUT.ENABLE(5000);
      CALL DBMS_OUTPUT.PUT_LINE('RET1 = '||RET1||'; RET2 = '||RET2);
END
SELECT DB2ADMIN.WRAPPER_DBMS_OUTPUT() A FROM SYSIBM.SYSDUMMY1;


BEGIN ATOMIC
      DECLARE RET1      INTEGER;
      DECLARE RET2      INTEGER;
      SET RET2 = 4;
      CALL PROC_IN_OUT(DEFAULT, RET2, RET1);
      CALL DBMS_OUTPUT.ENABLE(5000);
      CALL DBMS_OUTPUT.PUT_LINE('RET1 = '||RET1||'; RET2 = '||RET2);
END
SELECT DB2ADMIN.WRAPPER_DBMS_OUTPUT() A FROM SYSIBM.SYSDUMMY1;


BEGIN ATOMIC
      DECLARE RET1      INTEGER;
      DECLARE RET2      INTEGER;
      SET RET2 = 4;
      CALL PROC_IN_OUT( P_INOUT=>RET2, P_OUT=>RET1);
      CALL DBMS_OUTPUT.ENABLE(5000);
      CALL DBMS_OUTPUT.PUT_LINE('RET1 = '||RET1||'; RET2 = '||RET2);
END
SELECT DB2ADMIN.WRAPPER_DBMS_OUTPUT() A FROM SYSIBM.SYSDUMMY1;
```

Solution:

In SQL Server, when you call a procedure using EXEC for each OUT variable, you must add the OUTPUT keyword.

SQL Server Example:

```
CREATE PROCEDURE PROC_IN_OUT
      @P_IN1            INT   = 6,
      @P_INOUT    INT   OUTPUT,
      @P_OUT      INT   OUTPUT
AS
      SET @P_OUT  = @P_IN1 * @P_INOUT;
      SET @P_INOUT      = @P_IN1 + @P_INOUT;
GO

DECLARE @RET1     INT;
DECLARE @RET2     INT;
SET @RET2 = 4;
EXEC PROC_IN_OUT 5, @RET2 OUTPUT, @RET1 OUTPUT;
SELECT @RET1, @RET2
GO

DECLARE @RET1     INT;
DECLARE @RET2     INT;
SET @RET2 = 4;
EXEC PROC_IN_OUT DEFAULT, @RET2 OUTPUT, @RET1 OUTPUT;
SELECT @RET1, @RET2
GO

DECLARE @RET1     INT;
DECLARE @RET2     INT;
SET @RET2 = 4;
EXEC PROC_IN_OUT @P_INOUT = @RET2 OUTPUT, @P_OUT = @RET1 OUTPUT;
SELECT @RET1, @RET2
GO
```

3.5.2 User-Defined Functions

3.5.2.1 Overloaded Functions

In DB2, two functions can exist in one schema with the same name but different parameter types or a different number of parameters. SQL Server 2014 – SQL Server does not support this.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0 – DB2 zOS does not support overloaded procedures

DB2 Example 1:

```
CREATE FUNCTION FUNC1 (
      IN P1 INTEGER
)RETURNS INTEGER
LANGUAGE SQL
BEGIN
      RETURN P1*P1;
END;

CREATE FUNCTION FUNC1 (
```

```
      IN P1 VARCHAR(100)
)RETURNS INTEGER
LANGUAGE SQL
BEGIN
      RETURN LENGTH(P1);
END;
```

Solution:

In SQL Server, you must choose distinct names for the functions that have the same DB2 name but different parameter signatures.

```
CREATE FUNCTION FUNC1$OVL1( --WE ADD SUFFIX $OVL1 IN A NAME OF THE FIRST
PROCEDURE
      @P1 INT
)RETURNS INTEGER AS
BEGIN
      RETURN @P1*@P1;
END;

CREATE FUNCTION FUNC1$OVL2( --WE ADD SUFFIX $OVL2 IN A NAME OF THE FIRST
PROCEDURE
      @P1 VARCHAR(100)
)RETURNS INTEGER AS
BEGIN
      RETURN LEN(@P1);
END;
```

3.5.2.2 RETURNS ROW Function

In DB2, the RETURNS ROW function specifies that the output of the function is a single row. If the function returns more than one row, an error is raised (SQLSTATE 21505).

Solution:

In SQL Server, create a table-valued function and make sure that the SELECT statement returns one row. See RETURN Statement in Table Functions or Procedures later in this section for more details.

3.5.2.3 MODIFIES SQL DATA Clause

In DB2, the MODIFIES SQL DATA clause is used when a function can change data in a table (using Data Manipulation Language, or DML), either directly or by a call procedure.

Solution:

In SQL Server, user-defined functions (UDFs) can't use DML statements or executing procedures. You can emulate the MODIFIES SQL DATA clause by using an extended stored procedure.

3.5.3 Flow Control Constructs

This section covers differences in behavior between DB2 and SQL Server versions of various commands used in routines that control the flow of execution.

3.5.3.1 CALL Statement for a Procedure

In DB2, the CALL statement can call a procedure or a foreign procedure (but it cannot call a function).

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE PROCEDURE P_EX_CALL ( IN A INTEGER, OUT B INTEGER)
BEGIN
 SET B = A * 2;
END

BEGIN ATOMIC
 DECLARE B INTEGER;
 CALL P_EX_CALL(5, B);
 SET B = NULL;
 CALL P_EX_CALL(NULL, B);
END
```

Solution:

In SQL Server, use the EXECUTE (EXEC) statement to call a stored procedure.

SQL Server Example:

```
CREATE PROCEDURE P_EX_CALL ( @A INTEGER, @B INTEGER OUT)
AS
BEGIN
 SET @B = @A * 2;
END;
GO

DECLARE @B INTEGER;
EXECUTE P_EX_CALL 5, @B OUT;
SET @B = NULL;
EXEC P_EX_CALL NULL, @B OUT;
GO
```

3.5.3.2 CASE Statement

The CASE statement selects an execution path based on multiple conditions. This statement should not be confused with the CASE expression, which allows an expression to be selected based on the evaluation of one or more conditions.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

3.5.3.2.1 The simple-case-statement-when-clause

In DB2, the value of the simple-case-statement-when-clause expression prior to the first WHEN keyword is tested for equality with the value of each expression that follows the WHEN keyword. If the search condition is true, the THEN statement is executed. If the result is unknown or false, processing continues to the next

search condition. If the result does not match any of the search conditions, and an ELSE clause is present, the statements in the ELSE clause are processed.

DB2 Example:

```
CREATE PROCEDURE DB2_FLOW_CONTR_PROC.P_CASE_EX (IN A INTEGER)
BEGIN
 CASE A
 WHEN 5 THEN
 UPDATE DB2_FLOW_CONTR_PROC.CASE_EX_TAB SET AT3 = A * 3.14;
 UPDATE DB2_FLOW_CONTR_PROC.CASE_EX_TAB SET AT1 = AT3;
 WHEN 10 THEN
 UPDATE DB2_FLOW_CONTR_PROC.CASE_EX_TAB SET AT1 = A / 3.14;
 ELSE
 UPDATE DB2_FLOW_CONTR_PROC.CASE_EX_TAB SET AT1 = A + ID * 3.14;
 UPDATE DB2_FLOW_CONTR_PROC.CASE_EX_TAB SET AT3 = A - ID * 3.14;
 END CASE;
END;
```

Solution:

In SQL Server, because the CASE statement cannot be used in routines as control flow statements, use the IF....ELSE statement to convert the DB2 CASE statement. In SQL Server the IF or ELSE condition can affect the performance of only one Transact-SQL statement. To define a statement block, use the control-of-flow keywords BEGIN and END.

SQL Server Example:

```
CREATE PROCEDURE P_CASE_EX1 (@A INTEGER)
AS
BEGIN
 IF (@A = 5)
 BEGIN
 UPDATE TAB1 SET AT3 = @A * 3.14;
 UPDATE TAB1 SET AT1 = AT3;
 END
 ELSE
 IF (@A = 10)
 UPDATE TAB1 SET AT1 = @A / 3.14;
 ELSE
 BEGIN
 UPDATE TAB1 SET AT1 = @A + ID * 3.14;
 UPDATE TAB1 SET AT3 = @A - ID * 3.14;
 END;
END;
GO

EXECUTE P_CASE_EX1 7;
GO
```

3.5.3.3 The searched-case-statement-when-clause

In DB2, the searched-case-statement-when-clause is used to evaluate the search-condition following the WHEN keyword. If it evaluates to true, the statements in the associated THEN clause are processed. If it evaluates to false, or unknown, the next search-condition is evaluated. If no search-condition evaluates to true and an ELSE clause is present, the statements in the ELSE clause are processed.

DB2 Example:

```
CREATE PROCEDURE DB2_FLOW_CONTR_PROC.P_CASE_EX_WHEN (IN A INTEGER)
BEGIN
 CASE
 WHEN A = 5 THEN
 UPDATE DB2_FLOW_CONTR_PROC.CASE_EX_TAB SET AT3 = A * 3.14;
 UPDATE DB2_FLOW_CONTR_PROC.CASE_EX_TAB SET AT1 = AT3;
 WHEN A = 10 THEN
 UPDATE DB2_FLOW_CONTR_PROC.CASE_EX_TAB SET AT1 = A / 3.14;
 ELSE
 UPDATE DB2_FLOW_CONTR_PROC.CASE_EX_TAB SET AT1 = A + ID * 3.14;
 UPDATE DB2_FLOW_CONTR_PROC.CASE_EX_TAB SET AT3 = A - ID * 3.14;
 END CASE;
END;

CALL DB2_FLOW_CONTR_PROC.P_CASE_EX_WHEN(7);
```

Solution:

In SQL Server, the solution for the searched-case-statement is exactly the same as for the simple-case-statement: use the IF....ELSE statement.

SQL Server Example:

```
CREATE PROCEDURE P_CASE_EX2 (@A INTEGER)
AS
BEGIN
 IF (@A = 5)
 BEGIN
 UPDATE TAB1 SET AT3 = @A * 3.14;
 UPDATE TAB1 SET AT1 = AT3;
 END
 ELSE
 IF (@A = 10)
 UPDATE TAB1 SET AT1 = @A / 3.14;
 ELSE
 BEGIN
 UPDATE TAB1 SET AT1 = @A + ID * 3.14;
 UPDATE TAB1 SET AT3 = @A - ID * 3.14;
 END;
END;
GO

EXECUTE P_CASE_EX2 7;
GO
```

3.5.3.4 FOR Statement

In DB2, the FOR statement executes a statement or group of statements for each row of a table using a cursor. The cursor can be declared explicitly or implicitly.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE PROCEDURE DB2_FLOW_CONTR_PROC.FOR_EX
BEGIN
```

```
 DECLARE NEW_AT VARCHAR(100);
 FOR VL AS
 SELECT ID AS CURID, AT1 AS CURAT1, AT3 AS CURAT3
 FROM DB2_FLOW_CONTR_PROC.CASE_EX_TAB
 WHERE ID BETWEEN 1 AND 3
 DO
 SET NEW_AT = CAST((CURID + CURAT1 - CURAT3) AS VARCHAR(100));
 UPDATE DB2_FLOW_CONTR_PROC.CASE_EX_TAB SET AT2 = NEW_AT
 WHERE ID = CURID;
 END FOR;
END;

CALL DB2_FLOW_CONTR_PROC.FOR_EX;
```

Solution:

In SQL Server, you can organize the LOOP statement with the WHILE statement, and then use the @@FETCH_STATUS function to define the end of the loop. The cursor can be declared explicitly only.

SQL Server Example:

```
CREATE PROCEDURE FOR_EX
AS
BEGIN
 DECLARE @NEW_AT VARCHAR(100);
 DECLARE @CURID INTEGER;
 DECLARE @CURAT1 INTEGER;
 DECLARE @CURAT3 FLOAT(53);

 DECLARE V1 CURSOR LOCAL FOR
 SELECT ID AS CURID, AT1 AS CURAT1, AT3 AS CURAT3
 FROM TAB1
 WHERE ID BETWEEN 55 AND 75;

 OPEN V1;

 FETCH V1 INTO @CURID, @CURAT1, @CURAT3;

 WHILE @@FETCH_STATUS = 0
 BEGIN

 SET @NEW_AT = CAST((@CURID + @CURAT1 - @CURAT3) AS VARCHAR(100));

 UPDATE TAB1 SET AT2 = @NEW_AT
 WHERE ID = @CURID;

 FETCH V1 INTO @CURID, @CURAT1, @CURAT3; -- FOR LOOP DATA
 -- PROCESSING
 END;

 CLOSE V1;
 DEALLOCATE V1;
END;
GO

EXECUTE FOR_EX;
```

3.5.3.5 GET DIAGNOSTICS Statement

In DB2, the GET DIAGNOSTICS statement is used to obtain current execution environment information including information about the previous SQL statement (other than a GET DIAGNOSTICS statement) that was executed.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0.

3.5.3.6 ROW_COUNT Variable

In DB2, the ROW_COUNT variable identifies the number of rows associated with the previous SQL statement. If the previous SQL statement is a DELETE, INSERT, or UPDATE statement, ROW_COUNT identifies the number of rows that qualified for the operation. If the previous statement is a PREPARE statement, ROW_COUNT identifies the estimated number of result rows in the prepared statement.

DB2 Example:

```
CREATE PROCEDURE DB2_FLOW_CONTR_PROC.DIAGNOSTIC_ROW_CNT (IN V_ID INTEGER,
OUT ROW_C INTEGER)
BEGIN
 UPDATE DB2_FLOW_CONTR_PROC.CASE_EX_TAB SET AT2 = 'DEFINE'
 WHERE ID = V_ID;
 GET DIAGNOSTICS ROW_C = ROW_COUNT;
END;

BEGIN ATOMIC
 DECLARE ROW_C INTEGER;
 CALL DB2_FLOW_CONTR_PROC.DIAGNOSTIC_ROW_CNT(60, ROW_C);
END
```

Solution:

In SQL Server, you can use the @@ROWCOUNT function to return the number of rows that are associated with the previous SQL statement. However, there is not yet a solution for emulating the GET DIAGNOSTICS statement when the previous statement is a PREPARE statement.

SQL Server Example:

```
CREATE PROCEDURE DIAGNOSTICS_EX (@V_ID INTEGER, @ROW_C INTEGER OUT)
AS
BEGIN
 UPDATE TAB1 SET AT2 = 'DEFINE'
 WHERE ID = @V_ID;
 SET @ROW_C = @@ROWCOUNT;
END;
GO

DECLARE @ROW_C INTEGER;
EXEC DIAGNOSTICS_EX 60, @ROW_C OUT;
GO
```

3.5.3.7 DB2_RETURN_STATUS Variable

In DB2, the DB2_RETURN_STATUS identifies the status value returned from the procedure associated with the previously executed SQL statement, provided that the statement was a CALL statement invoking a procedure that returns a status. If the previous statement is not such a statement, then the value returned has no meaning and could be any integer.

DB2 Example:

```
CREATE PROCEDURE RETURN_VALUE_EX
BEGIN
 RETURN;
END

CREATE PROCEDURE RETURN_STATUS_EX
BEGIN
 DECLARE RETVAL INTEGER;
 CALL RETURN_VALUE_EX;
 GET DIAGNOSTICS RETVAL = DB2_RETURN_STATUS;
 IF RETVAL <> 0 THEN
 RETURN RETVAL;
 ELSE
 CALL WHILE_EX;
 END IF;
END

BEGIN ATOMIC
 CALL RETURN_STATUS_EX;
END
```

Solution:

A SQL Server procedure can return any integer value, and DB2_RETURN_STATUS can be emulated by returning an integer value from the SQL Server procedure (use EXECUTE and RETURN statements).

SQL Server Example:

```
CREATE PROCEDURE RETURN_VALUE_EX
AS
BEGIN
 RETURN;
END;
GO

CREATE PROCEDURE RETURN_STATUS_EX
AS
BEGIN
 DECLARE @RETVAL INTEGER;
 EXECUTE @RETVAL = REPEAT_EX;
 IF (@RETVAL <> 0)
 RETURN @RETVAL;
 ELSE
 EXECUTE WHILE_EX;
END;
GO

EXEC RETURN_STATUS_EX;
GO
```

3.5.3.8 condition-information

In DB2, condition-information specifies that the error or warning information for the previously executed SQL statement is to be returned. If information about an error is needed, the GET DIAGNOSTICS statement must be the first statement specified in the handler that will handle the error. If information about a warning is

needed, and if the handler will get control of the warning condition, the GET DIAGNOSTICS statement must be the first statement specified in that handler.

DB2 Example:

```
CREATE PROCEDURE RETURN_MESSAGE_EX(IN V_ID INTEGER,OUT RETSTR VARCHAR(70))
BEGIN
 DECLARE A INTEGER;

 DECLARE EXIT HANDLER FOR SQLEXCEPTION
 BEGIN
 GET DIAGNOSTICS EXCEPTION 1 RETSTR = MESSAGE_TEXT;
 END;

 SELECT V_ID / 0 INTO A FROM SYSIBM.SYSDUMMY1;
END

BEGIN ATOMIC
 DECLARE RETSTR VARCHAR(70) DEFAULT 'OK!';
 CALL RETURN_MESSAGE_EX(60, RETSTR);
END
```

Solution:

In SQL Server, you can use the TRY…CATCH statement and ERROR_MESSAGE ( ) function to set an output parameter value.

SQL Server Example:

```
CREATE PROCEDURE RETURN_MESSAGE_EX(@V_ID INTEGER, @RETSTR VARCHAR(70) OUT)
AS
BEGIN
 DECLARE @A INTEGER
 BEGIN TRY
 SELECT @A = @V_ID / 0
 END TRY
 BEGIN CATCH
 SET @RETSTR = ERROR_MESSAGE();
 END CATCH
END;
GO

DECLARE @RETSTR VARCHAR(70) = 'OK!'
EXEC RETURN_MESSAGE_EX 60, @RETSTR OUT;
GO
```

3.5.3.9 IF Statement

In DB2, the IF statement selects an execution path based on the evaluation of a condition.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE PROCEDURE IF_EX(IN V CHAR(1), INOUT STAT INTEGER)
BEGIN
 IF V IS NOT NULL THEN
 IF V = 'W' THEN
```

```
 CALL WHILE_EX;
 ELSEIF V = 'R' THEN
 CALL REPEAT_EX;
 ELSE
 CALL ITERATE_EX;
 END IF;
 ELSE
 SET STAT = 1;
 END IF;
 RETURN STAT;
END

BEGIN ATOMIC
 DECLARE STAT INTEGER DEFAULT 0;
 CALL IF_EX ('W', STAT);
END
```

Solution:

In SQL Server, the IF statement differs only in syntax and can be easily converted from DB2.

SQL Server Example:

```
CREATE PROCEDURE IF_EX(@V CHAR(1), @STAT INTEGER OUT)
AS
BEGIN
 IF @V IS NOT NULL
 IF @V = 'W'
 EXECUTE WHILE_EX;
 ELSE IF @V = 'R'
 EXECUTE REPEAT_EX;
 ELSE
 EXECUTE ITERATE_EX;
 ELSE
 SET @STAT = 1;
 RETURN @STAT;
END;
GO

DECLARE @STAT INTEGER = 0
EXECUTE IF_EX 'W', @STAT OUT;
GO
```

3.5.3.10 ITERATE Statement

In DB2, the ITERATE statement causes the flow of control to return to the beginning of a labeled loop.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example 1:

```
CREATE PROCEDURE ITERATE_EX
BEGIN
 DECLARE V_ID INTEGER;
 DECLARE CUR1 CURSOR FOR
 SELECT ID FROM TAB1 ORDER BY ID;
 OPEN CUR1;
```

```
 FETCH_LOOP:

 LOOP -- AFTER ITERATE STATEMENT, CONTROL RESUMES HERE

 FETCH CUR1 INTO V_ID;
 IF V_ID < 64 THEN
 ITERATE FETCH_LOOP;
 ELSEIF V_ID > 70 THEN
 LEAVE FETCH_LOOP;
 ELSE
 UPDATE TAB1 SET AT2 = 'ITERATE'
 WHERE ID = V_ID;
 END IF;
 END LOOP FETCH_LOOP;
 CLOSE CUR1;
END

BEGIN ATOMIC
 CALL ITERATE_EX;
END
```

Solution 1:

In SQL Server, the CONTINUE statement has similar functionality to the ITERATE statement in DB2, if the labeled loop is most inner loop enclosing the ITERATE statement, so use it instead of ITERATE. You do not need to use a label—using just the CONTINUE statement is sufficient.

SQL Server Example 1:

```
CREATE PROCEDURE ITERATE_EX
AS
BEGIN
 DECLARE @V_ID INTEGER;

 DECLARE CUR1 CURSOR LOCAL FOR
 SELECT ID FROM TAB1 ORDER BY ID;
 OPEN CUR1;

 FETCH_LOOP: -- WE KEPT THE LABEL ONLY FOR READABILITY

 WHILE 1 = 1 -- AFTER CONTINUE STATEMENT, CONTROL RESUMES HERE

 BEGIN

 FETCH CUR1 INTO @V_ID;

 IF @@FETCH_STATUS <> 0
 BREAK;

 IF (@V_ID < 64)
 CONTINUE; -- WE DO NOT NEED TO USE A LABEL
 -- IN THE MOST INNER LOOP
 ELSE IF (@V_ID > 70)
 BREAK;
 ELSE
 UPDATE TAB1 SET AT2 = 'ITERATE' WHERE ID = @V_ID;
 END;
 CLOSE CUR1;
 DEALLOCATE CUR1;
```

```
END;
GO

EXECUTE ITERATE_EX;
GO
```

DB2 Example 2:

An ITERATE statement can be issued from a nested block to cause that flow of control to return to the beginning of a loop at a higher level. In the following example, the ITERATE statement within the LAB2 compound statement causes the flow of control to return to the beginning of the LAB1 LOOP statement:

```
LAB1: LOOP -- AFTER ITERATE STATEMENT, CONTROL RESUMES HERE
 SET A = 0;
 LAB2: LOOP
 ...
 LAB3: LOOP
 ...
 ITERATE LAB1; -- MULTILEVEL ITERATE
 ...
 END LOOP LAB3;
 ...
 ITERATE LAB1; -- MULTILEVEL ITERATE
 ...
 END LOOP LAB2;
END LOOP LAB1;
```

Solution 2:

In SQL Server, we need to mark the end of the labeled loop with a constructed label (adding suffix ***_continue*** sounds as a reasonable choice) and use GOTO instead of ITERATE. If loop conversion creates emulated increment block, then this block should follow after the target label.

SQL Server Example 2:

```
LAB1: -- WE KEPT THE LABEL ONLY FOR READABILITY
WHILE 1 = 1

BEGIN
 SET A = 0
 LAB2:
 WHILE 1 = 1
 BEGIN
 ...
 LAB3:
 WHILE 1 = 1
 BEGIN
 ...
 GOTO LAB1_CONTINUE -- MULTILEVEL CONTINUE EMULATED BY GOTO
 ...
 END
 ...
 GOTO LAB1_CONTINUE -- MULTILEVEL CONTINUE EMULATED BY GOTO
 ...
 END

-- AFTER GOTO STATEMENT, CONTROL RESUMES HERE
 LAB1_CONTINUE: -- WE SHOULD PUT LABEL BELOW THE VERY LAST STATEMENT
```

```
 -- OF THE LOOP BODY
END
```

3.5.3.11 LEAVE Statement

In DB2, the LEAVE statement transfers program control out of a loop or a compound statement.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example 1:

```
CREATE PROCEDURE LEAVE_EX
BEGIN
 DECLARE V_ID INTEGER;
 DECLARE CUR1 CURSOR FOR
 SELECT ID FROM TAB1 ORDER BY ID;
 OPEN CUR1;
 FETCH_LOOP:
 LOOP
 FETCH CUR1 INTO V_ID;
 IF V_ID < 64 THEN
 UPDATE TAB1 SET AT2 = 'LEAVE'
 WHERE ID = V_ID;
 ELSE
 LEAVE FETCH_LOOP;
 END IF;
 END LOOP FETCH_LOOP;

-- AFTER LEAVE STATEMENT, CONTROL RESUMES HERE

 CLOSE CUR1;
END

BEGIN ATOMIC
 CALL LEAVE_EX;
END
```

Solution 1:

In SQL Server, the BREAK statement has similar functionality to the LEAVE statement in DB2, so use it instead of LEAVE. You do not need to use a label in the most inner loop - using just the BREAK statement is sufficient.

SQL Server Example 1:

```
CREATE PROCEDURE LEAVE_EX
AS
BEGIN
 DECLARE @V_ID INTEGER;
 DECLARE CUR1 CURSOR LOCAL FOR
 SELECT ID FROM TAB1 ORDER BY ID;
 OPEN CUR1;

 FETCH_LOOP: -- WE KEPT THE LABEL ONLY FOR READABILITY

 WHILE 1 = 1
 BEGIN
 FETCH CUR1 INTO @V_ID;
```

```
 IF @@FETCH_STATUS <> 0
           BREAK;

 IF (@V_ID < 64)
 UPDATE TAB1 SET AT2 = 'LEAVE' WHERE ID = @V_ID;
 ELSE
 BREAK; -- WE DO NOT NEED TO USE A LABEL IN THE MOST INNER LOOP
 END;

-- AFTER BREAK STATEMENT, CONTROL RESUMES HERE

 CLOSE CUR1;
 DEALLOCATE CUR1;
END;
GO

EXECUTE LEAVE_EX;
GO
```

DB2 Example 2:

A LEAVE statement can be issued from a nested block to leave a statement at a higher level. In the following example, the LEAVE statement within the LAB2 compound statement causes the LAB1 LOOP statement to terminate:

```
LAB1: LOOP
 ...
 LAB2: LOOP
 SET A = 0;
 ...
 LAB3: LOOP
 ...
 LEAVE LAB1; -- MULTILEVEL LEAVE
 ...
 END LOOP LAB3;
 ...
 LEAVE LAB1; -- MULTILEVEL LEAVE
 ...
 END LOOP LAB2;
END LOOP LAB1;

-- AFTER LEAVE STATEMENT, CONTROL RESUMES HERE
```

Solution 2:

In SQL Server, we need to mark the very first statement below the labeled loop with a constructed label (adding suffix ***_leave*** sounds as a reasonable choice) and use GOTO instead of LEAVE

SQL Server Example 2:

```
LAB1: -- WE KEPT THE LABEL ONLY FOR READABILITY
WHILE 1 = 1
BEGIN
 SET A = 0
 LAB2:
 WHILE 1 = 1
 BEGIN
 ...
```

```
 LAB3:
 WHILE 1 = 1
 BEGIN
 ...
 GOTO LAB1_LEAVE -- MULTILEVEL LEAVE EMULATED BY GOTO
 ...
 END
 ...
 GOTO LAB1_LEAVE -- MULTILEVEL LEAVE EMULATED BY GOTO
 ...
 END
END

-- AFTER GOTO STATEMENT, CONTROL RESUMES HERE
LAB1_LEAVE: -- WE SHOULD LABEL THE VERY FIRST STATEMENT
 -- BELOW THE LOOP BODY
```

3.5.3.12 LOOP Statement

In DB2, the LOOP statement repeats the execution of a statement or a group of statements.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE PROCEDURE LOOP_EX
BEGIN
 DECLARE V_ID INTEGER DEFAULT 0;
 LAB1:
 LOOP
 SET V_ID = V_ID + 1;
 IF V_ID = 100 THEN
 LEAVE LAB1;
 END IF;
 END LOOP LAB1;
 RETURN V_ID;
END

BEGIN ATOMIC
 CALL LOOP_EX;
END
```

Solution:

In SQL Server, you can organize the LOOP statement with the WHILE statement. In SQL Server the WHILE statement can repeats execution of an SQL statement or statement block. To define a statement block, use the control-of-flow keywords BEGIN and END.

SQL Server Example:

```
CREATE PROCEDURE LOOP_EX
AS
BEGIN
 DECLARE @V_ID INTEGER = 0;
 WHILE 1 = 1
 BEGIN
 SET @V_ID = @V_ID + 1;
 IF @V_ID = 100
```

```
 BREAK;
 END;
 RETURN @V_ID
END;
GO

EXECUTE LOOP_EX;
GO
```

3.5.3.13 REPEAT Statement

In DB2, the REPEAT statement executes a statement or a group of statements **until a search condition is true**.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE PROCEDURE REPEAT_EX
BEGIN
 DECLARE V_ID INTEGER;
 DECLARE END_R SMALLINT DEFAULT 0;
 DECLARE CUR1 CURSOR FOR
 SELECT ID FROM TAB1 ORDER BY ID;
 OPEN CUR1;
 FETCH_LOOP:
 REPEAT
 FETCH CUR1 INTO V_ID;
 IF V_ID < 64 THEN
 UPDATE TAB1 SET AT2 = 'REPEAT'
 WHERE ID = V_ID;
 ELSE
 SET END_R = 1;
 END IF;
 UNTIL END_R <> 0
 END REPEAT FETCH_LOOP;
 CLOSE CUR1;
END

BEGIN ATOMIC
 CALL REPEAT_EX;
END
```

Solution:

In SQL Server, the WHILE statement provides similar functionality to the REPEAT statement in DB2, so use it to emulate REPEAT. Because REPEAT is a cycle with a post-condition, you must check the exit-condition at the end of the cycle.

SQL Server Example:

```
CREATE PROCEDURE REPEAT_EX
AS
BEGIN
 DECLARE @V_ID INTEGER;
 DECLARE @END_R SMALLINT = 0;
 DECLARE CUR1 CURSOR LOCAL FOR
 SELECT ID FROM TAB1 ORDER BY ID;
```

```
 OPEN CUR1;
 WHILE 1 = 1
 BEGIN
 FETCH CUR1 INTO @V_ID;

 IF @@FETCH_STATUS <> 0
            BREAK;

 IF (@V_ID < 64)
 UPDATE TAB1 SET AT2 = 'REPEAT' WHERE ID = @V_ID;
 ELSE
 SET @END_R = 1;
 IF @END_R <> 0
 BREAK;
 END;
 CLOSE CUR1;
 DEALLOCATE CUR1;
END;
GO

EXECUTE REPEAT_EX;
GO
```

3.5.3.14 RESIGNAL Statement

In DB2, the RESIGNAL statement is used within a condition handler to resignal the condition that activated the handler, or to raise an alternate condition so that it can be processed at a higher level. It causes an exception, warning, or not found condition to be returned, along with optional message text.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE PROCEDURE RESIGNAL_EX(IN V INTEGER)
BEGIN
 DECLARE A INTEGER;
 DECLARE CONTINUE HANDLER FOR SQLSTATE '22012'
 BEGIN
 RESIGNAL SQLSTATE '75000'
 SET MESSAGE_TEXT = 'NOT ZERO PARAMETER IS REQUIRED.';
 END;

 SELECT 1000/V INTO A FROM SYSIBM.SYSDUMMY1;
END


BEGIN ATOMIC
 CALL RESIGNAL_EX(0);
END
```

Solution:

In SQL Server, the THROW statement provides similar functionality to the RESIGNAL statement in DB2, so use it to emulate RESIGNAL. For THROW statement error_number is int and must be greater than or equal to 50000 and less than or equal to 2147483647, message is nvarchar(2048). RESIGNAL without parameters should be converted into THROW without parameters. In SQL Server is absent possibility to create and use condition unlike DB2.

SQL Server Example:

```
CREATE PROCEDURE RESIGNAL_EX(@V INTEGER)
AS
BEGIN
 DECLARE @A INTEGER;
 BEGIN TRY
 SELECT @A = 1000/@V;
 END TRY
 BEGIN CATCH
 THROW 50000, 'NOT ZERO PARAMETER IS REQUIRED.', 1;
 END CATCH;
END;
GO

EXEC RESIGNAL_EX 0;
GO
```

3.5.3.15 RETURN Statement

In DB2, the RETURN statement is used to return from a routine. For SQL Server functions or methods, it returns the result of the function or method. For a SQL Server procedure, it optionally returns an integer status value.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

3.5.3.11.1 RETURN Statement in Scalar Functions or Procedures

If the routine is a function or method, then an expression, NULL, or fullselect must be specified after the RETURN keyword, and the data type of the result must be assignable to the RETURNS type of the routine. A procedure cannot return NULL or a fullselect.

DB2 Example:

```
CREATE FUNCTION RETURN_EX(X DOUBLE)
RETURNS DOUBLE
RETURN SIN(X)/COS(X)

SELECT RETURN_EX(100) FROM SYSIBM.SYSDUMMY1
```

Solution:

When a routine is a scalar function, the SQL Server RETURN statement can be used as in DB2.

SQL Server Example:

```
CREATE FUNCTION RETURN_EX(@X FLOAT(53))
 RETURNS FLOAT
AS
BEGIN
 RETURN SIN(@X)/COS(@X);
END;
GO

SELECT DBO.RETURN_EX(100)
```

3.5.3.16 RETURN Statement in Table Functions

DB2 functions can return a table or row. This feature is supported in SQL Server, but the syntax is different. Row should be emulated by table.

DB2 Example1:

```
CREATE FUNCTION TABLE_FUNCTION_EX(V_ID INTEGER)
RETURNS TABLE (ID INTEGER,
 AT1 INTEGER,
 AT2 VARCHAR(200),
 AT3 DOUBLE)
RETURN
 SELECT ID, AT1, AT2, AT3
 FROM TAB1
 WHERE ID = V_ID

SELECT * FROM TABLE(TABLE_FUNCTION_EX(60))
```

DB2 Example2:

```
CREATE TYPE TAB1ROW AS ROW ANCHOR ROW OF TAB1;

CREATE FUNCTION ROW_FUNCTION_EX(V_ID INTEGER)
RETURNS TAB1ROW
BEGIN
 DECLARE R1 TAB1ROW;
 SELECT * INTO R1
 FROM TAB1 WHERE ID = V_ID;
 RETURN R1;
END

CREATE PROCEDURE ROW_EX
BEGIN
 DECLARE R1 TAB1ROW;
 SET R1 = ROW_FUNCTION_EX(60);
END

BEGIN ATOMIC
 CALL ROW_EX;
END
```

Solution:

When a SQL Server function returns a table or row value, the function must be declared with the RETURNS TABLE keyword. In both cases function return the table, but for row emulation the table will be consist one row only. In inline table-valued functions (see Example1), the TABLE return value is defined through a single SELECT statement. Inline functions do not have associated return variables. In multistatement table-valued functions (see Example2), returned variable is a TABLE variable, used to store and accumulate the rows that should be returned as the value of the function.

SQL Server Example1:

```
CREATE FUNCTION TABLE_FUNCTION_EX(@V_ID INTEGER)
RETURNS TABLE AS
RETURN
 SELECT ID, AT1, AT2, AT3
 FROM TAB1
```

```
 WHERE ID = @V_ID;
GO

SELECT * FROM TABLE_FUNCTION_EX(60)
```

SQL Server Example2:

```
CREATE FUNCTION ROW_FUNCTION_EX(@V_ID INTEGER)
RETURNS @ROW_TABLE TABLE
 (ID INTEGER,
 AT1 INTEGER,
 AT2 VARCHAR(100),
 AT3 FLOAT(53)
)
AS
BEGIN
 INSERT @ROW_TABLE
 SELECT * FROM TAB1 WHERE ID = @V_ID;
 RETURN;
END;
GO

SELECT * FROM ROW_FUNCTION_EX(60);
```

3.5.3.17 SIGNAL Statement

In DB2, the SIGNAL statement is used to signal an error or warning condition. It causes an error or warning to be returned with the specified SQLSTATE, along with optional message text.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE PROCEDURE SIGNAL_EX(IN V INTEGER)
BEGIN
 IF V IS NULL THEN
 SIGNAL SQLSTATE '75000'
 SET MESSAGE_TEXT = 'NOT NULL PARAMETER IS REQUIRED.';
 END IF;
END

BEGIN ATOMIC
 CALL SIGNAL_EX(NULL);
END
```

Solution:

In SQL Server, the THROW statement provides similar functionality to the SIGNAL statement in DB2, so use it to emulate SIGNAL. For THROW statement error_number is int and must be greater than or equal to 50000 and less than or equal to 2147483647, message is nvarchar(2048). In SQL Server is absent possibility to create and use condition unlike DB2.

SQL Server Example:

```
CREATE PROCEDURE SIGNAL_EX(@V INTEGER)
AS
BEGIN
 IF @V IS NULL
 THROW 50000, 'NOT NULL PARAMETER REQUIRED.', 1;
```

```
END
GO

EXEC SIGNAL_EX NULL;
GO
```

3.5.3.18 WHILE Statement

The WHILE statement repeats the execution of a statement or group of statements while a specified condition is true.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE PROCEDURE WHILE_EX
BEGIN
 DECLARE V_ID INTEGER;
 DECLARE END_R SMALLINT DEFAULT 1;
 DECLARE CUR1 CURSOR FOR
 SELECT ID FROM TAB1 ORDER BY ID;
 OPEN CUR1;
 FETCH_LOOP:
 WHILE END_R <> 0
 DO
 FETCH CUR1 INTO V_ID;
 IF V_ID < 64 THEN
 UPDATE TAB1 SET AT2 = 'WHILE'
 WHERE ID = V_ID;
 ELSE
 SET END_R = 0;
 END IF;
 END WHILE FETCH_LOOP;
 CLOSE CUR1;
END

BEGIN ATOMIC
 CALL WHILE_EX;
END
```

Solution:

The WHILE statement can be converted almost as is, although it has a slightly different syntax in SQL Server. In SQL Server the WHILE statement can repeats execution of an SQL statement or statement block. To define a statement block, use the control-of-flow keywords BEGIN and END.

SQL Server Example:

```
CREATE PROCEDURE WHILE_EX
AS
BEGIN
 DECLARE @V_ID INTEGER;
 DECLARE @END_R SMALLINT = 1;
 DECLARE CUR1 CURSOR LOCAL FOR
 SELECT ID FROM TAB1 ORDER BY ID;
 OPEN CUR1;
 WHILE @END_R <> 0
 BEGIN
```

```
 FETCH CUR1 INTO @V_ID;

 IF @@FETCH_STATUS <> 0
           BREAK;

 IF (@V_ID < 64)
 UPDATE TAB1 SET AT2 = 'WHILE' WHERE ID = @V_ID;
 ELSE
 SET @END_R = 0;
 END;
 CLOSE CUR1;
 DEALLOCATE CUR1;
END;
GO

EXECUTE WHILE_EX;
GO
```

3.5.4 Cursors

This section covers differences between DB2 and SQL Server versions of cursor implementation, and gives some hints about how to handle cursors during migration.

DB2 supports static, forward-only, and scrollable cursors. There are two types of scrollable cursor: static and keyset-driven. The latter provides the ability to detect or make changes to the underlying data.

SQL Server supports all ANSI-style cursors: static, dynamic, forward only, and keyset-driven. This includes support for INSENSITIVE and SCROLL cursor behavior and for all fetch options (FIRST, LAST, NEXT, PRIOR, RELATIVE, and ABSOLUTE).

Cursor support is available through the following interfaces: ADO.NET, OLE DB, ODBC, DB-Library, and Transact-SQL.

3.5.4.1 Closing a Cursor

In DB2, the CLOSE statement closes a cursor. If a result table was created when the cursor was opened, that table is destroyed.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

The SQL Server CLOSE CURSOR statement closes the cursor but leaves the data structures accessible for reopening.

DB2 Example:

```
CLOSE MYCUR;
```

Solution:

SQL Server requires the DEALLOCATE CURSOR statement to remove the cursor data structures. The DEALLOCATE CURSOR statement differs from CLOSE CURSOR in that a closed cursor can be reopened. The DEALLOCATE CURSOR statement releases all data structures associated with the cursor and removes the definition of the cursor.

SQL Server Example:

```
CLOSE MYCUR;
DEALLOCATE MYCUR;
```

3.5.4.2 Returning a Result Set

DB2 always requires that cursors be used with SELECT statements, regardless of the number of rows requested from the database.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example :

```
CREATE PROCEDURE CURSOR_TO_CLIENT
RESULT SETS 1
BEGIN
 DECLARE MYCUR CURSOR WITH RETURN TO CLIENT FOR
 SELECT ID, VAL
 FROM TABLE_FF;
 OPEN MYCUR;
END;

BEGIN ATOMIC
 CALL CURSOR_TO_CLIENT;
END;
```

Solution:

In SQL Server, a SELECT statement that is not enclosed within a cursor returns rows to the client as a default result set. This is an efficient way to return data to a client application.

SQL Server Example :

```
CREATE PROCEDURE CURSOR_TO_CLIENT
AS
BEGIN
 SELECT ID, VAL
 FROM TABLE_FF;
END;
GO

EXEC CURSOR_TO_CLIENT;
GO
```

3.5.4.3 Auto Closing a Cursor

In DB2, all open cursors are automatically closed when the thread terminates, or when a rollback occurs, or when a commit is done—except if the cursor is defined "with hold." If the cursor is declared "with hold," it will remain open after a commit; otherwise it will be closed at commit time.

Solution:

In SQL Server, the cursor is not automatically closed on commit or on rollback. To emulate DB2 behavior, manually close all cursors after rollback, and close all cursors after commit—except the cursors defined as "with hold."

3.5.4.4 Cursor Loop with Handlers

In DB2, cursor loops can be organized using handlers. After the handler is invoked successfully, control is returned to the SQL statement that follows the statement that raised the exception.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example 1:

```
CREATE PROCEDURE CURSOR_LOOP_1
 BEGIN
 DECLARE S DECIMAL(9,2);
 DECLARE I DECIMAL(9,2);
 DECLARE EXITCODE INTEGER DEFAULT 0;
 DECLARE NO_MORE_ROWS CONDITION FOR SQLSTATE '02000';
 DECLARE MYCUR CURSOR FOR
 SELECT ID
 FROM TABLE_FF ORDER BY ID;
 DECLARE CONTINUE HANDLER FOR NO_MORE_ROWS
 SET EXITCODE = 1;
 SET S = 0;
 OPEN MYCUR;
 FETCH MYCUR INTO I;
 WHILE EXITCODE <> 1
 DO
 SET S = S + I;
 FETCH MYCUR INTO I;
 END WHILE;
 CLOSE MYCUR;
END

BEGIN ATOMIC
 CALL CURSOR_LOOP_1;
END;
```

Solution:

SQL Server does not provide such exceptions to emulate cursor loops using handlers emulating. To emulate this behavior, add the following block after each FETCH statement:

```
IF(@@FETCH_STATUS <> 0)
BEGIN
 SET @EXITCODE = 1 --HANDLER BODY
END
```

SQL Server Example:

```
CREATE PROCEDURE CURSOR_LOOP_1
AS
BEGIN
 DECLARE @S DECIMAL(9,2);
 DECLARE @I DECIMAL(9,2);
 DECLARE @EXITCODE INTEGER = 0;
 DECLARE MYCUR CURSOR LOCAL FOR
 SELECT ID
 FROM TABLE_FF ORDER BY ID;
 SET @S = 0
 OPEN MYCUR
 FETCH MYCUR INTO @I
 IF(@@FETCH_STATUS <> 0)
 BEGIN
 SET @EXITCODE = 1
 END
 WHILE @EXITCODE <> 1
 BEGIN
 SET @S = @S + @I
```

```
 FETCH MYCUR INTO @I
 IF(@@FETCH_STATUS <> 0)
 BEGIN
 SET @EXITCODE = 1
 END
 END
 CLOSE MYCUR
 DEALLOCATE MYCUR
END;
GO

EXEC CURSOR_LOOP_1;
GO
```

3.5.4.5 Cursor Loop with SQLCODE Check

In DB2, a cursor loop can be organized by using a direct check of the SQLCODE after each FETCH statement.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE PROCEDURE CURSOR_LOOP_2
BEGIN
 DECLARE SQLCODE INTEGER DEFAULT 0;
 DECLARE I DECIMAL(9,2);
 DECLARE S DECIMAL(9,2);
 DECLARE MYCUR CURSOR FOR
 SELECT ID
 FROM TABLE_FF ORDER BY ID;
 SET S = 0;
 OPEN MYCUR;
 FETCH MYCUR INTO I;
 WHILE SQLCODE <> 100
 DO
 SET S = S + I;
 FETCH MYCUR INTO I;
 END WHILE;
 CLOSE MYCUR;
END;

BEGIN ATOMIC
 CALL CURSOR_LOOP_2;
END;
```

Solution:

In SQL Server, analyze the string before SQLCODE <> 100. If it is a FETCH statement, then replace SQLCODE <> 100 with @@FETCH_STATUS = 0:

```
 FETCH MYCUR INTO I; --\ FETCH MYCUR INTO @I;
 WHILE SQLCODE <> 100 --/ WHILE @@FETCH_STATUS = 0
```

SQL Server Example:

```
CREATE PROCEDURE CURSOR_LOOP_2
AS
BEGIN
```

```
 DECLARE @S DECIMAL(9,2);
 DECLARE @I DECIMAL(9,2);
 DECLARE MYCUR CURSOR LOCAL FOR
 SELECT ID
 FROM TABLE_FF ORDER BY ID;
 SET @S = 0
 OPEN MYCUR
 FETCH MYCUR INTO @I
 WHILE @@FETCH_STATUS = 0
 BEGIN
 SET @S = @S+@I
 FETCH MYCUR INTO @I
 END
 CLOSE MYCUR
 DEALLOCATE MYCUR
END;
GO

EXEC CURSOR_LOOP_2;
GO
```

3.5.4.6 CURSOR_ROWCOUNT scalar function

The CURSOR_ROWCOUNT function returns the cumulative count of all rows fetched by the specified cursor since the cursor was opened.

Links: DB2 for Linux UNIX and Windows 10.5.0, SQL Server 2014.

DB2 Example:

```
CREATE PROCEDURE CURSOR_ROWCOUNT_FUNC()
BEGIN
 DECLARE ROWS_FETCH BIGINT;
 DECLARE ID DECIMAL(9,2);
 DECLARE EOF INT DEFAULT 0;
 DECLARE MYCUR CURSOR;

 DECLARE CONTINUE HANDLER FOR NOT FOUND
 SET EOF = 1;

 SET MYCUR = CURSOR FOR
 SELECT ID
 FROM TABLE_FF ORDER BY ID;

 OPEN MYCUR;

 FETCH MYCUR INTO ID;

 WHILE EOF <> 1
 DO
 SET ROWS_FETCH = CURSOR_ROWCOUNT(MYCUR);

 INSERT INTO DEBUG(MODULE, MESSAGE)
 VALUES ('CURSOR_ROWCOUNT_FUNC',
 COALESCE(TO_CHAR(ROWS_FETCH), 'NULL'));

 FETCH MYCUR INTO ID;
 END WHILE;
```

```
 CLOSE MYCUR;
END;

BEGIN ATOMIC
 CALL CURSOR_ROWCOUNT_FUNC;
END;
```

Solution:

In SQL Server, you could emulate DB2 CURSOR_ROWCOUNT scalar function with the simple variable. After each FETCH increase the variable.

SQL Server Example:

```
CREATE PROCEDURE CURSOR_ROWCOUNT_FUNC
AS
BEGIN
 DECLARE @ROWS_FETCH BIGINT = 0;
 DECLARE @ID DECIMAL(9,2);
 DECLARE @EOF INT = 0;
 DECLARE @MYCUR CURSOR;

 SET @MYCUR = CURSOR FOR
 SELECT ID
 FROM TABLE_FF ORDER BY ID;

 OPEN @MYCUR;

 FETCH @MYCUR INTO @ID;
 IF (@@FETCH_STATUS <> 0)
 SET @EOF = 1

 WHILE @EOF <> 1
 BEGIN
 SET @ROWS_FETCH += 1;

 PRINT @ROWS_FETCH

 FETCH @MYCUR INTO @ID;
 IF (@@FETCH_STATUS <> 0)
 SET @EOF = 1
 END;

 CLOSE @MYCUR;
 DEALLOCATE @MYCUR;
END;
GO

EXEC CURSOR_ROWCOUNT_FUNC;
GO
```

3.5.5 Variables

3.5.5.1 Variable Declaration

In DB2, local variable support in SQL procedures allows you to assign and retrieve SQL values in support of SQL procedure logic. Variables in SQL procedures are defined by using the DECLARE statement. When declaring a variable, you can specify a default value using the DEFAULT clause.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
BEGIN ATOMIC
 DECLARE V_RCOUNT INTEGER;
 DECLARE V_MAX DECIMAL (9,2);
 DECLARE V_ADATE, V_ANOTHER DATE;
 DECLARE V_TOTAL INTEGER DEFAULT 0;
END;
```

Solution:

In SQL Server, variables are declared in the body of a batch or procedure with the DECLARE statement. Variable names must begin with an 'at' (@) sign. You can declare variables by list, but data type must be defined for each variable in this list. You can assign a value to the variable in-line. The value can be a constant or an expression, but it must either match the variable declaration type or be implicitly convertible to that type.

SQL Server Example:

```
DECLARE @V_RCOUNT INTEGER;
DECLARE @V_MAX DECIMAL (9,2);
DECLARE @V_ADATE DATE, @V_ANOTHER DATE;
DECLARE @V_TOTAL INTEGER = 0;
```

3.6 Exceptions, Handlers, and Conditions

The elements of exception handling in SQL Server differ significantly from its DB2 counterparts. Here we give a few examples about handling these differences.

3.6.1 EXIT Handlers

In DB2, the EXIT handler executes SQL PL statements in the handler. After that, the handler continues execution at the end of the compound statement in which it was declared.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014

DB2 Example:

```
CREATE PROCEDURE PRC_EXIT
LANGUAGE SQL
BEGIN ATOMIC
      DECLARE RETURN_MESSAGE  VARCHAR(100)      DEFAULT 0;
      DECLARE RETURN_CODE     INTEGER     DEFAULT 0;
      DECLARE V_ID            INTEGER;
      DECLARE V_NAME          VARCHAR(100);
      DECLARE C1 CURSOR FOR SELECT ID, NAME FROM EMPLOYEES;
```

```
      /*** HANDLER DECLARATION ***/
      DECLARE EXIT HANDLER FOR NOT FOUND BEGIN
            /*** HANDLER STATEMENTS ***/
            SET RETURN_MESSAGE = 'ERROR. ROWS HAVE BEEN COMPLITED.';
            SET RETURN_CODE = 200;
            INSERT INTO MES(GROUP_ID, TEXT)VALUES('PRC_EXIT',
'RETURN_MESSAGE='||RETURN_MESSAGE);
            CLOSE C1;
      END;

      /*** CODE STATEMENTS ***/
      OPEN C1;
      LOOP
            FETCH C1 INTO V_ID, V_NAME;
            INSERT INTO MES(GROUP_ID, TEXT)VALUES('PRC_EXIT', 'GOOD:
V_NAME='||V_NAME);
      END LOOP;
      CLOSE C1;
END;
```

Solution:

In SQL Server, use a TRY/CATCH block. Because exception doesn't occur in SQL Server after unsuccessful fetch data from cursor, so we need to check the @@fetch_status variable and generate a manual exception. Code-statements need to place in try-block and handler-statements need to place in catch-block.

SQL Server Example:

```
/*** CREATE PROCEDURE ***/
CREATE PROCEDURE PROC_EXIT AS
BEGIN
      DECLARE @RETURN_MESSAGE       VARCHAR(100)      = 0;
      DECLARE @RETURN_CODE          INT         = 0;
      DECLARE @V_ID                 INT;
      DECLARE @V_NAME         VARCHAR(100);
      DECLARE C1 CURSOR FOR SELECT ID, NAME FROM EMPLOYEES;
      BEGIN TRY
            /*** CODE STATEMENTS ***/
            OPEN C1;
            WHILE 1=1 BEGIN
                  FETCH C1 INTO @V_ID, @V_NAME; IF (@@FETCH_STATUS <> 0)
THROW 51000, '', 1; /* THIS CLAUSE GENERATES USER'S EXCEPTION */
                  INSERT INTO MES(GROUP_ID, TEXT)VALUES('PRC_EXIT', 'GOOD:
V_NAME='+@V_NAME);
            END;
            CLOSE C1;
            DEALLOCATE C1;
      END TRY
      BEGIN CATCH /* THIS BLOCK PROCESSES THAT EXCEPTION */
            /*** HANDLER STATEMENTS ***/
            SET @RETURN_MESSAGE = 'ERROR. ROWS HAVE BEEN COMPLITED.';
            SET @RETURN_CODE = 200;
            INSERT INTO MES(GROUP_ID, TEXT)VALUES('PRC_EXIT', 'ERROR:
RETURN_MESSAGE='+@RETURN_MESSAGE);
            CLOSE C1;
            DEALLOCATE C1;
      END CATCH;
END;
```

```
/*** CALL PROCEDURE BLOCK ***/
DELETE MES
BEGIN
      EXEC PROC_EXIT;
END;

/*** SEE THE RESULT ***/
SELECT * FROM MES
```

3.6.2 UNDO Handlers

In DB2, an UNDO handler is similar to the EXIT handler: it continues with execution at the end of the compound statement in which it was declared. However, in the UNDO handler each executed statement is rolled back in this compound statement. The UNDO handler can only be used in ATOMIC compound statements.

DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014

DB2 Example:

```
/*** CREATE PROCEDURE ***/
CREATE PROCEDURE PRC_UNDO
LANGUAGE SQL
BEGIN ATOMIC
      DECLARE RETURN_MESSAGE  VARCHAR(100) DEFAULT 0;
      DECLARE RETURN_CODE     INTEGER DEFAULT 0;
      DECLARE V_ID            INTEGER;
      DECLARE V_NAME          VARCHAR(100);
      DECLARE C1 CURSOR FOR SELECT ID, NAME FROM EMPLOYEES;

      /*** HANDLER DECLARATION ***/
      DECLARE UNDO HANDLER FOR NOT FOUND BEGIN
            /*** HANDLER STATEMENTS ***/
            SET RETURN_MESSAGE = 'ERROR. ROWS HAVE BEEN COMPLITED.';
            SET RETURN_CODE = 200;
            INSERT INTO MES(GROUP_ID, TEXT)VALUES('PRC_UNDO', 'ERROR:
RETURN_MESSAGE='||RETURN_MESSAGE);
            CLOSE C1;
      END;

      /*** CODE STATEMENTS ***/
      OPEN C1;
      LOOP
      FETCH C1 INTO V_ID, V_NAME;
            INSERT INTO MES(GROUP_ID, TEXT)VALUES('PRC_UNDO', 'GOOD:
V_NAME='||V_NAME);
      END LOOP;
      CLOSE C1;
END;

/*** TEST RESULT ***/
DELETE MES;
BEGIN ATOMIC
      CALL DB2_OBJECTS.PRC_UNDO;
END;
SELECT * FROM MES ORDER BY ID;
```

Solution:

In SQL Server, specify a savepoint after the BEGIN statement and the ROLLBACK statement in a CATCH block.

SQL Server Example:

```
/*** CREATE PROCEDURE ***/
ALTER PROCEDURE PROC_UNDO AS
BEGIN
      DECLARE @RETURN_MESSAGE       VARCHAR(100)      = 0;
      DECLARE @RETURN_CODE          INT         = 0;
      DECLARE @V_ID                 INT;
      DECLARE @V_NAME         VARCHAR(100);
      DECLARE @TRANCOUNTER          INT; /* THIS VARIABLE STORES
TRANSACTION STATE */
      DECLARE C1 CURSOR FOR SELECT ID, NAME FROM EMPLOYEES;

      SET @TRANCOUNTER = @@TRANCOUNT;
      IF (@TRANCOUNTER > 0) SAVE TRAN SSMAPTN; ELSE BEGIN TRAN SSMAPTN; /*
FOR UNDO-EMULATION WE NEED TO PLACE SAVEPOINT-CLAUSE BEFORE TRY-BLOCK */

      BEGIN TRY
            /*** CODE STATEMENTS ***/
            OPEN C1;
            WHILE 1=1 BEGIN
                  FETCH C1 INTO @V_ID, @V_NAME; IF (@@FETCH_STATUS <> 0)
THROW 51000, '', 1; /* THIS CLAUSE GENERATES USER'S EXCEPTION */
                  INSERT INTO MES(GROUP_ID, TEXT)VALUES('PRC_UNDO', 'GOOD:
V_NAME='+@V_NAME);
            END;
            CLOSE C1;
            DEALLOCATE C1;
      END TRY
      BEGIN CATCH

            IF (@TRANCOUNTER = 0) /* FOR UNDO-EMULATION WE NEED TO PLACE
ROLLBACK-CLAUSE BEFORE HANDLER STATEMENTS */
                  ROLLBACK TRAN;
            ELSE
                  IF (XACT_STATE() <> -1) ROLLBACK TRAN SSMAPTN;

            /*** HANDLER STATEMENTS ***/
            SET @RETURN_MESSAGE = 'ERROR. ROWS HAVE BEEN COMPLITED.';
            SET @RETURN_CODE = 200;
            INSERT INTO MES(GROUP_ID, TEXT)VALUES('PRC_UNDO', 'ERROR:
RETURN_MESSAGE='+@RETURN_MESSAGE);
            CLOSE C1;
            DEALLOCATE C1;
      END CATCH;
END;


/*** TEST RESULT ***/
DELETE MES
BEGIN
      BEGIN TRAN;
      INSERT INTO MES(GROUP_ID, TEXT)VALUES('PRC_UNDO', 'FIRST');
      EXEC PROC_UNDO;
```

```
END;
SELECT * FROM MES ORDER BY ID;
```

3.6.3 CONTINUE Handlers

In DB2, the CONTINUE handler is the opposite of the EXIT handler. The CONTINUE handler continues execution at the statement that *follows* the statement that raised the exception.

DB2 Example:

```
CREATE PROCEDURE DB2_OBJECTS.CONT
LANGUAGE SQL
BEGIN ATOMIC
      DECLARE X INTEGER;
      DECLARE SQLSTATE CHAR(5) DEFAULT '00000';       /* IT IS REGISTER
SQLCODE */
      DECLARE SQLCODE INTEGER DEFAULT 0;        /* IT IS REGISTER SQLSTATE
*/
      DECLARE RSTATE CHAR(5) DEFAULT '00000';
      DECLARE RCODE INTEGER DEFAULT 0;
      DECLARE ZERO CONDITION FOR SQLSTATE '22012';

      DECLARE RESULT INTEGER;

      /* PROCESSING OF AN EXCEPTION "DIVISION BY ZERO" */
      DECLARE CONTINUE HANDLER FOR ZERO BEGIN
            SET (RCODE, RSTATE) = (SQLCODE, SQLSTATE); /* ON THE FIRST
STEP WE SHOULD SAVE SQLCODE AND SQLSTATE IN OTHER VARIABLES BECAUSE THEY
ARE CLEARED AFTER EACH STATEMENT. */
            INSERT INTO MES(GROUP_ID, TEXT)VALUES('PRC_EXIT',
'RCODE='||RCODE||'; RSTATE='||RSTATE||'; ERRM='||SYSPROC.SQLERRM (RSTATE,
'', '', 'EN_US', 1));
            INSERT INTO MES(GROUP_ID, TEXT)VALUES('PRC_EXIT',
'RCODE='||RCODE||'; RSTATE='||RSTATE||'; ERRM='||SYSPROC.SQLERRM
(REPLACE(RCODE,'-','SQL'),', ';', 'EN_US', 1));
      END;

      /* FIRST STATEMENT */
      SET X = 1/0; /* MAKE AN EXCEPTION DIVISION BY ZERO */

      /* SECOND STATEMENT */
      SET RESULT = (SELECT CAST('12' AS INTEGER) FROM SYSIBM.SYSDUMMY1);

      /* THIRD STATEMENT */
      SET RESULT = (SELECT COS(0)/SIN(0) FROM SYSIBM.SYSDUMMY1); /* MAKE
AN EXCEPTION DIVISION BY ZERO */
END;
```

Solution:

In SQL Server, use the same syntax as in the EXIT handler, but wrap every single statement in a TRY/CATCH block.

SQL Server Example:

```
CREATE PROCEDURE CONT AS
BEGIN
      DECLARE @X  INT;
```

```
      DECLARE @V_ERR    NUMERIC(38);
      DECLARE @RCODE    VARCHAR(10);
      DECLARE @RSTATE   VARCHAR(10);
      DECLARE @SQLERRM  VARCHAR(8000);
      DECLARE @SQLERRM1 VARCHAR(8000);
      DECLARE @SSMA$ZERO_CONDITION VARCHAR(10) = '22012'; /* DECLARE
CONDITION */

      DECLARE @RESULT INT;

      /* FIRST STATEMENT */
      BEGIN TRY
            SET @X = 1/0; /* MAKE AN EXCEPTION DIVISION BY ZERO */
      END TRY
      BEGIN CATCH
            SET @V_ERR   = ERROR_NUMBER();
            SET @RCODE   = SSMA_DB2.SQLCODE(@V_ERR);
            SET @RSTATE  = SSMA_DB2.SQLSTATE(@V_ERR);
            SET @SQLERRM = SSMA_DB2.SQLERRM(@RSTATE, '', '', 'EN_US', 1);
            SET @SQLERRM1= SSMA_DB2.SQLERRM (REPLACE(@RCODE,'-','SQL'),
'', '', 'EN_US', 1);
            IF @RSTATE != @SSMA$ZERO_CONDITION BEGIN /* CHECK ON CONDITION
*/
                  THROW;
            END;
            INSERT INTO MES(GROUP_ID, TEXT)VALUES('PRC_EXIT',
'RCODE='+@RCODE+'; RSTATE='+@RSTATE+'; ERRM='+@SQLERRM);
            INSERT INTO MES(GROUP_ID, TEXT)VALUES('PRC_EXIT',
'RCODE='+@RCODE+'; RSTATE='+@RSTATE+'; ERRM='+@SQLERRM1);
      END CATCH;

      /* SECOND STATEMENT */
      BEGIN TRY
            SELECT RESULT = CAST('12' AS INTEGER);
      END TRY
      BEGIN CATCH
            SET @V_ERR = ERROR_NUMBER();
            SET @RCODE = [SSMA_DB2].[SQLCODE](@V_ERR);
            SET @RSTATE = [SSMA_DB2].[SQLSTATE](@V_ERR);
            SET @SQLERRM = SSMA_DB2.SQLERRM(@RSTATE, '', '', 'EN_US', 1);
            SET @SQLERRM1= SSMA_DB2.SQLERRM (REPLACE(@RCODE,'-','SQL'),
'', '', 'EN_US', 1);
            IF @RSTATE != @SSMA$ZERO_CONDITION BEGIN /* CHECK ON
CONDITION. */
                  THROW;
            END;
            INSERT INTO MES(GROUP_ID, TEXT)VALUES('PRC_EXIT',
'RCODE='+@RCODE+'; RSTATE='+@RSTATE+'; ERRM='+@SQLERRM);
            INSERT INTO MES(GROUP_ID, TEXT)VALUES('PRC_EXIT',
'RCODE='+@RCODE+'; RSTATE='+@RSTATE+'; ERRM='+@SQLERRM1);
      END CATCH;



      /* THIRD STATEMENT */
      BEGIN TRY
            SELECT RESULT = COS(0)/SIN(0); /* MAKE AN EXCEPTION DIVISION
BY ZERO */
      END TRY
```

```
	BEGIN CATCH
		SET @V_ERR = ERROR_NUMBER();
		SET @RCODE = [SSMA_DB2].[SQLCODE](@V_ERR);
		SET @RSTATE = [SSMA_DB2].[SQLSTATE](@V_ERR);
		SET @SQLERRM = SSMA_DB2.SQLERRM(@RSTATE, '', '', 'EN_US', 1);
		SET @SQLERRM1= SSMA_DB2.SQLERRM (REPLACE(@RCODE,'-','SQL'),
'', '', 'EN_US', 1);
		IF @RSTATE != @SSMA$ZERO_CONDITION BEGIN /* CHECK ON CONDITION
*/
			THROW;
		END;
		INSERT INTO MES(GROUP_ID, TEXT)VALUES('PRC_EXIT',
'RCODE='+@RCODE+'; RSTATE='+@RSTATE+'; ERRM='+@SQLERRM);
		INSERT INTO MES(GROUP_ID, TEXT)VALUES('PRC_EXIT',
'RCODE='+@RCODE+'; RSTATE='+@RSTATE+'; ERRM='+@SQLERRM1);
	END CATCH;
END;
GO
```

3.7 Dynamic SQL

DB2 has specifics intended for creating dynamic statements that can be embedded only in an application program: the DESCRIBE statement and the SQL Descriptor Area (SQLDA). DB2 has four general types of dynamic SQL: caching dynamic SQL, non-SELECT dynamic SQL using EXECUTE, and dynamic SQL for fixed-list and varying-list SELECT.

3.7.1 DESCRIBE Statement

In DB2, the DESCRIBE statement obtains information about a prepared statement. DESCRIBE can be embedded only in an application program. In dynamic SQL, the DESCRIBE statement is commonly used with SQLDA; its usage is specific to DB2.

Solution:

In SQL Server, there is no direct solution to emulate the DESCRIBE statement or SQLDA. Instead, change these statements to another type of dynamic SQL in an application using varying-list SELECT, and then convert.

3.7.2 PREPARE Statement

in DB2, a PREPARE statement is used to save prepared dynamic statements in a cache pool that all application processes can use to save and retrieve prepared dynamic statements. After an SQL statement has been prepared and is automatically saved in the cache, subsequent PREPARE requests for that same SQL statement can avoid the costly preparation process by using the statement that is in the cache.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE PROCEDURE PREPARE_EX
BEGIN
	DECLARE STMT VARCHAR(1000);
	SET STMT = 'INSERT INTO TAB1(ID, AT1, AT2, AT3) VALUES(100, 1,
''1'', 1)';
	PREPARE STMT_EX FROM STMT;
	EXECUTE STMT_EX;
END;
```

```
BEGIN ATOMIC
      CALL PREPARE_EX;
END;
```

Solution:

SQL Server prepares and caches SQL statement only after execution. Therefore you can remove the PREPARE statement, and replace the EXECUTE statement syntax with the SQL Server EXECUTE statement.

SQL Server Example:

```
DECLARE @STMT VARCHAR(1000);
SET @STMT = 'INSERT INTO TAB1(ID, AT1, AT2, AT3) VALUES(100, 1, ''1'',
1)';
EXECUTE (@STMT);
```

3.7.3 EXECUTE Statement

Non-SELECT dynamic SQL uses PREPARE and EXECUTE statements to issue SQL statements. This type of dynamic SQL cannot issue the SELECT statement. Non-SELECT dynamic SQL can provide huge performance benefits over using the EXECUTE IMMEDIATE statement (see section 3.7.4 EXECUTE IMMEDIATE Statement) because with EXECUTE you can use a parameter marker (or bind variable), which is a placeholder for host variables in a dynamic SQL statement.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE PROCEDURE DYNAMIC_EX (
 IN DYN_T VARCHAR(20), IN DYN_V VARCHAR(100), IN VAL INTEGER)
BEGIN
 DECLARE STMT VARCHAR(1000);
 SET STMT = 'INSERT INTO ' || DYN_T ||
 ' (ID, AT1, AT2, AT3) VALUES( ?, ' || DYN_V || ')';
 PREPARE STMT_EX FROM STMT;
 WHILE (VAL < 200)
 DO
 EXECUTE STMT_EX USING VAL;
 SET VAL = VAL + 10;
 END WHILE;
END;

BEGIN ATOMIC
 DECLARE TLIST VARCHAR(20);
 DECLARE VLIST VARCHAR(100);
 DECLARE VAL INTEGER;
 SET TLIST = 'TAB1';
 SET VLIST = '1, ''1'', 1';
 SET VAL = 110;
 CALL DYNAMIC_EX(TLIST, VLIST, VAL);
END;
```

Solution:

In SQL Server, you can use the EXECUTE statement without previous use of the PREPARE statement. To convert a non-SELECT or caching dynamic SQL statement, you need to generate a dynamic string using a local variable and execute the string using the EXECUTE statement.

SQL Server Example:

```
CREATE PROCEDURE DYNAMIC_EX (
 @DYN_T VARCHAR(20), @DYN_V VARCHAR(100), @VAL INTEGER)
AS
BEGIN
 DECLARE @STMT VARCHAR(1000);
 WHILE (@VAL < 200)
 BEGIN
 SET @STMT = 'INSERT INTO ' + @DYN_T +
 ' (ID, AT1, AT2, AT3) VALUES( ' + CAST(@VAL AS VARCHAR(10)) +
 ', ' + @DYN_V + ')';
 EXECUTE (@STMT);
 SET @VAL = @VAL + 10;
 END;
END;
GO

DECLARE @TLIST VARCHAR(20);
DECLARE @VLIST VARCHAR(100);
DECLARE @VAL INTEGER;
SET @TLIST = 'TAB1';
SET @VLIST = '1, ''1'', 1';
SET @VAL = 110;
EXECUTE DYNAMIC_EX @TLIST, @VLIST, @VAL;
GO
```

3.7.4 EXECUTE IMMEDIATE Statement

DB2 includes the EXECUTE IMMMEDIATE call with a simple text parameter. It is used if one of the statements allowed for dynamic SQL is supposed to be constructed and executed.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE PROCEDURE DYNAMIC_IMMEDIATE_EX (
 IN DYN_T VARCHAR(20), IN DYN_V VARCHAR(100))
BEGIN
 DECLARE STMT VARCHAR(1000);
 SET STMT = 'INSERT INTO ' || DYN_T || ' ' || DYN_V;
 EXECUTE IMMEDIATE STMT;
END;

BEGIN ATOMIC
 DECLARE TLIST VARCHAR(20);
 DECLARE VLIST VARCHAR(100);
 SET TLIST = 'TAB1';
 SET VLIST = '(ID, AT1, AT2, AT3) VALUES(200, 1, ''1'', 1)';
 CALL DYNAMIC_IMMEDIATE_EX (TLIST, VLIST);
END;
```

Solution:

In SQL Server, the EXECUTE IMMEDIATE statement can be converted directly to the EXECUTE statement. There are only minor differences between DB2 and SQL Server in the syntax, such as in declaring variables and calling procedures.

SQL Server Example:

```
CREATE PROCEDURE DYNAMIC_IMMEDIATE_EX (@DYN_T VARCHAR(20), @DYN_V
VARCHAR(100))
AS
 DECLARE @STMT VARCHAR(1000);
 SET @STMT = 'INSERT INTO ' + @DYN_T + ' ' + @DYN_V;
 EXECUTE (@STMT);
GO

DECLARE @TLIST VARCHAR(20);
DECLARE @VLIST VARCHAR(100);
SET @TLIST = 'TAB1';
SET @VLIST = '(ID, AT1, AT2, AT3) VALUES(200, 1, ''1'', 1)';
EXECUTE DYNAMIC_IMMEDIATE_EX @TLIST, @VLIST;
GO
```

3.7.5 Dynamic SQL for a Fixed-List SELECT Statement

In DB2, you can use dynamic SQL for a fixed-list SELECT statement to explicitly prepare and execute SQL SELECT statements when the columns to be retrieved are known and unchanging. To do this you must use a cursor to fetch the results into local variables.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE PROCEDURE DYNAMIC_FIXED_LIST_EX (IN VAL INTEGER, OUT V_ID INTEGER,
OUT V_AT1 INTEGER, OUT V_AT2 VARCHAR(200), OUT V_AT3 DOUBLE)
BEGIN
 DECLARE STMT VARCHAR(1000);
 DECLARE CUR1 CURSOR FOR STMT_EX;
 SET STMT = 'SELECT ID, AT1, AT2, AT3 FROM TAB1 WHERE ID = ?';
 PREPARE STMT_EX FROM STMT;
 OPEN CUR1 USING VAL;
 FETCH CUR1 INTO V_ID, V_AT1, V_AT2, V_AT3;
 CLOSE CUR1;
END;

BEGIN ATOMIC
 DECLARE VAL, VID, VAT1 INTEGER;
 DECLARE VAT2 VARCHAR(200);
 DECLARE VAT3 DOUBLE;
 SET VAL = 60;
 CALL DYNAMIC_FIXED_LIST_EX(VAL, VID, VAT1, VAT2, VAT3);
END;
```

Solution:

In SQL Server, to convert dynamic SQL for a fixed-list SELECT statement you can use the sp_executesql stored procedure, This procedure makes it possible to create a cursor based on a dynamic string, and to use local variables as parameters of this cursor.

SQL Server Example:

```
CREATE PROCEDURE DYNAMIC_FIXED_LIST_EX(
 @V_VAL INTEGER,
 @V_ID INTEGER OUTPUT,
 @V_AT1 INTEGER OUTPUT,
```

```
 @V_AT2 VARCHAR(200) OUTPUT,
 @V_AT3 FLOAT(53) OUTPUT
)
AS
DECLARE @STMT NVARCHAR(1000);
DECLARE @PARAMS_DEFINITION NVARCHAR(500);
DECLARE @CUR1 CURSOR;
SET @STMT = (N'SET @CUR = CURSOR LOCAL FOR '+
 'SELECT ID, AT1, AT2, AT3 FROM TAB1 WHERE ID = @VAL; OPEN @CUR');
SET @PARAMS_DEFINITION = N'@VAL INTEGER, @CUR CURSOR OUTPUT';
EXECUTE SP_EXECUTESQL
 @STMT,
 @PARAMS_DEFINITION,
 @VAL = @V_VAL,
 @CUR = @CUR1 OUTPUT;
FETCH @CUR1 INTO @V_ID, @V_AT1, @V_AT2, @V_AT3;
CLOSE @CUR1;
DEALLOCATE @CUR1
GO

DECLARE @VAL INTEGER, @VID INTEGER, @VAT1 INTEGER;
DECLARE @VAT2 VARCHAR(200);
DECLARE @VAT3 FLOAT(53);
SET @VAL = 60;
EXECUTE DYNAMIC_FIXED_LIST_EX @VAL, @VID OUTPUT, @VAT1 OUTPUT, @VAT2
OUTPUT, @VAT3 OUTPUT;
GO
```

3.7.6 Dynamic SQL for a Varying-List SELECT Statement

In DB2, dynamic SQL for varying-list SELECT allows the execution of any SQL statement when you do not know in advance which columns will be retrieved. This type of dynamic SQL uses a SQL descriptor area (SQLDA) to contain information about the SQL statement. This type of dynamic SQL is usually used in applications.

Solution:

In SQL Server there is no solution to emulate SQLDA. Instead, change this statement to another type of dynamic SQL in an application using varying-list SELECT, and then convert.

3.8 Aliases

A DB2 alias can be defined for a table, view, nickname, or another alias. The alias should be created in the same database where the object for which it was created exists.

Links:

DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014

DB2 Example:

```
CREATE ALIAS DB2OBJECTS.A1 FOR DB2OBJECTS.EMPLOYEES;  /* CREATE ALIAS FOR
TABLE */
CREATE ALIAS DB2OBJECTS.A2 FOR DB2OBJECTS.A1;         /* CREATE ALIAS FOR
ALIAS */

CREATE PUBLIC ALIAS A3 FOR DB2OBJECTS.EMPLOYEES;            /* CREATE
PUBLIC ALIAS (IT DOES STORED IN THE SCHEMA SYSPUBLIC) */
```

Solution:

To convert a DB2 alias, you can use a SQL Server synonym. However, SQL Server does not allow the creation of a synonym for another synonym. To convert an alias that has been based on another alias, you should create a synonym for the underlying object instead.

Note: SQL Server does not support public synonyms, so we don't convert them automatically.

SQL Server Example:

```
CREATE SYNONYM DB2OBJECTS.A1 FOR DB2OBJECTS.EMPLOYEES; /* CREATE SYNONYM
FOR TABLE */

CREATE SYNONYM DB2OBJECTS.A2 FOR DB2OBJECTS.EMPLOYEES; /* CREATE SYNONYM
FOR THE SAME TABLE (WE MAKE EMULATION OF SYNONYMS ON EACH OTHER) */
```

3.9 Nicknames

In DB2, a nickname can be used to access objects either from other databases or from non-relational data sources. Both cases are discussed below.

3.9.1 References to Other Databases

Because DB2 doesn't allow the use of remote database names in queries, nicknames are used in DB2. Nicknames can be created for objects from another database or data source that is defined on a local DB2 instance.

DB2 Example:

```
CREATE NICKNAME DEPT FOR SALE.BUH.DEPARTMENT;
```

Solution:

SQL Server allows three-level object names in the format <*database-name*>.<*schema-name*>.<*object-name*>. You can convert DB2 nicknames that have been defined for tables, views, or stored procedures by using SQL Server synonyms. For remote databases, a SQL Server linked server object should be created.

SQL Server Example:

```
CREATE SYNONYM DEPT FOR SALE.BUH.DEPARTMENT;
```

3.9.2 References to Data from a Nonrelational Wrapper

In DB2, the CREATE NICKNAME statement can be used to define the data that is to be accessed through a nonrelational wrapper. This statement has options that permit the creation of nicknames for structured files so that the files can be treated like database tables.

DB2 Example:

```
CREATE NICKNAME DATA1
(DCODE INTEGER,
NAME CHAR(20),
DEPARTMENT CHAR(20))
FOR SERVER DEPTS
OPTIONS
(FILE_PATH '/USR/PAT/DATA1.TXT',
COLUMN_DELIMITER ',',
```

```
KEY_COLUMN 'DCODE',
SORTED 'Y',
VALIDATE_DATA_FILE 'Y');
```

Solution:

In SQL Server, the solution is the same as for DB2, except that you must create a linked server for the nonrelational data source; in this example, the nonrelational data source is a text file. SQL Server makes this possible using the OLE DB provider mechanism.

SQL Server Example:

```
CREATE SYNONYM DEPT FOR LINKED_SERVER.SALE.BUH.DEPARTMENT;
```

Note that this code is not applicable on Azure SQL DB as this version of SQL Server doesn't support working with file system.

3.10 User-Defined Types

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

There are six types of user-defined data type:

- Distinct type
- Structured type
- Reference type
- Array type
- Row type
- Cursor type

Each of these types is described in the following sections.

3.10.1 Distinct Type

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

A DB2 distinct type is a user-defined type that is based on an existing DB2 built-in data type. Internally, a distinct type shares its representation with an existing type (the source type), but is considered to be a separate and incompatible type. Values with a user-defined distinct type can only be compared with values of exactly the same user-defined distinct type. The user-defined distinct type must have been defined using the WITH COMPARISONS clause.

DB2 Example:

```
CREATE DISTINCT TYPE DB2_UDT.DISTINCT_TYPE AS VARCHAR (100) WITH
COMPARISONS
```

Solution:

In SQL Server, user-defined types are the same as DB2 user-defined distinct types. The WITH COMPARISONS clause can be omitted.

SQL Server Example:

```
CREATE TYPE DISTINCT_TYPE FROM VARCHAR(100);
```

3.10.2 Structured Type

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

A DB2 structured type is a user-defined type that contains one or more attributes, each of which has a name and a data type of its own. A structured type can serve as the type of a table or view in which each column of the table derives its name and data type from one of the attributes of the structured type. A structured type can also serve as a type of a column or a type for an argument to a routine.

A structured type also includes a set of method specifications. Methods enable you to define behaviors for structured types. Like user-defined functions (UDFs), methods are routines that extend SQL. In the case of methods, however, the behavior is integrated solely with a particular structured type.

A structured type can be used as the type of a table, view, or column. When used as the type for a table or view, that table or view is known as a typed table or typed view respectively. For typed tables and typed views, the names and data types of the attributes of the structured type become the names and data types of the columns of the typed table or typed view. Rows of the typed table or typed view can be thought of as a representation of instances of the structured type.

A structured type instance can be stored in the database as a row in a table, in which each column of the table is an attribute of the instance of the type, or as a value in a column. To store objects as rows in a table, the table is defined with the structured type, rather than by specifying individual columns in the table definition.

DB2 Example1:

```
CREATE TYPE DB2_UDT.STRUCTURED_TYPE AS (DEPT_NAME VARCHAR(100), MAX_EMPS
INT) MODE DB2SQL;

CREATE TABLE DB2_UDT.STRUCT_TYPE_TABLE OF DB2_UDT.STRUCTURED_TYPE (REF IS
OID USER GENERATED);

INSERT INTO DB2_UDT.STRUCT_TYPE_TABLE (OID, DEPT_NAME, MAX_EMPS) VALUES
(DB2_UDT.STRUCTURED_TYPE (GENERATE_UNIQUE( )), 'SALES', 20);

DB2 EXAMPLE2:

CREATE TYPE DB2_UDT.STRUCT_TYPE_REF_USING AS (DEPT_NAME VARCHAR(100),
MAX_EMPS INT) REF USING INTEGER MODE DB2SQL;

CREATE TABLE DB2_UDT.STRUCT_TYPE_TABLE_REF_USING OF
DB2_UDT.STRUCT_TYPE_REF_USING (REF IS OID USER GENERATED);

INSERT INTO DB2_UDT.STRUCT_TYPE_TABLE_REF_USING (OID, DEPT_NAME, MAX_EMPS)
VALUES (DB2_UDT.STRUCT_TYPE_REF_USING (1), 'SALES', 20);
```

Solution:

In SQL Server, a table type is a user-defined type that can partially emulate a DB2 typed table. However, SQL Server does not allow using table types as the types for columns in a table. In SQL Server a table type can be used to emulate a DB2 structured type only when the structured type is used as a local variable, or when a typed table is based on the structured type whose attributes are DB2 base types.

If, in DB2, a definition of a structured type on which a DB2 table is based has the REF USING rep-type expression, then in SQL Server you should add a primary key or unique constraint that contains the object identifier (OID) attribute of rep-type type to the SQL Server table type definition when converting a table of this type. In this situation, you should perform the conversion of DB2 base types to SQL Server by using type mapping.

If there is no REF USING expression in the definition of the DB2 structured type, then the OID attribute should be defined as varbinary(16) in SQL Server. You do not need to add an OID field and a key in a table type in SQL Server when you are converting a structured type that is used as a local variable.

There is no SQL Server solution yet to emulate DB2 typed views. To emulate reference types and structured type method hierarchies, you should use SQL Server CLR user-defined types.

SQL Server Example1:

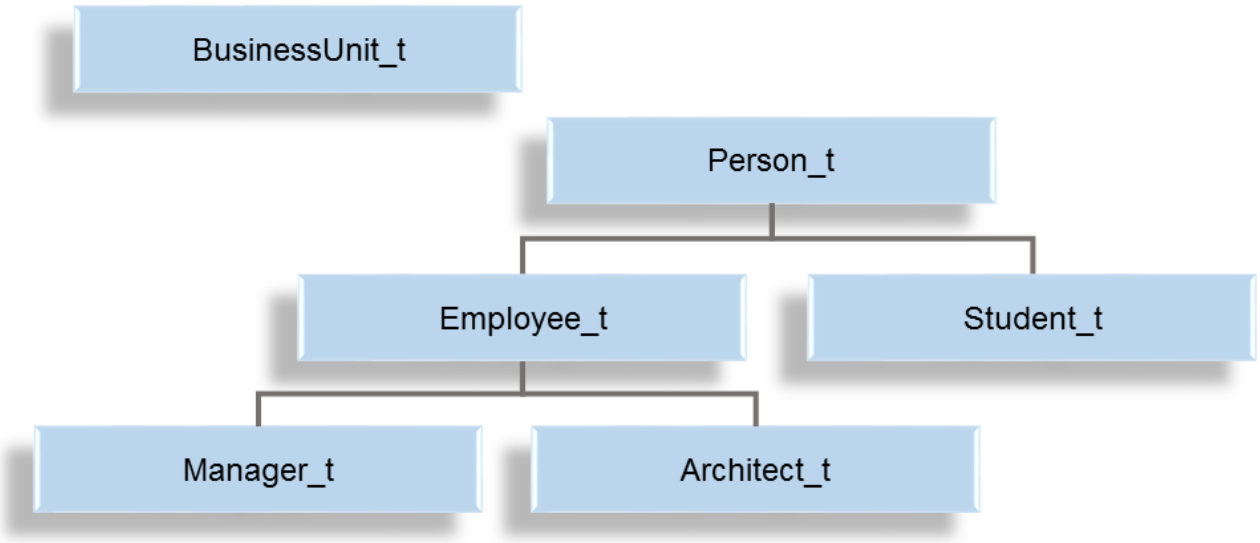
```
CREATE TYPE STRUCTURED_TYPE AS TABLE
( DEPT_NAME VARCHAR(20),
 MAX_EMPS INTEGER);

CREATE TABLE STRUCT_TYPE_TABLE
( OID INTEGER PRIMARY KEY,
 DEPT_NAME VARCHAR(20),
 MAX_EMPS INTEGER);
```

3.10.2.1 Structured Type Hierarchy

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

A structured type may be created under another structured type, in which case the newly created type is a subtype of the original structured type. The original type is the supertype. The subtype inherits all the attributes of the supertype, and can optionally have additional attributes of its own.
For example, a data model may need to represent a special type of employee called a manager. Managers have more attributes than employees who are not managers. The Manager_t type inherits the attributes defined for an employee, but also is defined with some additional attributes of its own, such as a special bonus attribute that is only available to managers. The type hierarchies are shown in following figure.

Type hierarchies



DB2 Example:

```
CREATE TYPE DB2_UDT.HIERACHY_BUSINESSUNIT_T AS (Name VARCHAR(20), Headcount INT)
MODE DB2SQL;

--To create the Person_t type hierarchy, issue the following SQL statements:
CREATE TYPE DB2_UDT.Hierachy_Address_t AS (street VARCHAR(30),number
CHAR(15),city VARCHAR(30), state VARCHAR(20), zip CHAR(10)) MODE DB2SQL;

CREATE TYPE DB2_UDT.Hierachy_Person_t AS (Name VARCHAR(20), Age INT, Address
DB2_UDT.Hierachy_Address_t) REF USING VARCHAR(13) FOR BIT DATA MODE DB2SQL;

CREATE TYPE DB2_UDT.Hierachy_Employee_t UNDER DB2_UDT.Hierachy_Person_t AS
(SerialNum INT, Salary DECIMAL(9,2), Dept REF (DB2_UDT.HIERACHY_BUSINESSUNIT_T))
MODE DB2SQL;
```

```
CREATE TYPE DB2_UDT.Hierachy_Student_t UNDER DB2_UDT.Hierachy_Person_t AS
(SerialNum CHAR(6), GPA DOUBLE) MODE DB2SQL;

CREATE TYPE DB2_UDT.Hierachy_Manager_t UNDER DB2_UDT.Hierachy_Employee_t AS
(Bonus DECIMAL(7,2)) MODE DB2SQL;

CREATE TYPE DB2_UDT.Hierachy_Architect_t UNDER DB2_UDT.Hierachy_Employee_t AS
(StockOption INTEGER) MODE DB2SQL;
```

3.10.2.2 Structured type includes a set of method specifications.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

The CREATE METHOD statement is used to associate a method body with a method specification that is already part of the definition of a user-defined structured type. The method specification must be previously defined using the CREATE TYPE or ALTER TYPE statement before CREATE METHOD.

DB2 Example:

```
CREATE TYPE DB2_UDT.TYPE_WITH_METHODS AS (PRINCIPLE INT, INTEREST
DECIMAL(5,2), YEAR INT)
 NOT FINAL
 MODE DB2SQL
 METHOD SI()
 RETURNS FLOAT
 LANGUAGE SQL,

 METHOD CI()
 RETURNS FLOAT
 LANGUAGE SQL,

 METHOD PROD(NUM INT)
 RETURNS INTEGER
 LANGUAGE SQL

CREATE METHOD SI( ) FOR DB2_UDT.TYPE_WITH_METHODS RETURN
(SELF..PRINCIPLE*SELF..INTEREST*SELF..YEAR)/100
CREATE METHOD CI() FOR DB2_UDT.TYPE_WITH_METHODS RETURN SELF..PRINCIPLE *
POWER((1 + SELF..INTEREST/100), SELF..YEAR)
CREATE METHOD PROD(NUM INT) FOR DB2_UDT.TYPE_WITH_METHODS RETURN
NUM+SELF..YEAR
```

DB2 Example:

```
CREATE TYPE DB2_UDT.METHOD_ADDRESS_T AS (STREET VARCHAR(30),NUMBER
CHAR(15),CITY VARCHAR(30), STATE VARCHAR(20), ZIP CHAR(10)) MODE DB2SQL;
CREATE TYPE DB2_UDT.TYPE_WITH_METH_C_LANG AS (STREET VARCHAR(30), NUMBER
CHAR(15), CITY VARCHAR(30), STATE VARCHAR(10))
 NOT FINAL
 MODE DB2SQL
 METHOD SAMEZIP_2 (ADDR DB2_UDT.METHOD_ADDRESS_T)
 RETURNS INTEGER
       LANGUAGE SQL,

 METHOD DISTANCE_2 (ADDR DB2_UDT.METHOD_ADDRESS_T)
 RETURNS FLOAT
 LANGUAGE C
```

```
      DETERMINISTIC
PARAMETER STYLE SQL
NO SQL
NO EXTERNAL ACTION
```

3.10.2.3 Reference type

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

A *reference type* is a companion type to a structured type. Similar to a distinct type, a reference type is a scalar type that shares a common representation with one of the built-in data types. This same representation is shared for all types in the type hierarchy. The reference type representation is defined when the root type of a type hierarchy is created. When using a reference type, a structured type is specified as a parameter of the type. This parameter is called the *target type* of the reference.

The target of a reference is always a row in a typed table or a typed view. When a reference type is used, it may have a *scope* defined. The scope identifies a table (called the *target table*) or view (called the *target view*) that contains the target row of a reference value. The target table or view must have the same type as the target type of the reference type. An instance of a scoped reference type uniquely identifies a row in a typed table or typed view, called the *target row*.

DB2 Example:

```
CREATE TYPE DB2_UDT.REF_TYPE AS (NAME VARCHAR(30), LOCATION VARCHAR(30))
MODE DB2SQL;
CREATE TABLE DB2_UDT.REF_TYPE_TAB OF DB2_UDT.REF_TYPE (REF IS SUPPNO USER
GENERATED);
INSERT INTO DB2_UDT.REF_TYPE_TAB VALUES
(DB2_UDT.REF_TYPE('1'),'DBDEST','KHARKIV');
INSERT INTO DB2_UDT.REF_TYPE_TAB VALUES
(DB2_UDT.REF_TYPE('2'),'NOVOSVIT','KHARKIV');


CREATE TYPE DB2_UDT.REF_TYPE_2 AS (DESCRIPT VARCHAR(20), SUPPLIED_BY
REF(DB2_UDT.REF_TYPE)) MODE DB2SQL;
CREATE TABLE DB2_UDT.REF_TYPE_TAB_2 OF DB2_UDT.REF_TYPE_2 (REF IS PARTNO
USER GENERATED, SUPPLIED_BY WITH OPTIONS SCOPE DB2_UDT.REF_TYPE_TAB);
INSERT INTO DB2_UDT.REF_TYPE_TAB_2 VALUES
(DB2_UDT.REF_TYPE_2('2'),'PROGRAMMS',DB2_UDT.REF_TYPE('1'));
INSERT INTO DB2_UDT.REF_TYPE_TAB_2 VALUES
(DB2_UDT.REF_TYPE_2('3'),'APPLICATION',DB2_UDT.REF_TYPE('2'));

SELECT PARTNO, SUPPLIED_BY->NAME FROM DB2_UDT.REF_TYPE_TAB_2
```

3.10.3 3.10.3 SQL PL data types

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

3.10.3.1 Array type

An array type is a user-defined data type consisting of an ordered set of elements of a single data type.

A user-defined array type is a data type that is defined as an array with elements of another data type. Every ordinary array type has an index with the data type of INTEGER and has a defined maximum cardinality. Every associative array has an index with the data type of INTEGER or VARCHAR and does not have a defined maximum cardinality.

Array type usage: An array type can only be used as the data type of:

– A local variable in a compound SQL (compiled) statement

– A parameter of an SQL routine
– A parameter of a Java™ procedure (non-nested ordinary arrays only)
– The returns type of an SQL function
– A global variable

A variable or parameter defined with an array type can only be used in compound SQL (compiled) statements

An ordinary array type has a defined upper bound on the number of elements and uses the ordinal position as the array index.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE TYPE DB2_UDT.SIMPLE_ARRAY AS INTEGER ARRAY[]

CREATE PROCEDURE DB2_UDT.SIMPLE_ARRAY_TYPE ()
 LANGUAGE SQL
 READS SQL DATA
BEGIN
DECLARE MYSIMPLEA DB2_UDT.SIMPLE_ARRAY;
 SET MYSIMPLEA[100] = 123;
END;
```

An associative array type has no specific upper bound on the number of elements and each element has an associated index value. The data type of the index value can be an integer or a character string but is the same data type for the entire array.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
CREATE TYPE DB2_UDT.INT_ARRAY AS INTEGER ARRAY[100];

CREATE TYPE DB2_UDT.STRING_ARRAY AS VARCHAR (100) ARRAY [100];

CREATE PROCEDURE DB2_UDT.ARRAY_TYPE ()
 LANGUAGE SQL READS SQL DATA
BEGIN
DECLARE INT_ARR DB2_UDT.INT_ARRAY;
DECLARE STR_ARR DB2_UDT.STRING_ARRAY;
SET INT_ARR = ARRAY[1,2,3];
SET STR_ARR = ARRAY['BOB', 'ANN', 'SUE'];
END;
```

3.10.3.2 Row type

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

A row type is a data type that is defined as an ordered sequence of named fields, each with an associated data type, which effectively represents a row. A row type can be used as the data type for variables and parameters in SQL PL to provide simple manipulation of a row of data.

Row type usage: A row type can only be used as the data type of:

- A local variable in a compound SQL (compiled) statement
- A parameter of an SQL routine
- The return type of an SQL function
- The element of an array type

- The field type in a row type
- A user-defined cursor type (only non-nested row types)
- A global variable

A variable or parameter defined with a row type can only be used in compound SQL (compiled) statements

DB2 Example:

```
CREATE TYPE DB2_UDT.ROW_TYPE AS ROW (DEPTNO VARCHAR(3), DEPTNAME
VARCHAR(29), MGRNO CHAR(6));

CREATE TABLE DB2_UDT.ROW_TYPE_TAB (DEPTNO VARCHAR(3), DEPTNAME
VARCHAR(29), MGRNO CHAR(6))

CREATE PROCEDURE DB2_UDT.ROW_TYPE_PROC ()
 LANGUAGE SQL
 MODIFIES SQL DATA
BEGIN
DECLARE R1 DB2_UDT.ROW_TYPE;
SET R1.DEPTNO = '1';
SET R1.DEPTNAME = 'SALES';
SET R1.MGRNO = '111111';
INSERT INTO DB2_UDT.ROW_TYPE_TAB VALUES (R1.DEPTNO, R1.DEPTNAME,
R1.MGRNO);
END;

CALL DB2_UDT.ROW_TYPE_PROC;
```

3.10.3.3 Anchored data type

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

An anchored type defines a data type based on another SQL object such as a column, global variable, SQL variable, SQL parameter, or the row of a table or view.

A data type defined using an anchored type definition maintains a dependency on the object to which it is anchored. Any change in the data type of the anchor object will impact the anchored data type. If anchored to the row of a table or view, the anchored data type is ROW with the fields defined by the columns of the anchor table or anchor view.

This data type is useful when declaring variables in cases where you require that the variable have the same data type as another object, for example a column in a table, but you do not know exactly what is the data type.

An anchored data type can be of the same type as one of:

– a row in a table
– a row in a view
– a cursor variable row definition
– a column in a table
– a column in a view
– a local variable, including a local cursor variable or row variable
– a global variable

Anchored data types can only be specified when declaring or creating one of the following:

– a local variable in an SQL procedure, including a row variable
– a local variable in a compiled SQL function, including a row variable
– a routine parameter
– a user-defined cursor data type using the CREATE TYPE statement.
  It cannot be referenced in a DECLARE CURSOR statement.
– a function return data type
– a global variable

DB2 Example:

```
CREATE PROCEDURE DB2_UDT.ANCHOR_TYPE_VAL()
BEGIN
 DECLARE V1 ANCHOR DATA TYPE TO DB2_UDT.ANCHOR_ROW_TYPE.DEPTNAME;
 SELECT DEPTNAME INTO V1 FROM DB2_UDT.DEPT_ROW_TYPE;
 INSERT INTO DB2_UDT.ANCHOR_ROW_TYPE VALUES (5, V1, 555);
END;

CALL DB2_UDT.ANCHOR_TYPE_VAL();
```

DB2 Example:

```
CREATE TYPE DB2_UDT.ANCHOR_TYPE AS ROW ANCHOR DATA TYPE TO ROW OF
DB2_UDT.DEPT_ROW_TYPE;

CREATE TABLE DB2_UDT.ANCHOR_ROW_TYPE LIKE DB2_UDT.DEPT_ROW_TYPE;

CREATE PROCEDURE DB2_UDT.ANCHOR_TYPE()
BEGIN
 DECLARE ANCHOR_VAR DB2_UDT.ANCHOR_TYPE;
 SELECT * INTO ANCHOR_VAR FROM DB2_UDT.DEPT_ROW_TYPE;
 INSERT INTO DB2_UDT.ANCHOR_ROW_TYPE VALUES (ANCHOR_VAR.DEPTNO, 'MANAGER',
ANCHOR_VAR.MGRNO);
END;

CALL DB2_UDT.ANCHOR_TYPE();
```

3.10.3.4 Cursor data type

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

A user-defined cursor type is a user-defined data type defined with the keyword CURSOR and optionally with an associated row type. A user-defined cursor type with an associated row type is astrongly-typed cursor type; otherwise, it is a weakly-typed cursor type. A value of a user-defined cursor type represents a reference to an underlying cursor.

Cursor type usage: A cursor type can only be used as the data type of:

– A local variable in a compound SQL (compiled) statement
– A parameter of an SQL routine
– The returns type of an SQL function
– A global variable
– A variable or parameter defined with a cursor type can only be used in compound SQL (compiled) statements
– A variable or parameter that has a strongly-typed cursor type must not be used to assign cursor values that are based on a statement-name instead of a select-statement

A user-defined cursor type with an associated row type is a strongly-typed cursor type; otherwise, it is a weakly-typed cursor type.

DB2 Example:

```
CREATE TYPE DB2_UDT.ANCHOR_TYPE AS ROW ANCHOR DATA TYPE TO ROW OF
DB2_UDT.DEPT_ROW_TYPE;

CREATE TYPE DB2_UDT.CURSOR_TYPE AS DB2_UDT.ANCHOR_TYPE CURSOR;

CREATE PROCEDURE DB2_UDT.CURSOR_TYPE (IN NOM VARCHAR(8))
LANGUAGE SQL
```

```
BEGIN
 DECLARE C1 DB2_UDT.CURSOR_TYPE;
 SET C1 = CURSOR FOR SELECT * FROM DB2_UDT.DEPT_ROW_TYPE WHERE DEPTNO =
NOM;
END;

CALL DB2_UDT.CURSOR_TYPE(1);
```

3.11 Special Registers

This section describes emulation of some common special registers from DB2 to SQL Server.

3.11.1 CURRENT TIMESTAMP

In DB2, the special register CURRENT TIMESTAMP (or CURRENT_TIMESTAMP) specifies a timestamp that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
SELECT CURRENT TIMESTAMP FROM SYSIBM.SYSDUMMY1
```

Solution:

In SQL Server, you can emulate the CURRENT TIMESTAMP register by using the CURRENT_TIMESTAMP or GETDATE() functions.

SQL Server Example:

```
SELECT CURRENT_TIMESTAMP;
SELECT GETDATE();
```

3.11.2 CURRENT TIMESTAMP WITH TIME ZONE, SYSTIMESTAMP

If you want a timestamp with a time zone, the special register can be referenced as CURRENT TIMESTAMP WITH TIME ZONE.

Solution:

In SQL Server, use SYSDATETIMEOFFSET function.

SQL Server Example:

```
SYSDATETIMEOFFSET()
```

3.11.3 CURRENT DATE

In DB2, the special register CURRENT DATE (or CURRENT_DATE) specifies a date that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
SELECT CURRENT DATE FROM SYSIBM.SYSDUMMY1
```

Solution:

In SQL Server, you can emulate CURRENT DATE using the CURRENT_TIMESTAMP function.

SQL Server Example:

```
SELECT CAST(CURRENT_TIMESTAMP AS DATE);

SELECT DATEFROMPARTS (DATEPART(YEAR, CURRENT_TIMESTAMP), DATEPART(MONTH,
CURRENT_TIMESTAMP), DATEPART(DAY, CURRENT_TIMESTAMP));

SELECT DATETIMEFROMPARTS (DATEPART(YEAR, CURRENT_TIMESTAMP),
DATEPART(MONTH, CURRENT_TIMESTAMP), DATEPART(DAY, CURRENT_TIMESTAMP), 0,
0, 0, 0);
```

3.11.4 CURRENT TIME

In DB2, the special register CURRENT TIME (or CURRENT_TIME) specifies a time that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
SELECT CURRENT TIME FROM SYSIBM.SYSDUMMY1
```

Solution:

In SQL Server, you can emulate CURRENT TIME by using the CURRENT_TIMESTAMP function.

SQL Server Example:

```
SELECT CAST(CURRENT_TIMESTAMP AS TIME);

SELECT TIMEFROMPARTS (DATEPART(HOUR, CURRENT_TIMESTAMP), DATEPART(MINUTE,
CURRENT_TIMESTAMP), DATEPART(SECOND, CURRENT_TIMESTAMP) , 0, 0);
```

3.11.5 CURRENT TIMEZONE, CURRENT TIMEZONE, CURRENT_TIMEZONE

In DB2, the special register CURRENT TIMEZONE (or CURRENT_TIMEZONE) specifies the difference between UTC (Coordinated Universal Time, formerly known as GMT) and local time at the application server.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
SELECT CURRENT TIMEZONE FROM SYSIBM.SYSDUMMY1
```

Solution:

In SQL Server, you can emulate CURRENT TIMEZONE by using the DATEPART and SYSDATETIMEOFFSET functions.

SQL Server Example:

```
WITH TZ (O) AS (SELECT DATEPART (TZOFFSET, SYSDATETIMEOFFSET ()))
SELECT (O / 60 * 100 + O % 60) * 100 FROM TZ;
```

3.11.6 CURRENT USER

In DB2, the special register CURRENT USER (or CURRENT_USER) contains the authorization ID that is to be used for statement authorization.

Links: DB2 for Linux UNIX and Windows 10.5.0, SQL Server 2014.

DB2 Example:

```
SELECT CURRENT USER FROM SYSIBM.SYSDUMMY1
```

Solution:

In SQL Server, you can emulate the CURRENT USER register by using the CURRENT_USER function.

SQL Server Example:

```
SELECT CURRENT_USER
```

3.11.7 SESSION_USER and USER

In DB2, the SESSION_USER special register specifies the authorization ID that is to be used for the current session. USER is a synonym for the SESSION_USER special register. SESSION_USER is the preferred spelling. Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
SELECT SESSION_USER FROM SYSIBM.SYSDUMMY1
```

Solution:

In SQL Server, you can emulate SESSION_USER and USER by using the SESSION_USER function.

SQL Server Example:

```
SELECT SESSION_USER
```

3.11.8 SYSTEM_USER

In DB2, the SYSTEM_USER special register specifies the authorization ID of the user who is connected to the database.

Links: DB2 for Linux UNIX and Windows 10.5.0, SQL Server 2014.

DB2 Example:

```
SELECT SYSTEM_USER FROM SYSIBM.SYSDUMMY1
```

Solution:

In SQL Server, you can emulate SYSTEM_USER by using the SYSTEM_USER function.

SQL Server Example:

```
SELECT SYSTEM_USER
```

3.11.9 CURRENT CLIENT_APPLNAME

In DB2, the CURRENT CLIENT_APPLNAME (or CLIENT APPLNAME) special register contains the value of the application name, derived from the client information specified for this connection.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
SELECT CURRENT CLIENT_APPLNAME FROM SYSIBM.SYSDUMMY1
```

Solution:

In SQL Server, you can emulate the CURRENT CLIENT_APPLNAME register by using the APP_NAME() function.

SQL Server Example:

```
SELECT APP_NAME()
```

3.11.10 CURRENT CLIENT_WRKSTNNAME

In DB2, the CURRENT CLIENT_WRKSTNNAME (or CLIENT WRKSTNNAME) special register contains the value of the workstation name, derived from the client information specified for this connection.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
SELECT CURRENT CLIENT_WRKSTNNAME FROM SYSIBM.SYSDUMMY1
```

Solution:

In SQL Server, you can emulate CURRENT CLIENT_WRKSTNNAME by using the HOST_NAME() function.

SQL Server Example:

```
SELECT HOST_NAME()
```

3.11.11 CURRENT LOCK TIMEOUT

In DB2, the CURRENT LOCK TIMEOUT special register specifies the number of seconds to wait for a lock, before returning an error that indicates that a lock cannot be obtained.

Links: DB2 for Linux UNIX and Windows 10.5.0, SQL Server 2014.

DB2 Example:

```
SET CURRENT LOCK TIMEOUT 1800;
SELECT CURRENT LOCK TIMEOUT FROM SYSIBM.SYSDUMMY1;
```

Solution:

In SQL Server, you can emulate CURRENT LOCK TIMEOUT by using the @@LOCK_TIMEOUT function.

SQL Server Example:

```
SET LOCK_TIMEOUT 1800;
SELECT @@LOCK_TIMEOUT;
```

3.11.12 CURRENT SCHEMA, CURRENT_SCHEMA

In DB2, the CURRENT SCHEMA (or CURRENT_SCHEMA) special register specifies a VARCHAR(128) value that identifies the schema name used to qualify database object references, where applicable, in dynamically prepared SQL statements.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
SELECT CURRENT SCHEMA FROM SYSIBM.SYSDUMMY1
```

Solution:

In SQL Server, you can emulate CURRENT SCHEMA by using the db_name() or schema_name() functions. It depends from mapping type (Schema - Database or Schema - Schema).

SQL Server Example:

```
SELECT DB_NAME();
SELECT SCHEMA_NAME();
```

3.11.13 CURRENT SERVER, CURRENT_SERVER

In DB2, the CURRENT SERVER (or CURRENT_SERVER) special register specifies a VARCHAR(18) value that identifies the current application server. The register contains the actual name of the application server, not an alias.

Links: DB2 for Linux UNIX and Windows 10.5.0, DB2 for z/OS 11.0.0, SQL Server 2014.

DB2 Example:

```
SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1
```

Solution:

In SQL Server, you can emulate CURRENT SERVER by using the @@SERVERNAME function.

SQL Server Example:

```
SELECT @@SERVERNAME
```

3.11.14 CURRENT ISOLATION

In DB2, the CURRENT ISOLATION special register holds a CHAR(2) value that identifies the isolation level (in relation to other concurrent sessions) for any dynamic SQL statements issued within the current session.

Links: DB2 for Linux UNIX and Windows 10.5.0, SQL Server 2014.

DB2 Example:

```
SELECT CURRENT ISOLATION FROM SYSIBM.SYSDUMMY1
```

Solution:

In SQL Server, you can emulate CURRENT ISOLATION by using the script in the following example. But this example returns isolation levels used in SQL Server. For more information about how to match a DB2 isolation level to a SQL Server isolation level, see section “Isolation Level and Lock Type”.

SQL Server Example:

```
SELECT
 CASE S.TRANSACTION_ISOLATION_LEVEL
 WHEN 4 THEN 'RR'
 WHEN 3 THEN 'RS'
 WHEN 2 THEN 'CS'
 WHEN 1 THEN 'UR'
 ELSE ' '
 END
FROM SYS.DM_EXEC_SESSIONS S
WHERE S.SESSION_ID = @@SPID
```

3.12 Synonyms

In DB2, a synonym is an alternate name for a table or view. A synonym can be used to reference a table or view in cases where an existing table or view can be referenced. However, synonyms are deprecated, therefore we convert them like the aliases. See Aliases for more details.

DB2 Example:

```
CREATE SYNONYM DB2_OBJECTS.A1 FOR DB2_OBJECTS.EMPLOYEES
```

SQL Server Example:

```
CREATE SYNONYM DB2_OBJECTS.A1 FOR DB2_OBJECTS.EMPLOYEES
```

4.0 Migrating DB2 Standard Functions

This section describes how to map DB2 standard functions to equivalent SQL Server functions, and provides solutions for emulating DB2 functions.

4.1 Equivalent Functions

The following DB2 system functions are usable as they stand, in SQL Server code:

ABS, ACOS, ASCII, ASIN, ATAN, ATN2, AVE, CEILING (CEIL), COALESCE (VALUE, NVL), COUNT, COUNT_BIG, COS, COT, CURRENT_TIMESTAMP, CURRENT_USER, DAY, DEGREES, DIFFERENCE, EXP, FLOOR, LEFT, LENGTH, LOWER, LN, LOCATE, LOG10, LTRIM, MAX, MIN, MONTH, NULLIF, POWER, RADIANS, RAND, REPLACE, RIGHT, ROUND, RTRIM, SESSION_USER, SUM, SYSTEM_USER, SIGN, SIN, SINH, SOUNDEX, SQRT, TAN, UNICODE, UPPER, YEAR.

4.2 Emulated Functions

The following DB2 system functions can be emulated by using various SQL Server functions or Transact-SQL constructions.

4.2.1 Functions with a Variable Parameter Number

In DB2, the following functions have a variable parameter number:

```
GREATEST(VALUE1, VALUE2, …)
LEAST(VALUE1, VALUE2, …)
```

Solution:

In SQL Server, you can emulate functions that have a variable parameter number by using the Transact-SQL IIF function.

4.2.2 String Functions

4.2.2.1 ASCII_CHR

In DB2, the ASCII_CHR function returns the character that has the ASCII code value that is specified by the argument.

DB2 Example:

```
ASCII_CHR(65)
```

Solution:

In SQL Server, use CHAR function.

SQL Server Example:

```
CHAR(65)
```

4.2.2.2 CHARACTER_LENGTH, CHAR_LENGTH, LENGTH, LENGTHB, LENGTH2, LENGTH4

In DB2, returns the length of an expression in the specified string-unit.

DB2 Example:

```
CHARACTER_LENGTH('JOHN SMIT ', OCTETS)
```

Solution:

In SQL Server, use LEN function.

The OCTETS as a CODEUNITS is compatible with LEN() only.

Replace trilling spaces or use other way to other character to correct length evaluate.

SQL Server Example:

```
LEN(REPLACE('JOHN SMIT ', ' ', '.'))
LEN('JOHN SMIT ' + '.') - 1
```

4.2.2.3 CHR

In DB2, returns the character that has the ASCII code value specified by the argument.

DB2 Example:

```
CHR(65)
```

Solution:

In SQL Server, use CHAR function.

SQL Server Example:

```
CHAR(65)
```

4.2.2.4 CONCAT

In DB2, returns the concatenation of two strings.

DB2 Example:

```
CONCAT('NUMBER ', '1')
```

Solution:

In SQL Server, use + (plus) operator.

SQL Server Example:

```
'NUMBER ' + '1'
```

4.2.2.5 CONTAINS

In DB2, the CONTAINS function searches a text search index using criteria that are specified in a search argument and returns a result of bit type about whether or not a match was found.

DB2 Example:

```
CONTAINS (PEOPLE.FIRST_NAME, 'THOM%')
```

Solution:

In SQL Server, use CONTAINS function. It is returning logical true or false, so you are need to wrap function to get bit result.

SQL Server Example:

```
IIF(CONTAINS(PEOPLE.FIRST_NAME, 'THOM%'), 1, 0)
```

4.2.2.6 DECRYPT_BIN, DECRYPT_BINARY, DECRYPT_BIT, DECRYPT_CHAR, DECRYPT_DB

In DB2, returns a value that is the result of decrypting encrypted data using a password string.

DB2 Example:

```
DECRYPT_BIN(@STR, @PASS)
DECRYPT_CHAR(@STR, @PASS)
```

Solution:

In SQL Server, use DECRYPTBYPASSPHRASE function which decrypts data that was encrypted with a passphrase.

SQL Server Example:

```
DECRYPTBYPASSPHRASE(@PASS, @STR)
```

4.2.2.7 ENCRYPT, ENCRYPT_TDES

In DB2, returns a value that is the result of encrypting a data string expression.

The ENCRYPT_TDES function uses the Triple DES encryption algorithm.

DB2 Example:

```
ENCRYPT(@STR)
ENCRYPT(@STR, @PASS)
ENCRYPT(@STR, @PASS, @HINT)
ENCRYPT_TDES(@STR)
ENCRYPT_TDES(@STR, @PASS)
ENCRYPT_TDES(@STR, @PASS, @HINT)
```

Solution:

In SQL Server, use ENCRYPTBYPASSPHRASE function, which encrypt data with a passphrase using the TRIPLE DES algorithm with a 128 key bit length.

Hint is ignored.

SQL Server Example:

```
ENCRYPTBYPASSPHRASE (@PASS, @STR)
```

4.2.2.8 GENERATE_UNIQUE

In DB2, returns a bit data character string that is unique compared to any other execution of the same function. The result of this function can be used to provide unique values in a table. Each successive value will be greater than the previous value, providing a sequence that can be used within a table. The sequence is based on the time when the function was executed.

DB2 Example:

```
GENERATE_UNIQUE()
```

Solution:

In SQL Server, use GETDATE function and SEQUENCE to generate unique value in necessary format.

The limitation is method does not work in function context because of SEQUENCE.

SQL Server Example:

```
CONVERT(VARBINARY(13), '0X' +
LEFT(REPLACE(REPLACE(REPLACE(REPLACE(CONVERT(VARCHAR(18), GETDATE(), 21),
' ', ''), '-', ''), '.', ''), ':', '') + '0000000000000000000', 18) +
LEFT(CAST(NEXT VALUE FOR SEQUENCE_GEN_UNIQUE AS VARCHAR(8)) + '00000000',
8), 1)
```

Note: Azure SQL DB doesn't support SEQUENCE objects.

4.2.2.9 INITCAP

In DB2, returns a string with the first character of each word converted to uppercase and the rest to lowercase.

DB2 Example:

```
INITCAP('JOHN SMIT')
```

Solution:

In SQL Server, create such function:

```
CREATE FUNCTION DBO.INITCAP(@S NVARCHAR(MAX)) RETURNS NVARCHAR(MAX)
BEGIN
    DECLARE
        @RESULT NVARCHAR(MAX) = '',
        @MODE INT = 0,
        @I INT = 1,
        @CUR INT,
        @CHAR NVARCHAR(1),
        @LEN INT = LEN(@S),
    WHILE @I <= @LEN
    BEGIN
        SET @CHAR = SUBSTRING(@S, @I, 1);
        SET @CUR = ISNUMERIC(@CHAR) +
            IIF(UNICODE(UPPER(@CHAR)) != UNICODE(LOWER(@CHAR)), 1, 0);
        SET @RESULT = @RESULT +
            IIF(@MODE != @CUR, UPPER(@CHAR), LOWER(@CHAR));
        SET @MODE = @CUR;
        SET @I = @I + 1;
```

```
    END
    RETURN @RESULT;
END
```

SQL Server Example:

```
DBO.INITCAP('JOHN SMIT')
```

4.2.2.10 INSERT, OVERLAY

In DB2, returns a string, where LENGTH bytes are deleted from SOURCE beginning at START, and INSERT is inserted into SOURCE beginning at START.

DB2 Example:

```
INSERT(@SOURCE_STR, @START, @LENGTH, @INSERT_STR)
OVERLAY(@SOURCE_STR, @INSERT_STR, @START, @LENGTH)
OVERLAY(@SOURCE_STR PLACING @INSERT_STR FROM @START FOR @LENGTH)
```

Solution:

In SQL Server, use STUFF function.

SQL Server Example:

```
STUFF($SOURCE_STR, $START, $LENGTH, $ INSERT_STR)
```

4.2.2.11 INSTR, INSTRB, INSTR2, INSTR4

In DB2, returns the starting position of a string within another string.

DB2 Example:

```
INSTR(@STR, @FRAGMENT, @START, @OCCURED)
INSTRB(@STR, @FRAGMENT, @START)
INSTR2(@STR, @FRAGMENT)
```

Solution:

In SQL Server, use such recursive solution:

```
CREATE FUNCTION DBO.INSTR(@S NVARCHAR(MAX), @FRAGMENT NVARCHAR(MAX),
    @START BIGINT, @OCCURED INT = NULL, @CODEUNITS VARCHAR(20) = NULL)
RETURNS BIGINT AS
BEGIN
    IF @ OCCURED IS NULL OR @ OCCURED < 1 SET @OCCURED = 1;
    IF @START < 1 SET @START = 1;

    DECLARE @POS BIGINT = CHARINDEX(@FRG, @S);

    IF @ OCCURED = 1 RETURN @POS;

RETURN DBO.INSTR(SUBSTRING(@S, @POS + 1, LEN(@S + '.') - 1), @FRAGMENT,
    1, @OCR - 1, @CODEUNITS) + @POS;
END
```

SQL Server Example:

```
DBO.INSTR(@STR, @FRAGMENT, @START, @OCCURED)
DBO.INSTR(@STR, @FRAGMENT, @START, DEFAULT)
DBO.INSTR(@STR, @FRAGMENT, DEFAULT, DEFAULT)
```

4.2.2.12 LCASE

In DB2, returns a string in which all the SBCS characters have been converted to lowercase characters.

DB2 Example:

```
LCASE(@S)
```

Solution:

In SQL Server, use LOWER function.

SQL Server Example:

```
LOWER(@S)
```

4.2.2.13 LOCATE, LOCATE_IN_STRING, POSITION, POSSTR

In DB2, returns the starting position of one string within another string.

DB2 Example:

```
LOCATE(@FRAGMENT, @STR, @START)
POSITION(@FRAGMENT, @STR)
```

Solution:

In SQL Server, use CHARINDEXfunction.

SQL Server Example:

```
CHARINDEX (@FRAGMENT, @STR, @START)
CHARINDEX (@FRAGMENT, @STR)
```

4.2.2.14 LPAD

In DB2, returns a string that is padded on the left with the specified character, or with blanks.

DB2 Example:

```
LPAD('123',  10, '0')
```

Solution:

In SQL Server, use RIGHT function.

SQL Server Example:

```
RIGHT('0000000000' + '123', 10)
```

4.2.2.15 OCTET_LENGTH

In DB2, returns the length of an expression in octets (bytes).

DB2 Example:

```
OCTET_LENGTH(@STR)
```

Solution:

In SQL Server, use function DATALENGTH which returns the number of bytes used to represent any expression.

SQL Server Example:

```
DATALENGTH(@STR)
```

4.2.2.16 REPEAT

In DB2, returns a character string composed of argument#1 repeated argument#2 times.

VARBINARY is supported too.

DB2 Example:

```
REPEAT(@STR, @COUNT)
```

Solution:

In SQL Server, use REPLICATE function.

SQL Server Example:

```
REPLICATE(@STR, @COUNT)
```

4.2.2.17 RPAD

In DB2, returns a string that is padded on the right with the specified character, string, or with blanks.

DB2 Example:

```
RPAD('F', 10, 'O')
```

Solution:

In SQL Server, use LEFT function.

SQL Server Example:

```
LEFT('F' + 'OOOOOOOOOO', 10)
```

4.2.2.18 SPACE

In DB2, returns a character string that consists of a specified number of blanks.

DB2 Example:

```
SPACE(N)
```

Solution:

In SQL Server, use REPLICATE function.

SQL Server Example:

```
REPLICATE (' ', N)
```

4.2.2.19 STRIP, TRIM

mode = { both | b | leading | l | trailing | t }

Removes leading or trailing blanks or other specified leading and/or trailing characters from a string expression.

DB2 Example:

```
STRIP(:S)
STRIP(:S, BOTH)
STRIP(:STR, LEADING, '0')
TRIM(:S)
TRIM(:S, BOTH)
TRIM(:STR, LEADING, '0')
```

Solution:

In SQL Server, create special function or use inline T-SQL code with cycles.

SQL Server Example:

```
WHILE @S LIKE '0%' SET @S = RIGHT(@S, LEN(@S) - 1);
WHILE @S LIKE '%0' SET @S = LEFT(@S, LEN(@S) - 1);
```

4.2.2.20 SUBSTR, SUBSTRB, SUBSTRING

In DB2, returns a substring of a string.

DB2 Example:

```
SUBSTR (:STR, :START, :LEN)
SUBSTR (:STR, :START)
```

Solution:

In SQL Server, use SUBSTRING function.

If nessesary negative LEN must be exchanged to 1.

SQL Server Example:

```
SUBSTRING(@STR, @START, @LEN)
SUBSTRING(@STR, @START, LEN(@STR))
```

4.2.2.21 UCASE

In DB2, the UCASE returns capitalized argument.

DB2 Example:

```
UCASE ('JOHN')
```

Solution:

In SQL Server, use UPPER function.

SQL Server Example:

```
UPPER('JOHN')
```

4.2.3 Numberic Functions

4.2.3.1 TRUNCATE, TRUNC

In DB2, returns a datetime value, truncated to the unit specified by format-string.

DB2 Example:

```
TRUNCATE(:DATETIME_VALUE, 'HH')
```

Solution:

In SQL Server, use DATEPART function to decompose argument to parts, then use DATEFROMPARTS or DATETIMEFROMPARTS to compose parts into truncated value.

4.2.3.2 ATAN2

In DB2, returns the arc tangent of x and y coordinates as an angle expressed in radians.

DB2 Example:

```
ATAN2(:X, :Y)
```

Solution:

In SQL Server, use ATAN function with precalculated parameter.

SQL Server Example:

```
ATAN(@Y / @X);
```

4.2.3.3 ATANH

In DB2, returns the hyperbolic arc tangent of a number, in radians.

DB2 Example:

```
ATANH(:x)
```

Solution:

In SQL Server, use formula for arc tangent.

SQL Server Example:

```
LOG((1 + @X) / (1 - @X)) / 2
```

4.2.3.4 COMPARE_DECFLOAT

In DB2, returns a SMALLINT value that indicates whether the two arguments are eq
ual or unordered, or whether one argument is greater than the other.

DB2 Example:

```
COMPARE_DECFLOAT(X, Y)
```

Solution:

In SQL Server, the function is converted to inline expression.

SQL Server Example:

```
CASE
      WHEN X IS NULL OR Y IS NULL THEN 3
      WHEN X = Y THEN 0
      WHEN X < Y THEN 1
      WHEN X > Y THEN 2
      ELSE 3
END
```

4.2.3.5 DECFLOAT_FORMAT, TO_NUMBER

In DB2, returns a DECFLOAT(34) from a character string.

DB2 Example:

```
DECFLOAT_FORMAT(:X)
```

Solution:

In SQL Server, instead DECFLOAT use similar DECIMAL(34, 10) data type.

Before casting clear argument from characters which enabled in DB2 but wrong in SQL Server number presentation.

SQL Server Example:

```
CAST(IIF(CHARINDEX('-', @X) > 0, '-', '') +
REPLACE(REPLACE(REPLACE(REPLACE(REPLACE
(@X, '-', ''),  '<', ''),  '>', ''),  ',', ''),  '$', '')
AS DECIMAL(34, 10));
```

4.2.3.6 DECFLOAT_SORTKEY

The DECFLOAT_SORTKEY function returns a binary value that can be used when sorting DECFLOAT values. The sorting occurs in a manner that is consistent with the IEEE 754R specification on total ordering.

DB2 Example:

```
DECFLOAT_SORTKEY(X)
```

Solution:

SQL Server, numbers don't have to be converted for comparison.

SQL Server Example:

```
@X
```

4.2.3.7 DIGITS

In DB2, returns a character-string representation of the absolute value of a number.

DB2 Example:

```
DIGITS(:X)
```

Solution:

In SQL Server, remove decimal point and add leading zeros to make length according to data type precission.

SQL Server Example:

```
RIGHT('00000000000000000000000000000000000000000' +
    REPLACE(CAST(
        IIF(CAST(SQL_VARIANT_PROPERTY(@X,'BASETYPE') AS VARCHAR(20)) IN
        ('FLOAT', 'MONEY', 'REAL', 'CHAR', 'NCHAR', 'NVARCHAR',
'VARCHAR'),
        CAST(@X AS DECIMAL(31, 6)), @X)
    AS VARCHAR(MAX)), '.', ''),
    CAST(SQL_VARIANT_PROPERTY(@X,'PRECISION') AS INT));
```

4.2.3.8 MOD

In DB2, returns the remainder of the first argument divided by the second argument.

DB2 Example:

```
MOD(:X, :Y)
```

Solution:

In SQL Server, use % operator, which returns the remainder of one number divided by another.

SQL Server Example:

```
@X % @Y
```

4.2.3.9 MULTIPLY_ALT

In DB2, returns the product of two arguments as a decimal value. This function is useful when the sum of the argument precisions is greater than 31.

DB2 Example:

```
MULTIPLAY_ALT(:X, :Y)
```

Solution:

In SQL Server, use * operator.

SQL Server Example:

```
@X * @Y
```

4.2.3.10 NORMALIZE_DECFLOAT

In DB2, returns a decimal floating-point value that is the result of the argument set to its simplest form.

DB2 Example:

```
NORMALIZE_DECFLOAT(:X)
```

Solution:

In SQL Server, use casting to REAL.

SQL Server Example:

```
CAST(@X AS REAL)
```

4.2.3.11 QUANTIZE

In DB2, returns a decimal floating-point number that is equal in value and sign to the first argument, and whose exponent is equal to the

exponent of the second argument.

Returns @val with the same precission (in fact) as @exp.

By other words, with the same count of significant digits.

DB2 Example:

```
SELECT QUANTIZE(:X, :EXP) FROM SYSIBM.SYSDUMMY1;
```

Solution:

In SQL Server, emulate necessary behavior by T-SQL code. You can wrap it into custom defined scalar function too.

SQL Server Example:

```
DECLARE @I INT = 0;
WHILE @I < 30
BEGIN
    IF CAST(@EXP AS BIGINT) > 0
```

```
        BEGIN
            SET @X = ROUND(@X, 1 - LEN(CAST(@EXP AS BIGINT) + @I);
            BREAK;
        END
        SET @EXP *= 10;
        SET @I += 1;
    END
    SELECT @X;
```

4.2.3.12 TANH

In DB2, returns the hyperbolic tangent of a number.

DB2 Example:

```
TANH(:X)
```

Solution:

In SQL Server, use hyperbolic tangent formula.

SQL Server Example:

```
(EXP(2 * @X) - 1) / (EXP(2 * @X) + 1)
```

4.2.3.13 TOTALORDER

In DB2, returns comparison order as -1, 0, or 1.

DB2 Example:

```
TOTALORDER(:X, :Y)
```

Compares pair values

Solution:

In SQL Server, use IIF function.

SQL Server Example:

```
IIF(@X < @Y, -1, IIF(@X > @Y, 1, 0))
```

4.2.4 Miscellaneous Functions

4.2.4.1 BITAND

In DB2, performs a bitwise AND operation.

DB2 Example:

```
BITAND(:X, :Y)
```

Solution:

In SQL Server, use & operator.

SQL Server Example:

```
@X & @Y
```

4.2.4.2 BITANDNOT

In DB2, clears any bit in the first argument that is in the second argument.

DB2 Example:

```
BITANDNOT(:X, :Y)
```

Solution:

In SQL Server, use & and ~ operators.

SQL Server Example:

```
@X & ~ @Y
```

4.2.4.3 BITNOT

In DB2, performs a bitwise NOT operation.

DB2 Example:

```
BITNOT(:X)
```

Solution:

In SQL Server, use ~ operator.

SQL Server Example:

```
~@X
```

4.2.4.4 BITOR

In DB2, performs a bitwise OR operation.

DB2 Example:

```
BITOR(:X, :Y)
```

Solution:

In SQL Server, use | operator.

SQL Server Example:

```
@X | @Y
```

4.2.4.5 BITXOR

In DB2. performs a bitwise exclusive OR operation.

DB2 Example:

```
BITXOR(:X, :Y)
```

Solution:

In SQL Server, use ^ operator.

SQL Server Example:

```
@X ^ @Y
```

4.2.4.6 DECODE

In DB2, compares each specified expression2 to expression1. If expression1 is equal to expression2, or both expression1 andexpression2 are null, the value of the following result-expresssion is returned. If no expression2 matches expression1, the value ofelse-expression is returned; otherwise a null value is returned.

DB2 Example:

```
DECODE(:X, :Y1, :Z1, :Y2, :Z2, :Z)
```

Solution:

In SQL Server, use IIF or CASE function.

SQL Server Example #1:

```
IIF(@X = @Y1, @Z1, IIF(@X = @Y2), @Z2, @Z) -- SUPPOSABLY @X IS NOT NULL
```

SQL Server Example #2:

```
CASE
   WHEN @X = @Y1 OR (@X IS NULL AND @Y1 IS NULL) THEN @Z1
   WHEN @X = @Y2 OR (@X IS NULL AND @Y2 IS NULL) THEN @Z2
   ELSE @Z
END
```

4.2.4.7 GREATEST, MAX

In DB2, returns the maximum value in a set of values.

DB2 Example:

```
GREATEST(:X1, :X2, :X3, :X4, :X5)
```

Solution:

In SQL Server, use MAX aggregate function.

SQL Server Example:

```
SELECT MAX(C) FROM (VALUES (@X1), (@X2), (@X3), (@X4), (@X5)) T(C)
```

4.2.4.8 HEX

In DB2, returns a hexadecimal representation of a value.

DB2 Example:

```
HEX(:X)
```

Solution:

In SQL Server, use casting to VARBINARY, then converting to VARCHAR.

SQL Server Example:

```
CONVERT(VARCHAR(MAX), CAST(@X AS VARBINARY(MAX)), 1)
```

4.2.4.9 IDENTITY_VAL_LOCAL

In DB2, returns the most recently assigned value for an identity column.

Solution:

In SQL Server, use SCOPE_IDENTITY function.

SQL Server Example:

```
SCOPE_IDENTITY()
```

4.2.4.10 IFNULL, NVL

In DB2, returns the first nonnull argument.

DB2 Example:

```
IFNULL(:X, :Y)
```

Solution:

In SQL Server, use COALESCE function.

SQL Server Example:

```
COALESCE(@X, @Y)
```

4.2.4.11 LEAST, MIN

In DB2, returns the minimum value in a set of values.

DB2 Example:

```
LEAST(:X1, :X2, :X3, :X4, :X5)
```

Solution:

In SQL Server, use MIN aggregate function.

SQL Server Example:

```
SELECT MIN(C) FROM (VALUES (@X1), (@X2), (@X3), (@X4), (@X5)) T(C)
```

4.2.4.12 NVL2

In DB2, returns the second argument when the first argument is not NULL. If the first argument is NULL, the third argument is returned.

DB2 Example:

```
NVL2(:TEST, :THEN, :ELSE)
```

Solution:

In SQL Server, use IIF function.

SQL Server Example:

```
IIF (@TEST IS NOT NULL, @THEN, @ELSE)
```

4.2.5 Date / Time Functions

4.2.5.1 ADD_MONTHS

In DB2, returns a datetime value that represents expression plus a specified number of months.

DB2 Example:

```
ADD_MONTHS(:X, :QTY)
```

Solution:

In SQL Server, use function DATEADD which returns a specified date with the specified number interval (signed integer) added to a specified datepart of that date.

SQL Server Example:

```
DATEADD(MONTH, @QTY, @X)
```

4.2.5.2 DAYNAME

In DB2, returns a character string containing the name of the day (for example, Friday) for the day portion of expression, based on locale-name or the value of the special register CURRENT LOCALE LC_TIME.

DB2 Example:

```
DAYNAME(@X)
```

Solution:

In SQL Server, use function DATENAME which returns a character string that represents the specified datepart of the specified date

SQL Server Example:

```
DATENAME(WEEKDAY, @X)
```

4.2.5.3 DAYOFMONTH

The DAYOFMONTH function returns the day part of a value. The function is similar to the DAY function, except DAYOFMONTH does not support a date or timestamp duration as an argument.

DB2 Example:

```
DAYOFMONTH(@X)
```

Solution:

In SQL Server, use function DAY which returns an integer representing the day (day of the month) of the specified date.

SQL Server Example:

```
DAY(@X)
```

4.2.5.4 DAYOFWEEK

In DB2, returns the day of the week from a value, where 1 is Sunday and 7 is Saturday.

DB2 Example:

```
DAYOFWEEK(@X)
```

Solution:

In SQL Server, use tricky formula from the example.

SQL Server Example:

```
(DATEPART(WEEKDAY, @X) + @@DATEFIRST + 6) % 7 + 1
```

4.2.5.5 DAYOFWEEK_ISO

In DB2, returns the day of the week from a value, where 1 is Monday and 7 is Sunday.

DB2 Example:

```
DAYOFWEEK_ISO(:X)
```

Solution:

In SQL Server, use tricky formula from the example.

SQL Server Example:

```
(DATEPART(WEEKDAY, @X) + @@DATEFIRST + 5) % 7 + 1
```

4.2.5.6 DAYOFYEAR

In DB2, returns the day of the year from a value.

DB2 Example:

```
DAYOFYEAR(:X)
```

Solution:

In SQL Server, use DATEPART function.

SQL Server Example:

```
DATEPART(DAYOFYEAR, @X)
```

4.2.5.7 DAYS

In DB2, returns an integer representation of a date.

DB2 Example:

```
DAYS(:X)
```

Solution:

In SQL Server, use DATEDIFF function with the birth of Christ as first argument.

SQL Server Example:

```
DATEDIFF(DD, CAST('0001-01-01' AS DATE), @X) + 1
```

4.2.5.8 EXTRACT

In DB2, returns a portion of a date or timestamp based on the arguments.

DB2 Example:

```
EXTRACT(YEAR FROM :X)
```

Solution:

In SQL Server, use DATEPART function.

SQL Server Example:

```
DATEPART(YEAR, @X)
```

4.2.5.9 HOUR

In DB2, returns the hour part of a value.

DB2 Example:

```
HOUR(:X)
```

Solution:

In SQL Server, use DATEPART function.

SQL Server Example:

```
DATEPART(HOUR, @X)
```

4.2.5.10 JULIAN_DAY

In DB2, returns an integer value representing the number of days from January 1, 4712 B.C. to the date specified in the argument.

DB2 Example:

```
JULIAN_DAY(:X)
```

Solution:

In SQL Server, use DATEDIFF function with the birth of Christ as first argument plus 1721426 as difference between calendars.

SQL Server Example:

```
DATEDIFF(DD, CAST('0001-01-01' AS DATE), @DATE) + 1721426
```

4.2.5.11 LAST_DAY

In DB2, returns a datetime value that represents the last day of the month of the argument.

DB2 Example:

```
LAST_DAY(EXPR)
```

Solution:

In SQL Server, use EOMONTH function.

SQL Server Example:

```
EOMONTH(@X)
```

4.2.5.12 MICROSECOND

In DB2, returns the microsecond part of a value.

DB2 Example:

```
MICROSECOND(:X)
```

Solution:

In SQL Server, use DATEPART function.

SQL Server Example:

```
DATEPART(MICROSECOND, @X)
```

4.2.5.13 MIDNIGHT_SECONDS

In DB2, returns an integer value representing the number of seconds between midnight and a specified time value.

DB2 Example:

```
MIDNIGHT_SECOND(:X)
```

Solution:

In SQL Server, calculate number of seconds since midnight using DATEDIFF.

SQL Server Example:

```
DATEDIFF(SECOND, CAST('00:00:00' AS TIME), CAST (@X AS TIME))
```

4.2.5.14 MINUTE

In DB2, returns the minute part of a value.

DB2 Example:

```
MINUTE(:X)
```

Solution:

In SQL Server, use DATEPART function.

SQL Server Example:

```
DATEPART(MINUTE, @X)
```

4.2.5.15 MONTHNAME

In DB2, returns a character string containing the name of the month (for example, January) for the month portion of expression.

DB2 Example:

```
MONTHNAME(:X)
```

Solution:

In SQL Server, use function DATENAME which returns a character string that represents the specified datepart of the specified date

SQL Server Example:

```
DATENAME(MONTH, @X)
```

4.2.5.16 MONTHS_BETWEEN

Returns an estimate of the number of months between ARGUMENT #1 and ARGUMENT #2.

DB2 Example:

```
MONTHS_BETWEEN(:ARG1, :ARG2)
```

Solution:

In SQL Server, use DATEDIFF function, but note that strictly speaking it is no equivalent to DB2 result.

SQL Server Example:

```
DATEDIFF(MONTH, @ARG1, @ARG2)
```

4.2.5.17 QUARTER

In DB2, returns an integer that represents the quarter of the year in which a date resides.

DB2 Example:

```
QUARTER(:X)
```

Solution:

In SQL Server, use DATEPART function.

SQL Server Example:

```
DATEPART(QUARTER, @X)
```

4.2.5.18 SECOND

In DB2, returns the seconds part of a value.

DB2 Example:

```
SECOND(:X)
```

Solution:

In SQL Server, use formula.

SQL Server Example:

```
DATEPART(SECOND, @X) +
ROUND(DATEPART(NANOSECOND, @DATE) / 1000000000.000000000 , @SCALE, 12)
```

4.2.5.19 TIMESTAMP_ISO, TO_TIMESTAMP

In DB2, returns a timestamp value based on a date, time, or timestamp argument. If the argument is a date, it inserts zero for all the time elements. If the argument is a time, it inserts the value of CURRENT DATE for the date elements, and zero for the fractional time element.

DB2 Example:

```
TIMESTAMP_ISO(:X)
```

Solution:

In SQL Server, use convertion into DATETIME2 data type.

SQL Server Example:

```
CAST(@X AS DATETIME2)
```

4.2.5.20 WEEK

In DB2, returns the week of the year from a value, where the week starts with Sunday.

DB2 Example:

```
WEEK(:X)
```

Solution:

In SQL Server, use xxxxx function.

SQL Server Example:

```
DATEDIFF(WEEK, DATENAME(YEAR, @X) + '-01-01', @X) + 1
```

4.2.5.21 WEEK_ISO

In DB2, returns the week of the year from a value, where the week starts with Monday.

DB2 Example:

```
WEEK_ISO(:X)
```

Solution:

In SQL Server, use DATEPART function.

SQL Server Example:

```
DATEPART(ISO_WEEK, @X)
```

4.2.6 Casting Functions

4.2.6.1 BIGINT

In DB2, converts argument to Big Integer.

Date and time are converted particularly (not as cast()).

DB2 Example:

```
BIGINT(:X)
```

Solution:

In SQL Server, use castint to BIGINT. Different data types are converted by different way. See examples.

SQL Server Example for DATE:

```
CAST(CONVERT(VARCHAR(10), @X, 112) AS BIGINT)
```

SQL Server Example for TIME:

```
CAST(REPLACE(CONVERT(VARCHAR(MAX), @X, 120), ':', '') AS BIGINT)
```

SQL Server Example for DATETIME, DATETIME2, DATETIMEOFFSET, SMALLDATETIME:

```
CAST(REPLACE(REPLACE(REPLACE(CONVERT(VARCHAR(MAX), CAST(@X AS DATETIME2),
120), '-', ''), ' ', ''), ':', '') AS BIGINT)
```

4.2.6.2 BINARY, VARBINARY, BLOB

In DB2, the BINARY function returns a BINARY (fixed-length binary string) representation of a string of any type or of a row ID type.

DB2 Example:

```
BINARY(:STR)
VARBINARY(:STR, LEN)
```

Solution:

In SQL Server, use casting to VARBINARY.

SQL Server Example:

```
CAST(S AS VARBINARY(MAX))
CAST(S AS VARBINARY(LEN))
```

4.2.6.3 CLOB, TO_CLOB

In DB2, the CLOB function returns a CLOB representation of a character string type. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string data type before the function is executed.

DB2 Example:

```
CLOB(:STR)
TO_CLOB(:STR, :LEN)
```

Solution:

In SQL Server, use casting to VARCHAR.

SQL Server Example:

```
CAST(@STR AS VARCHAR(MAX))
CAST(@STR AS VARCHAR(@LEN))
```

4.2.6.4 DATE

In DB2, returns a DATE from a argument.

DB2 Example:

```
DATE ('2015-10-20')
```

Solution:

In SQL Server, use different ways for different dada types. See examples.

SQL Server Example for numeric argument:

```
DATEADD(D, CAST(@X AS BIGINT) - 1, CAST('00010101' AS DATE))
```

SQL Server Example for string argument contain '.':

```
CONVERT(DATE, @X, 104)
```

SQL Server Example for string argument contain '-':

```
CONVERT(DATE, @X, 120)
```

SQL Server Example for argument of other data type:

```
CAST(@X AS DATE)
```

4.2.6.5 DBCLOB

In DB2, returns a DBCLOB representation of a string.

DB2 Example:

```
DBCLOB(:STR)
DBCLOB(:STR, :LEN)
```

Solution:

In SQL Server, use casting to VARCHAR.

SQL Server Example:

```
CAST(@STR AS VARCHAR(MAX))
CAST(@STR AS VARCHAR(@LEN))
```

4.2.6.6 DECFLOAT

In DB2, returns the decimal floating-point representation of a value.

DB2 Example:

```
DECFLOAT(:STR)
DECFLOAT(:STR, :DEC)
```

Solution:

In SQL Server, use casting into decimal(38, 10).

SQL Server Example:

```
CAST(@STR AS DECIMAL(38,10))
CAST(REPLACE(@STR, @DEC, '.') AS DECIMAL(38,10))
```

4.2.6.7 DECIMAL, DEC

In DB2, returns a DECIMAL representation of a value. Dates are converted to YYYYMMDDHHmiSS.mmm

DB2 Example:

```
DECIMAL(:X, :SCALE, :DEC)
DECIMAL(:X, :SCALE)
DECIMAL(:X)
```

Solution:

In SQL Server, use casting into decimal(31, 10) after cleaning argument from wrong characters.

SQL Server Example for DATE:

```
CAST(CONVERT(VARCHAR(10), CAST(@X AS DATE), 112) AS DECIMAL(31, 10))
```

SQL Server Example for TIME:

```
CAST(REPLACE(CONVERT(VARCHAR(MAX), @X, 120), ':', '') AS DECIMAL(31, 10))
```

SQL Server Example for DATETIME, DATETIME2, DATETIMEOFFSET, SMALLDATETIME:

```
CAST(REPLACE(REPLACE(REPLACE(CONVERT(VARCHAR(30), @X, 120),
'-', ''), ' ', ''), ':', '') AS DECIMAL(31, 10))
```

SQL Server Example for strings:

```
CAST(REPLACE(@X), @DEC, '.') AS DECIMAL(31, 10))
```

SQL Server Example for other data types:

```
CAST(CAST(@X AS VARCHAR(MAX)) AS DECIMAL(31, 10)
```

4.2.6.8 DOUBLE_PRECISION, DOUBLE

In DB2, returns the floating-point representation of a value.

DB2 Example:

```
DOUBLE_PRECISION(:X)
```

Solution:

In SQL Server, use casting into DOUBLE PRECISSION.

SQL Server Example:

```
CAST(@X AS DOUBLE PRECISION)
```

4.2.6.9 EMPTY_BLOB

In DB2, returns a zero-length value of the associated data type.

DB2 Example:

```
EMPTY_BLOB
```

Solution:

In SQL Server, use VARBINARY of zero length.

SQL Server Example:

```
CAST('' AS VARBINARY)
```

4.2.6.10 EMPTY_CLOB, EMPTY_DBCLOB, and EMPTY_NCLOB

Return a zero-length value of the associated data type.

Solution:

In SQL Server, use empty string

SQL Server Example:

```
''
```

4.2.6.11 FLOAT

In DB2, returns a FLOAT representation of a value.

DB2 Example:

```
FLOAT(:X)
```

Solution:

In SQL Server, use casting to FLOAT(37).

SQL Server Example:

```
CAST(@X AS FLOAT(37))
```

4.2.6.12 INTEGER, INT

In DB2, returns an INTEGER representation of a value.

Date and time are converted particularly (not as cast()).

DB2 Example:

```
INTEGER(:x)
```

Solution:

In SQL Server, use castint to INT. Different data types are converted by different way. See examples.

SQL Server Example for DATE:

```
CAST(CONVERT(VARCHAR(10), @X, 112) AS INT)
```

SQL Server Example for TIME:

```
CAST(REPLACE(CONVERT(VARCHAR(MAX), @X, 120), ':', '') AS INT)
```

SQL Server Example for DATETIME, DATETIME2, DATETIMEOFFSET, SMALLDATETIME:

```
CAST(REPLACE(REPLACE(REPLACE(CONVERT(VARCHAR(MAX), CAST(@X AS DATETIME2),
120), '-', ''), ' ', ''), ':', '') AS INT)
```

4.2.6.13 NCLOB, TO_NCLOB

In DB2, convert argument to nclob presentation.

DB2 Example:

```
NCLOB(:STR)
NCLOB(:STR, :LEN)
```

Solution:

In SQL Server, instead NCLOB the VASRCHAR is used.

SQL Server Example:

```
CAST(@STR AS VARCHAR(MAX))
CAST(@STR AS VARCHAR(@LEN))
```

4.2.6.14 REAL

In DB2, returns the single-precision floating-point representation of a value.

DB2 Example:

```
REAL(:X)
```

Solution:

In SQL Server, use casting in REAL.

SQL Server Example:

```
CAST(@X AS REAL)
```

4.2.6.15 SMALLINT

In DB2, returns a SMALLINT representation of a value.

DB2 Example:

```
SMALLINT(:x)
```

Solution:

In SQL Server, use casting into SMALLINT.

SQL Server Example:

```
CAST(@X AS SMALLINT)
```

4.2.6.16 TIME

In DB2, returns a TIME from a value.

DB2 Example:

```
TIME(:X)
```

Solution:

In SQL Server, use xxxxx function.

SQL Server Example:

```
CAST(CAST(@X AS DATETIME2) AS TIME)
```

4.2.6.17 TIMESTAMP

In DB2, returns a TIMESTAMP from a value.

For strings 'yyyyxxddhhmmss.d' format is supported.

DB2 Example:

```
TIMESTAMP(X)
```

Solution:

In SQL Server, use xxxxx function.

SQL Server Example for strings:

```
IIF(LENGTH(@X) = 13,
      CONVERT(DATETIME, SUBSTRING(CAST(@X AS VARCHAR(MAX)), 1, 8), 112) +
      CAST(SUBSTRING(@X),  9, 2) + ':' +
              SUBSTRING(@X), 11, 2) + '.' +
              SUBSTRING(@X), 13, 1) AS DATETIME),
      CAST(@X AS DATETIME2))
```

SQL Server Example for another data types:

```
CAST(@X AS DATETIME2)
```

4.2.7 Aggregation Functions

4.2.7.1 LISTAGG

In DB2, aggregates a set string elements into one string by concatenating the strings.

DB2 Example:

```
SELECT LISTAGG(COL1) FROM TABLE1
```

Solution:

In SQL Server, use FOR XML clause.

SQL Server Example:

```
SELECT COL1 + '' FROM TABLE1 FOR XML PATH('')
```

4.2.7.2 MEDIAN

The MEDIAN function returns the median of a set of numbers.

DB2 Example:

```
SELECT MEDIAN(SALARY)
FROM EMPLOYEE
WHERE WORKDEPT = 'D11'
```

Solution:

In SQL Server, emulate median evaluating through SQL queries.

SQL Server Example:

```
SELECT TOP 1 SALARY
FROM
(
      SELECT TOP 50 PERCENT SALARY
      FROM EMPLOYEE
      WHERE WORKDEPT = 'D11'
      ORDER BY SALARY
) Q
ORDER BY SALARY DESC
```

4.2.7.3 STDDEV, STDDEV_SAMP

In DB2, returns the standard deviation of a set of numbers.

DB2 Example:

```
STDDEV(DISTINCT COL1)
```

Solution:

In SQL Server, use STDEVP function which returns the statistical standard deviation for the population for all values in the specified expression.

SQL Server Example:

```
STDEVP(DISTINCT COL1)
```

4.2.7.4 VARIANCE, VARIANCE_SAMP

In DB2, returns the variance of a set of numbers.

DB2 Example:

```
VARIANCE(COL1)
```

Solution:

In SQL Server, use function VARP which returns the statistical variance for the population for all values in the specified expression.

SQL Server Example:

```
VARP(COL1)
```

5.0 Data Migration

Data migration is the process of transferring data from a source database to a target database. Both SQL Server and DB2 provide built-in utilities for data migration.

DB2 provides the EXPORT utility for exporting data from DB2 to flat (text) files. For information about DB2 EXPORT, go to pic.dhe.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=%2Fcom.ibm.db2.luw.sql.ref.doc%2Fdoc%2Fr0059482.html.

SQL Server offers the Bulk Copy Program (bcp), the BULK INSERT statement, and the SQL Server Integration Services (SSIS) platform of utilities for importing data. These utilities can be used to perform data migration from DB2 to SQL Server.

Bulk Copy Program (bcp) is a command-line utility that uses the ODBC bulk copy API. For bcp, data must be input as ASCII text files. For more information about BCP, go to msdn.microsoft.com/en-us/library/ms162802.aspx.

BULK INSERT is a T-SQL statement that has same functionality as bcp, and also imports DB2 data as flat files. The major difference from bcp is that BULK INSERT is available from within a database session. For more information about BULK INSERT, go to msdn.microsoft.com /en-us/library/ms188365.aspx.

SQL Server Integration Services (SSIS) is a powerful set of tools that can be used to extract, transform, and load DB2 data to SQL Server databases. SSIS wizards simplify the process of defining and performing the import. SSIS also provides access to the BULK INSERT statement in the BULK INSERT task. SSIS offers two methods for importing:

- Load data from flat files.
- Provide direct connectivity with DB2 using OLEDB AND ODBC providers to extract data from DB2.

This section details the steps to perform while migrating the data, in these three areas:

- Pre-implementation
- Implementation
- Post-implementation

The various SQL Server options for migrating data from DB2 to SQL Server are presented in section 5.2, “Implementation Tasks.”

Note that Azure SQL DB doesn’t support working with the file system.

5.1 Pre-Implementation Tasks

Perform these seven tasks before performing the migration:

1. In SQL Server, back up the empty target SQL Server database.
2. In DB2, export DB2 data to flat (text) files using the DB2 UDB export utility, exporting one flat file for each table.
3. In SQL Server, set the recovery model of the database to Bulk-logged using the command below:

```
ALTER DATABASE <DATABASENAME>
SET RECOVERY BULK LOGGED
```

4. In SQL Server, disable all constraints.

   Note: By default, constraints are not checked when you use bcp or BULK INSERT.

5. In SQL Server, disable triggers on all tables into which data will be loaded using the command below:

```
ALTER TABLE <TABLENAME> DISABLE TRIGGER ALL
```

In the BULK INSERT statement, use the FIRE_TRIGGERS option to enable and disable triggers.

6. In SQL Server, handle identity column inserts:
   - In bcp, use the –E switch.
   - In BULK INSERT, use the KEEPIDENTITY argument.

5.2 Implementation Tasks

The data migration implementation is the actual transfer of data from the source DB2 database to the SQL Server database. Tis can be performed in either one step or two steps. When trying to determine the most efficient method for loading a database, take into account the volume of data to be loaded.

Single-step process: Data is extracted from the DB2 database by direct connectivity to DB2, using OLE DB or ODBC. In situations in which a small to moderate amount of data is involved, consider using the single-step method with SSIS Import and Export Wizard. The single-step method is also preferable when the data row contains large binary objects or graphical objects, or when the length of the data row varies significantly, which makes it costly to store and encode the fields in the text file format.

Two-step process: Data is extracted from DB2 to ASCII flat files, and then loaded to the SQL Server database using bcp or the BULK INSERT statement. The bcp utility and the BULK INSERT statement provide the two most efficient methods of loading large data sets. Use BULK INSERT instead of the bcp utility if you are running the process on the computer that is running SQL Server; BULK INSERT is available from within a database session.

The following sections outline the steps for performing the implementation using different tools.

5.2.1 One-Step Process Using SSIS Import and Export Wizard

The following steps outline how to create a SSIS package to directly connect to DB2 database using an OLE DB provider, and how to import data directly from a DB2 database without creating an intermediate file.

To import by direct connection using SSIS

1. From SQL Server Management Studio, launch the SSIS Import and Export Wizard: right-click the target SQL Server database, point to **Tasks**, and then click **Import Data**.
2. On the **Welcome** page, click **Next**.
3. On the **Choose a Data Source** page, in the **Data Source** list, click **IBM OLE DB Provider for DB2**, and then click **Properties**.
4. In the **Properties** dialog box, enter the properties for your data source.
5. On the **Choose a Destination** page, in the **Destination** box, select **Microsoft OLEDB for SQL Server** and provide server and login information for the destination SQL Server database.
6. In the **Specify Table Copy or Query** page, do one of the following:
   - Select the **Copy data from one or more tables or views** option. From the list of tables and views that appears, choose which tables and views you want to import.
   - Select the **Write a query to specify the data to transfer** option. From the **Provide a Source Query** page, enter your query.
7. On the **Save and Run Package** page, click **Finish** to run the import package immediately, or click **Save** to run the package at later time.

   Note: To customize the SSIS package, click **Save**.

5.2.2 Two-Step Process Using the bcp Utility

The bcp utility is a command-line utility that can be used to load data to SQL Server from flat files.

Syntax

```
BCP DBNAME.SCHEMA.TABLENAME IN DATA_FILE –T FIELD_TERMINATOR –S
SERVER_NAME –U LOGIN_ID –P PASSWORD
```

where -t specifies the field delimiter.

To make the bcp process repeatable, create a format file and use the -f switch to specify the format file for future imports.

The following arguments should be taken into consideration while importing large volume of data from DB2.

-e error_file logs errors to an error file during the import procedure.

-b batch_size specifies the size of the batch; the default is all the rows. Each batch is imported and logged in a separate transaction. After a given transaction is committed, the rows that have been imported by that transaction are committed. If the operation fails, only those rows that have been imported from the current batch are rolled back, and you can resume importing data starting at the beginning of the failed batch, rather than at the beginning of the data file.

-E specifies that the identity value or values in the imported data file are to be used for the identity column. If -E is not specified, then the identity values for this column in the data file being imported are ignored, and SQL Server automatically assigns unique values based on the seed and increment values that were specified during table creation.

Note that Azure SQL DB doesn't support working with the file system.

5.2.3 Two-Step Process Using BULK INSERT

BULK INSERT is a Transact-SQL statement that has the same functionality as bcp, and can be used to import DB2 data to SQL Server from a flat file. Choose BULK INSERT rather than bcp to make the table migration an integral part of a Transact-SQL batch or a stored procedure.

Syntax

```
BULK INSERT DBNAME.OWNER.TABLE FROM FILENAME WITH(DATAFILETYPE='CHAR',
FIELDTERMINATOR=',' , ROWTERMINATOR='\N')
```

The following additional arguments should be taken into consideration while importing large volume of data from DB2:

KEEPIDENTITY specifies that identity values in the flat file are to be used for the identity column.

FIRETRIGGERS specifies that triggers are not suppressed.

CHECK_CONSTRAINTS specifies that constraints on the table must be checked.

BATCHSIZE specifies the number of rows in the batch. Each batch is copied as an individual transaction.

TABLOCK specifies that a table-level lock is acquired for the duration of the bulk-import operation.

5.2.4 Two-Step Process Using SSIS

This section outlines two ways that SSIS can be used to import data from flat (text) files in a DB2 database: with and without the SSIS Import and Export Wizard. The wizard is easier to use, but the other option is more flexible.

To import from a flat file using the SSIS Import and Export Wizard

1. From SQL Server Management Studio, launch the **SSIS Import and Export Wizard**: right-click the target SQL Server database, point to **Tasks**, and then click **Import Data**.
2. On the **Welcome** page, click **Next**.
3. On the **Choose a Data Source** page, do the following:

- In the **Data Source** list, select **Flat File Source**.
- In the **File name** box, enter the name of the file that contains the DB2 table data.
- Move to the **Columns** subpage, and then choose the appropriate column delimiter and row delimiter.

4. On the **Choose a Destination** page, in the **Destination** box, select **Microsoft OLEDB for SQL Server** and provide server and login information for the destination SQL Server database.
5. On the **Save and Run Package** page, click **Finish** to run the import package immediately, or click **Save** to run the package at later time or to do customization.

To import from a flat file using SSIS

1. In Microsoft Visual Studio®, create a new project from the **Integration Services Project** template.
2. On the **Data Flow** tab, click **Data Flow Sources** in the **Toolbox**, and then choose **Flat File Source**.
3. Create a new flat file connection manager that is based on your flat file configuration; have the manager point to the file containing the DB2 table data.
4. Right-click the component, and then click **Show Advanced Editor**.
5. On the **Input and Output Properties** tab, expand **Flat File Source Output**, and then expand **Output Columns**.
6. For each column, set the **Fast Parse** custom property to **True** to increase performance.

   Note: The **FastParse** property indicates whether the column uses the quicker but locale-insensitive fast parsing routines that SSIS provides, or the locale-sensitive standard parsing routines. If the data flow in the package requires locale-sensitive parsing, standard parse is recommended instead of fast parse, in which case set this to False.

7. From **Data Flow Destinations** in the **Toolbox**, click **OLE DB Destination**.
8. In the **OLE DB Destination Editor** dialog box, do the following:
   - In the **OLE DB connection manager** list, click the name of the destination connection that should point to the target SQL Server database.
   - In the **Data access mode** list, click **Table or view - fast load**.
   - In the **Name of the table or the view** list, click the name of the table where you want to dump the data from the file.
9. Select the **Table Lock** and **Keep Identity** check boxes.
10. From the **File** menu, click **Save All** to save the package.
11. Press F5 to run the package.

5.2.5 Methods for Optimizing Bulk Import Performance

Here are a few suggestions for improving the performance of a bulk data import.

- Use BULK INSERT instead of the bcp utility if you are running the process on the computer that is running SQL Server.
- Loading of large amount of data in parallel creates the best performance when dealing with a multiprocessor server. The parallelism can be intra-table or inter-table. Follow these guidelines for parallel data loading:
- Load the data into the same table from multiple clients in parallel, thereby improving the performance of the bulk-copy operation. Run as many load processes as you have available CPUs. If you have eight CPUs, run eight parallel loads.
- Use the TABLOCK hint during parallel loads. Both bcp and BULK INSERT support this hint.

- Divide the data to be imported among clients into same number of data files as there are clients. Place one of file for each of the clients.
- To use the processor most effectively, distribute data evenly among the clients and make sure that all the data files are approximately the same size.
- Make sure that indexes do not exist for any table, because if indexes exist on a table, you cannot perform a parallel load operation by using the TABLOCK hint.
- Use TABLOCK to avoid row-at-a-time locking. Table locking can improve the performance of the bulk-import operation by reducing lock contention on the table.
- Use minimal logging by setting the database recovery model to Bulk-logged or Simple.
- Make the batch size as large as practical. Typically, the larger the batch size, the better the performance of the bulk-import operation. To achieve maximum performance, the batch size specified for each client should be the same as the size of the client's data file.
- Disable triggers in target tables.
- Disable constraints. By default, constraints are ignored when bcp or BULK INSERT is used.

5.3 Post-Implementation Tasks

To ensure the success of the data transfer, perform the following tasks to undo the changes that were made to the schema prior to implementation.

1. Validate the data migration.
2. Re-enable the constraints by typing the following command:

```
ALTER TABLE <TABLE> CHECK CONSTRAINTS ALL
```

3. Recreate the indexes using saved scripts.
4. Enable the triggers by typing the following command:

```
ENABLE TRIGGER ALL ON <TABLE>
```

5. Set the recovery model to FULL by executing the following commands:

```
ALTER DATABASE DATABASE_NAME
SET RECOVERY FULL
```

6. Create a backup of the migrated database.

To validate migrated data, perform the following tasks:

1. **Verify the data load:**
   A. Check the data transfer logs for errors or failures.
   B. Check row counts of every table in the destination SQL Server database; row counts should match the source DB2 database.
   C. If any discrepancy is found in step B, troubleshoot to find the missing rows using logs.
   D. Check the sum (SUM) of several numeric columns in both the source and target databases, to make sure they are the same.
2. **Validate the data integrity.** Ensure that integrity is automatically checked by enabling or creating constraints by using the WITH CHECK clause when adding constraints.

6.0 Terminology Mapping

Table 6-1 shows the terminology mapping of physical objects between DB2 UDB and SQL Server.

Table 6-1: Physical Objects

DB2 UDB	SQL Server
Tablespace	Filegroup

Table 6-2 shows the terminology mapping of logical objects between DB2 UDB and SQL Server.

Table 6-2: Logical Objects

DB2 UDB	SQL Server
Server	Server
Instance	Instance
Database	Database
Database Manager configuration file	Windows registry
Database configuration file	Windows registry
Catalog tables	System tables (master DB)

Table 6-3 shows the terminology mapping of database objects between DB2 UDB and SQL Server.

Table 6-3: Database Objects

DB2 UDB	SQL Server
Schema	Schema
Table	Table
Table constraint	Rule and table constraint
Index	Index
View	View
Transaction log (also called Recovery log)	Transaction log
Archive log	Transaction log dump
Users and groups (operating system)	Users, groups, and roles

Table 6-4 shows the terminology mapping of administration and usage between DB2 UDB and SQL Server.

Table 6-4: Administration and Usage

DB2 UDB	SQL Server
Control Center	Management Studio
Tables assigned to a tablespace	Tables assigned to a filegroup
Containers assigned to tablespaces	Database files assigned to filegroups
Administration commands (get db cfg)	System stored procedures

Binding packages	n/a
Backup database	Backup database
Archive online logs files	Backup log
Restore from backup	Restore database
Roll-forward recovery	Point in Time Recovery
Crash recovery	Automatic recovery
Run statistics (runstats command)	UPDATE statistics, CREATE statistics
LOAD, IMPORT, and EXPORT	BCP (bulk copy program)

7.0 Conclusion

From this migration guide you learned the differences between DB2 and SQL Server 2014 database platforms, and the steps necessary to convert a DB2 database to SQL Server.

It explains the algorithms that SSMA for DB2 uses to perform this conversion so that you can better understand the processes that are executed when you run the SSMA **Convert Schema** and **Migrate Data** commands. For those cases when SSMA does not handle a particular migration issue, approaches to manual conversion are included.

7.1 About DB Best Technologies

DB Best Technologies is a leading provider of database and application migration services and custom software development. We have been focused on heterogeneous database environments (SQL Server, Oracle, Sybase, DB2, MySQL) since starting at 2002 in Silicon Valley. Today, with over 75 employees in the United States and Europe, we develop database tools and provide services to customers worldwide.

DB Best developed migration tools to automate conversion between SQL dialects. In 2005 Microsoft acquired this technology, which later became a family of SQL Server Migration Assistant (SSMA) products. We continue to develop new versions of SSMA, and support Microsoft customers who are migrating to SQL Server.

We also provide migration services covering all major steps of a typical migration project: complexity assessment, schema conversion, data migration, application conversion, testing, integration, deployment, performance tuning, training, and support.

For more details, visit us at www.dbbest.com, e-mail us at info@dbbest.com, or call 1-408-202-4567.

7.2 Useful Resources

Database migration portal powered by DB Best Technologies, available at www.databasemigrate.com/.

DB Best blogs dedicated to various aspects of database migration, available at http://dbbest.com/blog/.

For more information:

http://www.microsoft.com/sqlserver/: SQL Server Web site

http://technet.microsoft.com/en-us/sqlserver/: SQL Server TechCenter

http://msdn.microsoft.com/en-us/sqlserver/: SQL Server DevCenter

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

- Are you rating it high due to having good examples, excellent screenshots, clear writing, or another reason?
- Are you rating it low due to poor examples, fuzzy screenshots, unclear writing?

This feedback will help us improve the quality of the white papers we release.

Send feedback.