Guide to Migrating from MySQL to SQL Server 2014 and Azure SQL DB

SQL Server Technical Article

Writers: Alexander Pavlov (DB Best Technologies), Yuri Rusakov (DB Best Technologies), Valentin Panarin (DB Best Technologies), Natalia Vilenskaia (DB Best Technologies)

Technical Reviewer: Dmitry Balin (DB Best Technologies)

Published: November 2014

Applies to: SQL Server 2014

Summary: In this migration guide you will learn the differences between the MySQL and SQL Server 2014 database platforms, and the steps necessary to convert a MySQL database to SQL Server.

Created by: DB Best Technologies LLC

2535 152nd Ave NE, Redmond, WA 98052

Tel.: +1-855-855-3600

E-mail: info@dbbest.com

Web: www.dbbest.com

Copyright

Contents

Introduction

This migration guide outlines the procedures, problems, and solutions for migrating from Oracle® MySQL 5.6 to the Microsoft® SQL Server® 2014 database software.

Inside you will find three main sections:

Migrating MySQL Data Types**.** Explains the data type mapping and adds remarks about the related conversion issues.

MySQL Migration Issues**.** Explores the challenges you might encounter when migrating from MySQL to SQL Server 2014 and offers possible solutions.

Migrating MySQL System Functions**.** Examines MySQL system function references, divided into equivalent functions, nonsupported functions, and emulated functions.

MySQL to SQL Server 2014 Migration

Following are the basic, high-level steps for migrating a MySQL database to SQL Server 2014 and what you must know about converting database objects.

Main Migration Steps

To migrate a MySQL database:

1. Decide how you will map MySQL databases to SQL Server 2014. You have two main options:

    - Map each MySQL database to a separate SQL Server database. For example, you could map the MyDB MySQL database to MyDB SQL Server database.

    - Map each MySQL database to a single SQL Server database but a separate schema. For example, you could map the MyDB MySQL database to MySQLDatabases SQL Server database, schema MyDB.

      In SQL Server, schemas are not necessarily linked to a specific user or a login, and one server contains multiple databases.

2. Convert database objects; these are tables, tables constraints, indexes, view, procedures, functions, and triggers.
3. Map data types from the MySQL data type to a SQL Server data type.
4. Rewrite your views, procedures, and functions according to SQL Server syntax.
5. Change your applications as necessary so that they can connect and work with SQL Server 2014.

After a successful database conversion, migrate your data from the old MySQL database to the newly created SQL Server 2014 database. For this task you could use SQL Server Integration Services (SSIS), for example.

Converting Database Objects

This section contains considerations that you must know when converting database objects.

Schema Object Names

In SQL Server 2014, an object name can be up to 128 characters long.

Nonquoted identifier names must follow these rules:

- The first character must be alphanumeric, an underscore (_), an at sign (@), or a number sign (#).

- Subsequent characters can include alphanumeric characters, an underscore, an at (@) sign, a number sign, or a dollar sign.

- The identifier must not be a Transact-SQL reserved word.

- Embedded spaces or special characters are not allowed.

Identifiers that start with @ or a number sign have special meanings. Identifiers starting with @ are local variable names. Those that start with a number sign are temporary table names.

To quote an identifier name in Transact-SQL, you must use square brackets ([]).

Tables, Constraints, Indexes, and Views

Convert tables by using column data type mapping (see Type Mapping later in this guide).

SQL Server 2014 supports the following table (column) constraints: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, and CHECK. Convert each type of constraint according to Transact-SQL syntax.

SELECT statements with VIEW should also be converted according to Transact-SQL SELECT syntax.

Stored Procedures and User Defined Functions

Convert stored procedures and functions by using Transact-SQL syntax.

SQL Server 2014 does not support DML statements in user-defined functions. This means that you cannot change any data from within the function.

Triggers

SQL Server 2014 does not have BEFORE triggers.

Convert multiple BEFORE triggers to a single INSTEAD OF trigger.

Migrating MySQL Data Types

This section explains mappings and differences between MySQL and SQL Server 2014 data types, specific data type handling, and provides solutions for problems related to data types.

Type Mapping

Following are the recommended type mappings for converting table columns, subroutine arguments, returned values, and local variable data types.

MySQL type	SQL Server 2014 mapping	Conversion remarks	Possible mappings
BIT (M)	**varbinary** (8)	The binary value has M bits. M = 1..64	Not applicable
TINYINT (M) BOOL, BOOLEAN =	**smallint**	M is the number of decimal places in the output for this value.	**tinyint**, **smallint**, **int**, **bigint**, **numeric(p,s)**, **decimal(p,s)**, **float(p)**, **double precision**, **real**,

TINYINT (1)			**smallmoney**, **money**
SMALLINT (M)	**smallint**	M is the number of decimal places in the output for this value.	**tinyint**, **smallint**, **int**, **bigint**, **numeric(p,s)**, **decimal(p,s)**, **float(p)**, **double precision**, **real**, **smallmoney**, **money**
MEDIUMINT (M)	**int**	M is the number of decimal places in the output for this value.	**tinyint**, **smallint**, **int**, **bigint**, **numeric(p,s)**, **decimal(p,s)**, **float(p)**, **double precision**, **real**, **smallmoney**, **money**
INT (M) INTEGER (M)	**int**	M is the number of decimal places in the output for this value.	**tinyint**, **smallint**, **int**, **bigint**, **numeric(p,s)**, **decimal(p,s)**, **float(p)**, **double precision**, **real**, **smallmoney**, **money**
BIGINT (M)	**bigint**	M is the number of decimal places in the output for this value.	**tinyint**, **smallint**, **int**, **bigint**, **numeric(p,s)**, **decimal(p,s)**, **float(p)**, **double precision**, **real**, **smallmoney**, **money**
FLOAT (P)	**float (P)**	None.	**numeric(p,s)**, **decimal(p,s)**, **float(p)**, **double precision**, **real**, **smallmoney**, **money**
FLOAT [(P, S)]	**float (24)**	MySQL allows a nonstandard syntax: FLOAT(P,S) or REAL(P,S) or DOUBLE PRECISION(P,S). Here, "(P,S)" means that values are displayed with up to P digits total, of which S digits may be after the decimal point. MySQL performs rounding when storing values.	**numeric(p,s)**, **decimal(p,s)**, **float(p)**, **double precision**, **real**, **smallmoney**, **money**
DOUBLE [(P, S)] DOUBLE PRECISION [(P, S)] REAL [(P, S)]	**float (53)**		**numeric(p,s)**, **decimal(p,s)**, **float**(p), **double precision**, **real**, **smallmoney**, **money**

		If M and D are omitted, values are stored up to the size limits allowed by the hardware.	
DECIMAL [(P [, S])] DEC [(P [, S])] NUMERIC [(P [, S])] FIXED [(P [, S])]	decimal [(P [, S])] numeric [(P [, S])]	Decimal types can have up to 65 digits. For a decimal with a precision of more than 38, use the float or double data type.	**numeric(p,s)**, **decimal(p,s)**, **float(p)**, **double precision**, **real**, **smallmoney**, **money**
DATETIME [(D)]	**datetime2**	MySQL can store dates from 0000-00-00 to 9999-12-31. MySQL can store zero-value of year, month, and year.	**smalldatetime**, **datetime**, **datetime2**
DATE [(D)]	**date**		**smalldatetime**, **datetime**, **datetime2**, **date**
TIME	**time**	Range is '-838:59:59' to '838:59:59'.	**smalldatetime**, **datetime**, **datetime2**, **time**, **varchar**, **nvarchar**
TIMESTAMP	**smalldatetime**	Range is '1970-01-01 00:00:00' to partway through the year 2037. If not defined during conversion, this type gets the current datetime value.	**datetime**, **datetime2**, **rowversion**, **timestamp**, **varbinary(8)**, **binary(8)**
YEAR [(2| 4)]	**smallint**	In four-digit format, allowable values are 1901 to 2155, and 0000. In two-digit format, allowable values are 70 to 69, representing years from 1970 to 2069.	**datetime**, **date**, **varchar(4)**
[NATIONAL] CHAR (N)	**nchar (N)**	Range of N is 0 to 255 characters.	**char**, **varchar**, **nchar**, **nvarchar**

[NATIONAL] CHAR	**nchar**		
[NATIONAL] VARCHAR (N) CHARACTER VARYING (N)	**nvarchar (N | max)**	Range of N is 0 to 65,535. If N is less than or equal to 8000, use nvarchar(N). If it is greater than 8000, use nvarchar(max).	**char**, **varchar**, **nchar**, **nvarchar**
TINYTEXT	**nvarchar (255)**	None.	**char**, **varchar**, **nchar**, **nvarchar**
TEXT (N)	**nvarchar (N | max)**	A TEXT column with a maximum length of 65,535 characters. If N is less than or equal to 8000, use nvarchar(N). If it is greater than 8000, use nvarchar(max).	**char**, **varchar**, **nchar**, **nvarchar**, **varchar(max)**, **nvarchar(max)**
MEDIUMTEXT	**nvarchar (max)**	None.	**char**, **varchar**, **nchar**, **nvarchar**, **varchar(max)**, **nvarchar(max)**
LONGTEXT	**nvarchar (max)**	None.	**char**, **varchar**, **nchar**, **nvarchar**, **varchar(max)**, **nvarchar(max)**
BINARY (N)	**binary (N)**	None.	**binary**, **varbinary**, **char**, **varchar**, **nchar**, **nvarchar**
VARBINARY (N)	**varbinary (N)**	None.	**binary**, **varbinary**, **char**, **varchar**, **nchar**, **nvarchar**
TINYBLOB	**varbinary (255)**	None.	**binary**, **varbinary**, **varbinary(max)**
BLOB (N)	**varbinary (N | max)**	A BLOB column with a maximum length of 65,535 bytes.	**binary**, **varbinary**, **varbinary(max)**

		If N is less than or equal to 8000, use nvarchar(N). If it is greater than 8000, use nvarchar(max).	
MEDIUMBLOB	**varbinary (max)**	None.	**binary**, **varbinary**, **varbinary(max)**
LONGBLOB	**varbinary (max)**	None.	**binary**, **varbinary**, **varbinary(max)**
ENUM		See ENUM and SET Data Types in this guide.	
SET		See ENUM and SET Data Types in this guide.	

Note: MySQL numeric types can have an UNSIGNED flag. These should be converted to the bigger numeric type.

Data Type Migration Issues

This section describes data type conversion issues. Each issue is caused by a MySQL feature that is not supported in SQL Server.

Numeric Data Types

Issue: Unsigned Data Types

All integer types in MySQL (TINYINT, SMALLINT, MEDIUMINT, INT, and BIGINT) can have the UNSIGNED optional attribute. Unsigned values can be used to allow only nonnegative numbers in a column when you need a large upper numeric range for the column.

Unsigned values can also be used to allow only nonnegative values in a column with floating-point (FLOAT, DOUBLE) and fixed-point (DECIMAL) data types.

If you are not using strict mode (that is, neither STRICT_TRANS_TABLES nor STRICT_ALL_TABLES is enabled), MySQL inserts adjusted values for invalid or missing values and produces warnings.

Example:

```
create table numeric_unsigned (t tinyint unsigned, s smallint unsigned,
m mediumint unsigned, i int unsigned, b bigint unsigned);
insert numeric_unsigned values
(255, 65535, 16777215, 4294967295, 18446744073709551615);
```

```
create table point_unsigned (f float unsigned, d double unsigned);
insert point_unsigned values (-1.1234567890,-1.12345678901234567890);
insert point_unsigned values ( 5.1234567890, 5.12345678901234567890);
select * from point_unsigned;
--       0                0
-- 5.12346 5.12345678901235
```

Solution:

To avoid negative values, use CHECK constraints. This is the simplest method but it has one disadvantage—you always get an exception if you try to assign an invalid value.

Another way to avoid negative values is to use an INSERT or UPDATE trigger. This method allows the correction of invalid values before storing them in database.

Issue: Operations with Unsigned Values

If you use subtraction between integer values where one is of type UNSIGNED, the result is unsigned. Prior to MySQL 5.5.5, if the result would otherwise have been negative, it becomes the maximum integer value.

Example:

```
create table unsign (a int unsigned, b int);
insert unsign values (1,2),(4,3),(10,100);

select a-b from unsign;
-- 18446744073709551615
-- 1
-- 18446744073709551526
```

Solution:
Use the CASE function to calculate the result of an operation that uses unsigned values.

Issue: Display Width of Integer Values and ZEROFILL Attribute

MySQL supports specifying the display width of an integer (TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT) value in parentheses following the base keyword for the type (for example, INT(4)). This optional display width specification is used to left-pad the display of values having a width that is less than the width specified for the column.

The display width does not constrain either the range of values that can be stored in the column or the number of digits that are displayed for values having a width that exceeds that specified for the column.

See also: ZEROFILL Attribute

Solution:

Ignore these attributes during the conversion. Format the output data by using functions such as STR and CONVERT.

Issue: ZEROFILL Attribute

When used in conjunction with the optional extension attribute ZEROFILL, the default padding of spaces is replaced with zeros in MySQL.

If you specify ZEROFILL for a numeric column, MySQL automatically adds the UNSIGNED attribute to the column.

Example:

```
create table table_zerofill (a int(2) zerofill, b int(4) zerofill,
c int(8) zerofill, d decimal(5,2) zerofill);
insert table_zerofill values (2,4,8,1.23);
select concat('BEGIN',a,b,c,d,'END') from table_zerofill;
-- BEGIN02000400000008001.23END

create table point_zerofill (f float zerofill, d double zerofill);
insert point_zerofill values (-1.1234567890,-1.12345678901234567890);
insert point_zerofill values ( 5.1234567890, 5.12345678901234567890);
select * from point_zerofill;
-- 000000000000 00000000000000000000
-- 000005.12346 0000005.12345678901235
```

Solution:

Ignore these attributes during the conversion. Format the output data by using functions such as STR, REPLICATE, and REPLACE.

Issue: FLOAT and DOUBLE Data Type Precision and Scale

In MySQL, FLOAT and DOUBLE data types can have precision and scale.

Example:

```
create table table_float (f2 float|double(10,2),
                          f5 float|double(10,5), f7 float|double(10,7));
insert into table_float values (1.1234567,1.1234567,1.1234567);
insert into table_float values (12345.1234567,12345.1234567,12345.1234567);
select * from table_float;
--     1.12     1.12346    1.1234567
-- 12345.12 12345.12305 1000.0000000
```

Solution:

FLOAT and DOUBLE values with precision and scale can be rounded in triggers or in DML statements by using the ROUND function.

Issue: Maximum Number of Digits for DECIMAL Data Type

In MySQL, the maximum number of digits for the DECIMAL data type is 65.

Example:

```
create table table_decimal (d decimal(65), m decimal(65,30));
insert table_decimal values
(12345678901234567890123456789012345678901234567890123456789012345,
 12345678901234567890123456789012345678901234567890123456789012345);
select * from table_decimal;
-- 99999999999999999999999999999999999999999999999999999999999999999
-- 99999999999999999999999999999999.999999999999999999999999999999999
```

Solution:

For decimals with a precision greater than 38, use the **float** or **double** data types.

Date and Time Types

Issue: "Zero" Values

MySQL allows you to store '0000-00-00' as a “dummy date” if you are not using the NO_ZERO_DATE SQL or strict modes.

Illegal DATETIME, DATE, YEAR, or TIMESTAMP values are converted to the “zero” value of the appropriate type ('0000-00-00 00:00:00', '0000-00-00' or '0000').

For date and time types other than TIMESTAMP in MySQL, the default is the appropriate zero value for the type. For the first TIMESTAMP column in a table, the default value is the current date and time.

Example:

```
create table date_zero (dt datetime not null, d date not null,
       t time not null, y year not null, ts timestamp not null);
insert date_zero values ();
select * from date_zero;
-- 0000-00-00 00:00:00 | 0000-00-00 | 00:00:00 | 0000 | 2006-12-19 18:24:49
insert date_zero values
('20060229150000','20060229','900:15:20','0321','19000101140000');
select * from date_zero;
-- 0000-00-00 00:00:00 | 0000-00-00 | 838:59:59 | 0000 | 0000-00-00 00:00:00
```

Solution:

Replace zero date values with "1753 January 01" date.

Another method is to use a string or number data type to store zero dates.

Issue: Zeros in Year, Day, or Month

In MySQL, you can store dates where the year, day, or month is zero in a DATE or DATETIME column.

Example:

```
create table date_zeropart (d datetime null);
insert date_zeropart values
('00001215'),('20060015'),('20061200'),('20060000'),('20060229');
-- 0000-12-15 00:00:00
-- 2006-00-15 00:00:00
-- 2006-12-00 00:00:00
-- 2006-00-00 00:00:00
-- 0000-00-00 00:00:00
```

Solution:

Use string or number data types to store these values.

Issue: Invalid Dates

MySQL accepts invalid dates in ALLOW_INVALID_DATES SQL mode.

In ALLOW_INVALID_DATES mode, MySQL verifies only that the month is in the range from 0 to 12 and that the day is in the range from 0 to 31.

Example:

```
create table date_inval (d datetime null);
set sql_mode='ALLOW_INVALID_DATES';
insert date_inval values ('20061131');
insert date_inval values ('20061132');
set sql_mode='';
insert date_inval values ('20061131');
select * from date_inval;
-- 2006-11-31 00:00:00
-- 0000-00-00 00:00:00
-- 0000-00-00 00:00:00
```

Solution:

Use string or number data types to store these values.

Issue: Supported Range of the DATETIME Data Type

The supported range of the MySQL DATETIME data type is '0000-00-00 00:00:00' to '9999-12-31 23:59:59'.

Example:

create table datetime_range (d datetime);

insert datetime_range values ('0000-00-00 00:00:01');

insert datetime_range values ('0000-02-28 23:00:01');

insert datetime_range values ('0170-04-30 08:05:01');

insert datetime_range values ('9999-12-31 23:59:59');

Solution:

Use string or number data types to store these values or the **datetime2, datetimeoffset** data type.

Issue: MySQL YEAR, DATE, TIME Data Types

MySQL supports the YEAR data type, which is not present in SQL Server. The TIME data type differs in ranges.

The range of the YEAR data type is 1901 to 2155. Invalid YEAR values are converted to 0000.

TIME values may range from '-838:59:59' to '838:59:59'. By default, values that lie outside the TIME range but are otherwise valid are clipped to the closest endpoint of the range. For example, '-850:00:00' and '850:00:00' are converted to '-838:59:59' and '838:59:59'. Invalid TIME values are converted to '00:00:00'. In SQL Server, the **time** data type ranges 00:00:00.0000000 through 23:59:59.9999999.

Example:

```
create table time_range (t time);
insert time_range values ('2 01:30:54'); -- 49:30:54
```

```
insert time_range values ('201:03:45'); -- 201:03:45
insert time_range values ('900:42:14'); -- 838:59:59
insert time_range values ('-900:42:14'); -- -838:59:59
insert time_range values ('-1 05:15:20'); -- -29:15:20

create table year2 (y year(2));
insert year2 values (20),(1920),(80),(2080);
select * from year2; -- 20 20 80 80
create table year4 (y year);
insert year4 select y from year2;
select * from year4; -- 2020 1920 1980 2080
```

Solution:

String or number data types can be used to store these values.

Issue: TIMESTAMP and DATETIME Data Types

The TIMESTAMP data type is identical to the DATETIME data type and can have duplicate values.

Example:

```
create table table_ts (
id int auto_increment not null, d datetime null,
t timestamp not null default current_timestamp on update current_timestamp,
key(id));
insert table_ts (d) values (now()),(now()),(now()),(now()),(now());
select t, count(*) from table_ts group by t
-- 2006-12-22 19:20:38 | 5
```

Solution:

The TIMESTAMP type is easily emulated by using a trigger on INSERT and UPDATE that saves the current datetime in a datetime field.

String Types

Issue: VARCHAR and VARBINARY Maximum Size

The maximum size of the MySQL VARCHAR and VARBINARY data types is 65,535.

MySQL VARCHAR data longer than 65,535 characters is transformed into MEDIUMTEXT or LONGTEXT.

MySQL VARBINARY data longer than 65,535 is transformed into MEDIUMBLOB or LONGBLOB.

Example:

```
create table t_varchar (v varchar(65532));
describe t_varchar; -- v varchar(65532) ...

create table t_varchar (v varchar(65536));
describe t_varchar; -- v mediumtext ...

create table t_varbinary (v varbinary(65532));
describe t_varbinary; -- v varbinary(65532) ...

create table t_varbinary (v varbinary(65536));
describe t_varbinary; -- v mediumblob ...
```

Solution:

Use the **varchar(max)** and **varbinary(max)** data types to store character and binary data that is longer than 8,000 bytes.

Issue: BINARY Attribute for Fields with CHAR and VARCHAR Data Types

The MySQL BINARY attribute causes the binary collation for the column character set to be used. For example, CHAR(5) BINARY in MySQL is treated as CHAR(5) CHARACTER SET latin1 COLLATE latin1_bin, assuming that the default character set is latin1.

Example:

```
create table char_binary_ci (v varchar(8));
insert char_binary_ci values ('a'),('A'),('C'),('B');
select * from char_binary_ci order by v; -- 'a' 'A' 'B' 'C'

create table char_binary_cs (v varchar(8) binary);
insert char_binary_cs values ('a'),('A'),('C'),('B');
select * from char_binary_cs order by v; -- 'A' 'B' 'C' 'a'
```

Solution:

Use binary collation for these columns, for example:

```
create table char_binary_cs (v varchar(8) collate Latin1_General_BIN);
```

Issue: CHAR, VARCHAR, and TEXT Data Types Can Have Unicode Character Sets

MySQL has two Unicode character sets: ucs2 (UCS-2 Unicode) and utf8 (UTF-8 Unicode).

SQL Server contains the **nchar** and **nvarchar** data types to store Unicode data and uses the Unicode UCS-2 character set.

Example:

```
create table unicode_ucs2 (v varchar(10) character set ucs2);
create table unicode_utf8 (v varchar(10) character set utf8);
create table collation_cp (v varchar(10) charset cp1251);
insert unicode_ucs2 values ('Привет!');
insert unicode_ucs2 values ('您好您');
insert unicode_utf8 values ('Привет!');
insert unicode_utf8 values ('您好您');
insert collation_cp values ('Привет!');
select length(v) from unicode_ucs2; -- 14 6
select length(v) from unicode_utf8; -- 13 9
select length(v) from collation_cp; -- 7
select char_length(v) from unicode_ucs2; -- 7 3
select char_length(v) from unicode_utf8; -- 7 3
select char_length(v) from collation_cp; -- 7
```

Solution:

Convert CHAR, VARCHAR, and TEXT data types with Unicode character sets to the SQL Server **nchar** and **nvarchar** data types.

Issue: BLOB and TEXT Data Types Can Be Indexed

For indexes on BLOB and TEXT columns, you must specify an index prefix length in MySQL. Prefixes can be up to 1,000 bytes long (767 bytes for InnoDB tables).

Example:

```
create table blob_index (blob_col blob, index(blob_col(20)));
```

Solution:

You can use **varbinary(max)** and **varchar(max)** columns as included columns in index on another columns.

Issue: String Constants Can Contain ESCAPE Sequences

Each ESCAPE sequence in a string constant begins with a backslash (‘\’) in MySQL.

Example:

```
select 'This is \'Quoted string\'';
-- This is 'Quoted string'
```

Solution:

String constants must be changed by duplicating the single quote character:

```
SELECT 'This is ''Quoted string''';
```

ENUM and SET Data Types

Issue: ENUM (*enumerate*) Data Type

MySQL supports the ENUM (*enumerate*) data type. An ENUM is a string object with a value chosen from a list of allowed values that are enumerated explicitly in the column specification at table creation time.

If you insert an invalid value into an ENUM, the empty string is inserted instead as a special error value. If an ENUM column is declared to allow NULL, the NULL value is a valid value for the column, and the default value is NULL. If an ENUM column is declared NOT NULL, its default value is the first element of the list of allowed values.

Each enumeration value has a numeric index.

ENUM values are sorted according to the order in which the enumeration members were listed in the column specification.

Example:

```
create table table_enum (e enum ('a','b','c') not null);

insert into table_enum values ('a');
```

```
insert into table_enum values ('d');
insert into table_enum values ('a,c');
insert into table_enum values ('b,b,b');
insert into table_enum values ('b');
insert into table_enum values ();

select * from table_enum; -- 'a','','','','b','a';
select * from table_enum where e=1 -- 'a', 'a'

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

  create procedure proc_enum (e enum ('a','b','c'))
  begin
  if e!=''
     then select e;
     else select 'Invalid argument';
  end if;
  end

call proc_enum ('a'); -- 'a'
call proc_enum ('t'); -- 'Invalid argument'
```

Solution:

Try to emulate ENUM data type as a lookup table, such as in the following example code:

```
create table someenumtype (_id integer, _value varchar(max))
```

The original table will have a reference to this hash-table by _id.

You must add joins to all queries where the ENUM field value is used.

Issue: SET Data Type

MySQL supports the SET data type. A SET is a string object that can have zero or more values, each of which must be chosen from a list of allowed values specified when the table is created.

If you set a SET column to an unsupported value, the value is ignored.

MySQL stores SET values numerically, with the low-order bit of the stored value corresponding to the first set member.

Example:

```
CREATE TABLE table_set (s set('a','b','c') not null);

INSERT INTO table_set values ('a');
INSERT INTO table_set values ('d');
INSERT INTO table_set values ('a,c');
INSERT INTO table_set values ('b,b,b');
INSERT INTO table_set values ('b');
INSERT INTO table_set values ();


SELECT * FROM table_set; -- 'a','','a,c','b','b',''
SELECT * FROM table_set where s='a,c' -- 'a,c'

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

  CREATE PROCEDURE proc_set (p char(1), s set ('a','b','c'))
  BEGIN
  if find_in_set(p,s)>0
     then SELECT p;
     else SELECT 'Invalid argument';
 end if;
 END

call proc_set ('a','b,c,a'); -- 'a'
call proc_set ('a','b,c'); -- 'Invalid argument'
```

Solution:

The SET data type has dual nature—it is both an integer (up to 64 bits) and a string. Each bit in SET corresponds to a string description. The string representation of a SET value consists of appropriate strings, concatenated by commas.

Data manipulation is possible with both integer and string representations of SET.

Internally, SET is stored as integer; the size depends on the number of SET values (from 1 to 8 bytes). SQL Server emulation of the SET data type should be based on bigint, the largest possible integer data type.

To hold the string representation of bits in a SET value and to define all possible bits, create a “lookup table” as in the following example code:

```
CREATE TABLE lookup_set(
```

```
schemaname sysname not null,  -- schema name
tablename sysname not null, -- table name
colname sysname not null, -- column name
bitmask bigint not null, -- bitmask for value
position int not null, -- position in list
description varchar(512) not null,  -- character description of value
constraint pk_lookup_set
  primary key clustered (schemaname,tablename,colname,bitmask)
)
```

In addition, create a set of user-defined functions (UDFs) to support operations that use the SET data type.

UDF	Description
char_to_set	Converts a string representation to an integer representation. **Note**: char_to_set always acts as if strict mode is disabled or the IGNORE word in an INSERT or UPDATE clause is present.
set_to_char	Converts an integer representation to a char representation.
clean_set	Removes invalid bits from an integer representation. **Note**: Usage of clean_set may depend on strict mode or the IGNORE word in INSERT and UPDATE clauses.
check_set	Tests to see whether a given integer is a valid SET value.
find_in_set	Emulates the MySQL function FIND_IN_SET. Check the FIND_IN_SET function for a possible name clash with the second emulation of FIND_IN_SET.

Other Types

Issue: MySQL Spatial Data Types

MySQL has data types that correspond to OpenGIS classes (MySQL spatial data types).

Example:

```
create table spatial_type (g geometry, p point,
       l linestring, pg polygon, mp multipoint)
```

Solution:

Use spatial data types, which exist in SQL Server 2014.

Implicit Data Type Conversion

Issue: Implicit Data Type Conversion in MySQL

When a value of one type is used in a context that requires a value of another type, MySQL automatically performs extensive type conversion according to the kind of operation that is performed.

Examples:

```
select 100+'ABC' -- 100

select 100+'23ABC' -- 123

select concat('ABC',345,now(),50.4789) -- ABC3452006-11-08 19:00:0050.4789


drop table if exists table_date;
create table table_date
(d datetime, b smallint, i int(10) zerofill, f float, s varchar(64));
set @d=19980514;
insert into table_date values (@d, @d, @d, @d, @d);
select * from table_date;
-- 1998-05-14 00:00:00 32767 0019980514 1.99805e+007 19980514
set @d=now();
insert into table_date values (@d, @d, @d, @d, @d);
select * from table_date;
```

```
-- 2006-11-08 19:24:25 2006 0000002006 2006 2006-11-08 19:26:27
```

Solution:

No implicit conversions in SQL Server 2014, Hence use explicit conversion where needed.

Data Type Default Values

Issue: Implicit DEFAULT Values

If a column definition does not include an explicit DEFAULT value, MySQL determines the default value as follows:

- If the column can take NULL as a value, the column is defined with an explicit DEFAULT NULL clause.
- If the column cannot take NULL as a value, MySQL defines the column without an explicit DEFAULT clause. For data entry, if an INSERT or REPLACE statement does not include a value for a column, MySQL handles the column according to the SQL mode that is in effect at the time:
  - If strict SQL mode is not enabled, MySQL sets the column to the implicit default value for the column data type.
  - If strict mode is enabled, an error occurs for transactional tables and the statement is rolled back. For nontransactional tables, an error occurs, but if this happens for the second or subsequent row of a multiple-row statement, the preceding rows will have been inserted.

Example:

```
create table table_default (i int not null, d datetime not null,
       s varchar(64) not null, e enum ('a','b','c') not null, n int null);
insert table_default values ();
insert table_default values (default,default,'ABC',default,default);
select * from table_default;
-- 0 0000-00-00 00:00:00     a NULL
-- 0 0000-00-00 00:00:00 ABC a NULL


-- DEFAULT function example
create table table_defaultfunc (a int not null default 1,
                                b int not null default 2);
insert table_defaultfunc values ();
```

```
insert table_defaultfunc values (default(b),default(a));
select * from table_defaultfunc;
-- 1 2
-- 2 1
```

Solution:

No implicit conversions in SQL Server 2014, Hence use explicit defaults where needed.

MySQL Migration Issues

This section identifies problems that may occur during migration from MySQL 5 to SQL Server 2014 and suggests ways to handle them.

Operators

This section explains the differences between operators in MySQL and SQL Server 2014.

Comparison Operators

Issue: Comparison Operators in DML Statements

Unlike SQL Server, MySQL allows comparison operators in DML statements.

Example:

```
create table table_logic (id int not null,
             v varchar(64) not null, b int not null);

insert table_logic values (1,'1=2',1=2);
insert table_logic values (2,'1>2',1>2);
insert table_logic values (3,'1<2',1<2);
select * from table_logic; -- 1 1=2 0 | 2 1>2 0 | 3 1<2 1

select 1=2, 1>2, 1<2 from dual; -- 0 0 1

update table_logic set v='2=3', b=2=3 where id=3;
select * from table_logic; -- 3 2=3 0

update table_logic set v='NULL IS UNKNOWN', b=NULL IS UNKNOWN where id=3;
select * from table_logic; -- 3 NULL IS UNKNOWN 1

select @a is unknown, @a is null, @a is not null; -- 1 1 0

set @a=5-1=3+1
select @a -- 0

select 'a' in ('a','b','c'), 'a' not in ('a','b','c'); -- 1 0

select 1=2=0=5=0, 2>1=1<7=1<0 -- 1 0
```

Solution:

Emulate the comparison operators in DML statements by using the CASE function.

Issue: NULL-Safe Equal Comparison operator <=>

This MySQL operator performs an equality comparison like the = operator, but it returns 1 rather than NULL if both operands are NULL, and 0 rather than NULL if one operand is NULL. SQL Server does not have an identical operator.

Example:

```
select 1 <=> 1, null <=> null, 1 <=> null, @d <=> null;
-- 1 1 0 1
select 1 = 1, null = null, 1 = null, @d = null;
-- 1 NULL NULL NULL
```

Solution:

Use the CASE statement to perform an equality comparison.

Example:

```
select CASE
WHEN @a IS NULL AND @b IS NULL THEN 1
WHEN @a IS NULL OR @b IS NULL THEN 0
WHEN @a = @b THEN 1
ELSE 0 END
```

Issue: IS [NOT] *boolean_value* Comparison Operator

This MySQL operator tests a value against a Boolean value, where *boolean_value* can be TRUE, FALSE, or UNKNOWN. SQL Server does not have a similar operator.

Example:

```
create table table_is_int (i int);
insert table_is_int values (-1),(0),(1),(2),(3),(null);
select i is true from table_is_int; -- 1 0 1 1 1 0
select i is false from table_is_int; -- 0 1 0 0 0 0
select i is unknown from table_is_int; -- 0 0 0 0 0 1
select i=0 is true from table_is_int; -- 0 1 0 0 0 0
select * from table_is_int where (i is true) is false; -- 0 NULL
select 'A' is false, 'A' is true,
```

```
       '7A' is false, '7A' is true, now() is true;
-- 1 0 0 1 1
```

Solution:

Emulate this comparison operator by using the CASE function.

Issue: IS NULL Comparison Operator Extra Features

MySQL supports extra features for the IS NULL comparison operator.

In MySQL, you can find the row that contains the most recent AUTO_INCREMENT value by issuing a statement in the following form immediately after generating the value:

```
SELECT * FROM tbl_name WHERE auto_col IS NULL
```

For DATE and DATETIME columns that are declared as NOT NULL, you can find the special date '0000-00-00' by using a statement such as the following:

```
SELECT * FROM tbl_name WHERE date_column IS NULL
```

Example:

```
create table auto_inc (id int not null auto_increment,
                       v varchar(64) not null, key(id));
insert auto_inc (v) values ('ABC');
insert auto_inc (v) values ('DEF');
insert auto_inc (v) values ('GHI');
select * from auto_inc where id is null;
-- 3 'GHI'


create table auto_date (d datetime not null, v varchar(64) not null);
insert auto_date set v='A';
insert auto_date set v='B';
insert auto_date set v='C', d=now();
select * from auto_date where d is null;
-- 0000-00-00 00:00:00 A
-- 0000-00-00 00:00:00 B
```

Solution:

Use the SCOPE_IDENTITY() function to get the row that contains the most recent AUTO__INCREMENT value.

Example:

```
create table auto_inc (id int not null identity(1,1) primary key,
                       v varchar(64) not null);
insert auto_inc (v) values ('ABC');
insert auto_inc (v) values ('DEF');
insert auto_inc (v) values ('GHI');
select * from auto_inc where id = SCOPE_IDENTITY();
```

Bit Operators

Issue: Bit Shift Operators

MySQL has bit shift operators (<< and >>), which are not supported in SQL Server.

Example:

```
create procedure bit_shift (count int)
begin
declare v bigint; declare i int;
set v:=1; set i:=1;
while i<=count do
  set v := v << 1;
  select v, i;
  set i := i+1;
end while;
end;

call bit_shift (70);
-- 2                    1
-- 4611686018427387904  62
-- 9223372036854775807  63
-- 9223372036854775807  64
```

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Assignment Operators

Issue: Variable Assignment in SET Statements

In MySQL, variables can be assigned in a SET statement by using the := or = operators.

MySQL can also assign a value to a user variable in statements other than SET. In this case, the assignment operator must be := and not = because = is treated as a comparison operator in non-SET statements.

Unlike SQL Server, if a variable is assigned in a MySQL SELECT statement, the recordset is returned.

Example:

```
set @a=1; set @b:=2;
select @a, @b, @a=@b; -- 1 2 0
select @a:=@b; -- 2
select @a, @b, @a=@b; -- 2 2 1

create table assign_var (i int not null);
insert assign_var values (1),(2),(3);
select @i, @i:=i from assign_var order by i;
-- NULL 1
-- 1    2
-- 2    3
select @i:=i, @i from assign_var order by i;
-- 1 1
-- 2 2
-- 3 3
select @i; -- 3
```

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Variables

This section explains differences between variables in MySQL and SQL Server 2014.

Issue: Types of Variables Supported

MySQL supports two types of variables:

- User-defined variables @var_name
- Local variables (variables in stored routines)
  `DECLARE var_name[,...] type [DEFAULT value];`

In MySQL, user-defined variables do not use the DECLARE statement for initialization. They are initialized implicitly at the moment of first set (with SET or SELECT statement) or use. If you refer to a variable that has not been initialized with a SET or SELECT statement, the variable has a value of NULL and a type of string.

User-defined variables are connection-specific. SQL Server does not have connection-specific variables.

Example 1:

```
 create procedure proc ()
 begin
 select @a;
 end

set @a=100;
call proc2 ();
```

Example 2:

```
 create procedure proc (inout par_a int)
 begin
 set par_a=200;
 end

call proc2 (@b);
select @b;
```

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Issue: Case Sensitivity of User-Defined Variables

User-defined variables names are case-sensitive in versions earlier than MySQL 5.0 and not case-sensitive in MySQL 5.0 and later.

This should be considered when the SQL Server collation is chosen.

Solution:

Set case-sensitive collation on the server.

Issue: Default Value of Local Variables

MySQL local variables can have a default value.

Example:

```
create procedure ProcA ()
begin
declare var_a int default 100;
declare var_b varchar(8) default 'ABCDEFGHIJKLMN';
declare var_c datetime default now();
declare var_d int;
select var_a, var_b, var_c, var_d;
-- 100 ABCDEFGH 2006-11-08 15:05:04 (NULL)
end
```

Solution:

Use SQL Server DECLARE with a default declaration.

Example:

```
create procedure ProcA
as
begin
declare var_a int = 100;
declare var_b varchar(8) = 'ABCDEFGHIJKLMN';
declare var_c datetime2 = sysdatetime();
declare var_d int;
select var_a, var_b, var_c, var_d;
-- 100 ABCDEFGH 2006-11-08 15:05:04 (NULL)
end
```

Utility Statements

Issue: DELIMITER Command

The MySQL DELIMITER command allows the statement's delimiter to be changed.

Example:

```
create table table_a (id int);
select * from table_a;

delimiter //

select * from table_a//
drop table table_a//
```

Solution:

Change the delimiter to a standard semicolon ";".

Issue: HELP Command (HELP Syntax)

The HELP statement returns online information from the MySQL Reference manual.

```
HELP 'search_string'
```

Example:

```
HELP 'replace'


Syntax:
REPLACE(str,from_str,to_str)
Returns the string str with all occurrences of the string from_str
replaced by the string to_str. REPLACE() performs a case-sensitive
match when searching for from_str.
```

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Data Definition Statements

This section explains differences between the MySQL and SQL Server 2014 Data Definition Languages (DDLs) and provides common solutions for particular migration issues. It covers creation of tables, schemas, and views; conversion of temporary tables; and other DDL specific issues.

IF NOT EXISTS, IF EXISTS, OR REPLACE Clauses

Issue: IF NOT EXISTS Clause in CREATE DATABASE, CREATE TABLE, and CREATE EVENT Statements

The keywords IF NOT EXISTS prevent an error from occurring if the table (database, event) exists.

Note: If you use IF NOT EXISTS in a CREATE TABLE...SELECT statement, any rows selected by the SELECT part are inserted regardless of whether the table already exists.

MySQL example:

```
create database db_exists;
create database db_exists;
-- Error Code : 1007 Can't create database 'db_exists'; database exists
A: create database if not exists db_exists;
-- No Action


create table exists_a (i int not null);
create table exists_a (i int not null);
-- Error Code : 1050 Table 'exists_a' already exists
B: create table if not exists exists_a (i int not null, v varchar(64) null);
-- No Action
show create table exists_a; -- create table exists_a (i int(11) not null)
C: create table if not exists exists_a select now() as d from dual;
show create table exists_a; -- create table exists_a (i int(11) not null)
select * from exists_a; -- 2007
```

Solutions:

- CREATE DATABASE

  Replace the IF NOT EXISTS clause with the following condition:

  ```
  IF NOT EXISTS (SELECT name FROM sys.databases
      WHERE name = N'<db_name>')
  ```

```
BEGIN
<create_database_statement_without_if_not_exists>
END
```

- CREATE TABLE

  If the CREATE TABLE...SELECT syntax is not used, replace the IF NOT EXISTS clause with the one of the following conditions:

  - For permanent tables:

    ```
    IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id =
       OBJECT_ID(N'<table_name>') AND type in (N'U'))
    BEGIN
    <create_table_statement_without_if_not_exists>
    END
    ```

  - For temporary tables:

    ```
    IF NOT EXISTS (SELECT * FROM tempdb.sys.objects WHERE
    object_id =
       OBJECT_ID(N'tempdb..<#table_name>') AND type in (N'U'))
    BEGIN
    <create_#table_statement_without_if_not_exists>
    END
    ```

  If the CREATE TABLE...SELECT syntax is used, replace the IF NOT EXISTS clause with one of the following conditions:

  - For permanent tables:

    ```
    IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id =
       OBJECT_ID(N'<table_name>') AND type in (N'U'))
    BEGIN
    <create_table_statement_without_if_not_exists>
    <insert_select_statement>
    END
    ELSE BEGIN
    <insert_select_statement>
    END
    ```

  - For temporary tables:

    ```
    IF NOT EXISTS (SELECT * FROM tempdb.sys.objects WHERE
    object_id =
       OBJECT_ID(N'tempdb..<#table_name>') AND type in (N'U'))
    ```

```
BEGIN
<create_#table_statement_without_if_not_exists>
<insert_select_statement>
END
ELSE BEGIN
<insert_select_statement>
END
```

SQL Server example:

```
A: IF NOT EXISTS (SELECT name FROM sys.databases WHERE name = N'db_exists')
BEGIN
CREATE DATABASE db_exists
END

B: IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id =
   OBJECT_ID(N'exists_a') AND type in (N'U'))
BEGIN
CREATE TABLE exists_a (i int NOT NULL, v varchar(64) NULL)
END

C: IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id =
   OBJECT_ID(N'exists_a') AND type in (N'U'))
BEGIN
CREATE TABLE exists_a (d datetime NOT NULL default '1753-01-01 00:00:00')
INSERT exists_a SELECT getdate() AS d
END
ELSE BEGIN
INSERT exists_a SELECT getdate() AS d
END
```

Note: sys.objects system table is included as a view for backward compatibility. It is recommended that use the current SQL Server system views instead.

Issue: IF EXISTS Clause in DROP DATABASE, DROP TABLE, DROP VIEW, DROP EVENT, DROP PROCEDURE, and DROP FUNCTION Statements

IF EXISTS is used to prevent an error from occurring if the database (or table, view, event, procedure, or function) does not exist.

MySQL example:

```
A: drop database if exists db_exists;
   drop database db_exists;
   -- Error Code : 1008 Can't drop database 'db_exists'; database doesn't
exist
B: drop table if exists exists_a;
C: drop view if exists exists_view;
D: drop procedure if exists exists_proc;
E: drop function if exists exists_func;
```

Solutions:

- DROP DATABASE. Replace the IF EXISTS clause with the following condition:

```
IF EXISTS (SELECT name FROM sys.databases WHERE name = N'<db_name>')
BEGIN
<drop_database_statement_without_if_exists>
END
```

- DROP TABLE. Replace the IF EXISTS clause with the following condition:

```
IF EXISTS (SELECT * FROM tempdb.sys.objects WHERE object_id =
   OBJECT_ID(N'tempdb..<#table_name>') AND type in (N'U'))
BEGIN
<drop_#table_statement_without_if_exists>
END
ELSE BEGIN
IF EXISTS (SELECT * FROM sys.objects WHERE object_id =
   OBJECT_ID(N'<table_name>') AND type in (N'U'))
BEGIN
<drop_table_statement_without_if_exists>
END
```

- DROP TEMPORARY TABLE. Replace the IF EXISTS clause with the following condition:

```
IF EXISTS (SELECT * FROM tempdb.sys.objects WHERE object_id =
   OBJECT_ID(N'tempdb..<#table_name>') AND type in (N'U'))
BEGIN
<drop_#table_statement_without_if_exists>
END
```

- DROP VIEW. Replace the IF EXISTS clause with the following condition:

```
IF EXISTS (SELECT * FROM sys.views WHERE object_id =
   OBJECT_ID(N'<view_name>'))
BEGIN
<drop_view_statement_without_if_exists>
END
```

- DROP PROCEDURE. Replace the IF EXISTS clause with the following condition:

```
IF EXISTS (SELECT * FROM sys.objects WHERE object_id =
   OBJECT_ID(N'<proc_name>') AND type in (N'P', N'PC'))
BEGIN
<drop_procedure_statement_without_if_exists>
END
```

- DROP FUNCTION. Replace the IF EXISTS clause with the following condition:

```
IF EXISTS (SELECT * FROM sys.objects WHERE object_id =
   OBJECT_ID(N'<func_name>')
   AND type in (N'FN', N'IF', N'TF', N'FS', N'FT'))
BEGIN
<drop_function_statement_without_if_exists>
END
```

SQL Server example:

```
A: IF EXISTS (SELECT name FROM sys.databases WHERE name = N'db_exists')
BEGIN
DROP DATABASE db_exists
END

B: IF EXISTS (SELECT * FROM sys.objects WHERE object_id =
   OBJECT_ID(N'exists_a') AND type in (N'U'))
BEGIN
DROP TABLE exists_a
END

C: IF EXISTS (SELECT * FROM sys.views WHERE object_id =
   OBJECT_ID(N'exists_view'))
BEGIN
```

```
DROP VIEW exists_view
END

D: IF EXISTS (SELECT * FROM sys.objects WHERE object_id =
   OBJECT_ID(N'<exists_proc>') AND type in (N'P', N'PC'))
BEGIN
drop procedure exists_proc
END

E: IF EXISTS (SELECT * FROM sys.objects WHERE object_id =
   OBJECT_ID(N'exists_func')
   AND type in (N'FN', N'IF', N'TF', N'FS', N'FT'))
BEGIN
drop function exists_func;
END
```

Issue: OR REPLACE Clause in CREATE VIEW Statements

The MySQL OR REPLACE clause replaces an existing view.

MySQL example:

```
create or replace view repl_view as select now();
create or replace view repl_view as select version();
```

Solution:

Replace the OR REPLACE clause with the following condition:

```
IF EXISTS (SELECT * FROM sys.views WHERE object_id =
   OBJECT_ID(N'<view_name>'))
BEGIN
<drop_view_statement>
exec dbo.sp_executesql @statement
     = N'<create_view_statement_without_or_replace>'
END
ELSE BEGIN
exec dbo.sp_executesql @statement
     = N'<create_view_statement_without_or_replace>'
END
```

SQL Server example:

```
IF EXISTS (SELECT * FROM sys.views WHERE object_id =
   OBJECT_ID(N'repl_view'))
BEGIN
DROP VIEW repl_view
EXEC dbo.sp_executesql @statement
     = N'CREATE VIEW repl_view AS SELECT getdate() AS d'
END
ELSE BEGIN
EXEC dbo.sp_executesql @statement
     = N'CREATE VIEW repl_view AS SELECT getdate() as d'
END
```

Temporary Tables

Issue: MySQL TEMPORARY Tables Not Dropped When They Go Out Of Scope

SQL Server temporary tables created by the CREATE TEMPORARY TABLE statement are visible only to the current connection, and they are dropped automatically when the connection is closed. However, MySQL temporary tables are not dropped when they go out of scope.

In SQL Server:

- A local temporary table created in a stored procedure is dropped automatically when the stored procedure is finished. The table can be referenced by any nested stored procedures executed by the stored procedure that created the table. The table cannot be referenced by the process that called the stored procedure that created the table.
- A local temporary table created within a stored procedure or trigger can have the same name as a temporary table that was created before the stored procedure or trigger is called. However, if a query references a temporary table and two temporary tables with the same name exist at that time, there is no way to indicate which table the query should be resolved against. Nested stored procedures can also create temporary tables with the same name as a temporary table that was created by the stored procedure that called it. However, for modifications to resolve to the table that was created in the nested procedure, the table must have the same structure and the same column names as the table that was created in the calling procedure.

MySQL example 1:

```
create procedure proctemptable ()
begin
create temporary table table_temp (d datetime);
insert table_temp values (now());
end

call proctemptable();
select * from table_temp; -- 2006-11-20 11:18:58
call proctemptable(); -- Error Code : 1050 Table 'table_temp' already exists
```

MySQL example 2:

```
create procedure test2 ()
begin
create temporary table t (x int);
insert into t values (2);
select x as test2col from t;
```

```
end

create procedure test1 ()
begin
create temporary table t (x int);
insert into t values (1);
select x as test1col from t;
call test2 ();
end

call test1 ();
-- test1col = 1 (1 row)
-- ERROR 1050 (42S01): Table 't' already exists
```

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Issue: TEMPORARY Keyword in CREATE TABLE and DROP TABLE Statements

In MySQL, the TEMPORARY keyword in a CREATE TABLE statement is used to create a temporary table.

In MySQL, the TEMPORARY keyword in a DROP TABLE statement is used to drop a temporary table.

MySQL example:

```
A: create temporary table temp_table_a (a int not null);
B: create temporary table if not exists atest.temp_table_b (b int not null);

C: drop temporary table temp_table_a;
D: drop temporary table if exists atest.temp_table_b;
```

Solution:

Replace the TEMPORARY keyword with a single pound sign before the table name (#). Omit the database name.

SQL Server example:

```
A: CREATE TABLE #temp_table_a (a int NOT NULL)
B: IF NOT EXISTS (SELECT * FROM tempdb.sys.objects WHERE object_id =
   OBJECT_ID(N'tempdb..#temp_table_b') AND type in (N'U'))
BEGIN
CREATE TABLE #temp_table_b (b int NOT NULL)
END

C: DROP TABLE #temp_table_a;
D: IF EXISTS (SELECT * FROM tempdb.sys.objects WHERE object_id =
   OBJECT_ID(N'tempdb..#temp_table_b') AND type in (N'U'))
BEGIN
DROP TABLE #temp_table_b;
END
```

Issue: Temporary Tables with the Same Name as Nontemporary Tables

In MySQL, if a temporary table is created with same name as an existing nontemporary table, the existing nontemporary table is hidden until the temporary table is dropped.

MySQL example 1:

```
create table permanent_temp (v varchar(4) not null, d datetime not null);
insert permanent_temp values ('ABCD',now());
select * from permanent_temp; -- 'ABCD' '2007-02-08 16:19:40'
create temporary table permanent_temp (i int not null);
insert permanent_temp values (1);
select * from permanent_temp; -- 1
drop table permanent_temp; -- drop temporary table
select * from permanent_temp; -- 'ABCD' '2007-02-08 16:19:40'
drop table permanent_temp; -- drop permanent table
```

MySQL example 2:

```
create table permanent_temp (i int not null);
insert permanent_temp values (0);
select * from permanent_temp; -- 0
create temporary table permanent_temp (i int not null);
insert permanent_temp values (1);
select * from permanent_temp; -- 1
drop temporary table permanent_temp; -- drop temporary table
```

```
select * from permanent_temp; -- 0
drop temporary table permanent_temp;
-- Error Code : 1051 Unknown table   'permanent_temp'
drop table permanent_temp; -- drop permanent table
```

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Issue: Temporary Tables in Functions

In MySQL, temporary tables can be used in functions. SQL Server does not allow this.

MySQL example:

```
create function temp_func_sum () returns double
begin
declare s double;
select sum(i) into s from temp_func;
return s;
end

session 1:
create temporary table temp_func (i int not null);
insert into temp_func values (1),(2),(3),(4);
select temp_func_sum(); -- 10

session 2:
create temporary table temp_func (i numeric(19,9) not null);
insert into temp_func values (1.1000),(1.0100),(1.0010),(1.0001);
select temp_func_sum(); -- 4.1111

session 3:
create temporary table temp_func (i varchar(4) not null);
insert into temp_func values ('ABCD'),('5A'),('3.14P'),('1')
select temp_func_sum(); -- 9.14
```

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

SCHEMA Keyword in DATABASES Statements

Issue: SCHEMA Keyword in DATABASE Statements

In DATABASE statements (ALTER, CREATE, DROP, and RENAME) the SCHEMA keyword can be used as synonym of DATABASE keyword.

Solution:

Replace the SCHEMA keyword with the DATABASE keyword.

CHARACTER SET and COLLATE Clauses in DDL Statements

Issue: CHARACTER SET and COLLATE Clauses in DDL Statements

In MySQL terminology, a *character set* is a set of symbols and encodings, and a *collation* is a set of rules for comparing characters in a character set. The term *collation* in SQL Server combines both of these meanings.

MySQL example:

```
create database db_a character set latin1 collate latin1_swedish_ci;
```

Solution:

Convert the MySQL CHARACTER SET and COLLATE clauses to the SQL Server COLLATE clause.

Issue: CONVERT TO CHARACTER SET and [DEFAULT] CHARACTER SET Clauses in ALTER TABLE Statements

Unlike MySQL, SQL Server does not support collation changes at the table level.

Solution:

Change collation at the column level.

Note: Collations can be specified at the server, database, column, expression, and identifier levels only.If collation is specified, the default collation for the database is the default collation for the server instance.

CREATE INDEX Statement

Issue: NULL Values in UNIQUE Indexes

A MySQL UNIQUE index allows multiple NULL values for columns that can contain NULL.

MySQL example:

```
create table tabindex_b (i int null);
create unique index idx_tabindex_b on tabindex_b (i);
insert tabindex_b values (1);
insert tabindex_b values (2);
insert tabindex_b values (3);
insert tabindex_b values (1);
-- Duplicate entry '1' for key 1
insert tabindex_b values (null);
-- 1 row(s) affected
insert tabindex_b values (null);
-- 1 row(s) affected
```

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Issue: FULLTEXT Indexes

MySQL can create FULLTEXT indexes on tables that do not have a unique key index.

MySQL can create more than one FULLTEXT index on a table.

The WITH PARSER option can be used only with FULLTEXT indexes. It associates a parser plug-in with the index if full-text indexing and searching operations need special handling.

MySQL example:

```
create table tabindex_full (i int not null, t varchar(2048) null)
engine = myisam;
create unique index unique_tabindex_full on tabindex_full (i);
create fulltext index full_tabindex_full on tabindex_full (t);
```

Solution:

FULLTEXT indexes can be converted if all of the following conditions are met:

- Only one FULLTEXT index exists for the table.
- The table has a valid index to enforce a full-text search key.
- The FULLTEXT index does not use a parser plug-in.

FULLTEXT indexes not supported in the current version of Azure SQL DB.

SQL Server example:

```
CREATE TABLE tabindex_full (i int NOT NULL, t varchar(2048) NULL)
CREATE UNIQUE INDEX unique_tabindex_full ON tabindex_full (i)
CREATE FULLTEXT INDEX ON tabindex_full (t) key index unique_tabindex_full
```

Issue: Index Prefix Length

In MySQL, for CHAR, VARCHAR, BINARY, and VARBINARY columns, indexes can be created that use only the leading part of column values, by using col_name(length) syntax to specify an index prefix length. BLOB and TEXT columns also can be indexed, but a prefix length must be given. Prefix lengths are given in characters for non-binary string types and in bytes for binary string types. That is, index entries consist of the first length characters of each column value for CHAR, VARCHAR, and TEXT columns, and the first length bytes of each column value for BINARY, VARBINARY, and BLOB columns.

MySQL example:

```
create table tabindex_text (t text not null);
create unique index unique_tabindex_text on tabindex_text (t(4));
insert tabindex_text values ('ABCDEFG');
insert tabindex_text values ('ABCEFGD');
insert tabindex_text values ('ABKLMND');
insert tabindex_text values ('ABCKLMN');
insert tabindex_text values ('ABCDLMN');
-- Duplicate entry 'ABCD' for key 1
create table tabindex_blob (b blob null);
create unique index unique_tabindex_blob on tabindex_blob (b(4));
insert tabindex_blob values (0x01020304050607);
insert tabindex_blob values (0x01020310111204);
insert tabindex_blob values (0x01020304101112);
-- Duplicate entry '    ' for key 1
```

Solution:

Add the computed columns, which emulate the index prefix length functionality, to the table. Create the index on the computed columns.

SQL Server example:

```
CREATE TABLE tabindex_text (t varchar(max) NOT NULL,
                            t_comp AS convert(varchar(4),t))
CREATE UNIQUE CLUSTERED INDEX unique_tabindex_text ON tabindex_text (t_comp)
INSERT tabindex_text VALUES ('ABCDEFG')
INSERT tabindex_text VALUES ('ABCEFGD')
INSERT tabindex_text VALUES ('ABKLMND')
INSERT tabindex_text VALUES ('ABCKLMN')
INSERT tabindex_text VALUES ('ABCDLMN')
-- Cannot insert duplicate key row ...
CREATE TABLE tabindex_blob (b varbinary(max) NULL,
                            b_comp AS   convert(varbinary(4),b));
CREATE UNIQUE CLUSTERED INDEX unique_tabindex_blob ON tabindex_blob (b_comp);
INSERT tabindex_blob VALUES (0x01020304050607);
INSERT tabindex_blob VALUES (0x01020310111204);
INSERT tabindex_blob VALUES (0x01020304101112);
-- Cannot insert duplicate key row ...
```

CREATE TABLE Statement

Issue: CONSTRAINT Names

MySQL allows you to omit constraint names. It also allows duplicate constraint names.

MySQL example:

```
create table tab_constr (
id int not null, d  datetime,
fk int not null,
constraint primary key (id),
constraint unique (d),
constraint foreign key (fk) references tab_a (i)
);

create table tab_constr_dub (
id int not null, d  datetime,
constraint key_tab_constr_dub primary key (id),
constraint key_tab_constr_dub unique (d)
);
```

Solution:

Generate valid and unique constraint names.

SQL Server example:

```
CREATE TABLE tab_constr (
id int NOT NULL, d  datetime,
fk int NOT NULL,
CONSTRAINT pk_tab_constr primary key (id),
CONSTRAINT uq_tab_constr unique (d),
CONSTRAINT fk_tab_constr foreign key (fk) references tab_a (i)
);

CREATE TABLE tab_constr_dub (
id int NOT NULL, d  datetime,
CONSTRAINT pk_tab_constr_dub primary key (id),
CONSTRAINT uq_tab_constr_dub unique (d)
);
```

Issue: Index Definitions

In MySQL, indexes can be defined in the body of a table declaration.

MySQL example:

```
create table tab_index (
i int not null,
n int not null,
d datetime null,
v varchar(2048) not null,
primary key (i),
index idx_tab_index (n),
key (d),
fulltext index ft_tab_index (v)) engine = myisam;
```

Solution:

1. Remove index declarations from table declarations. Convert them into separate CREATE INDEX statements.sq
2. Generate valid and unique index names.
3. Replace the KEY keyword with the INDEX keyword.

FULLTEXT indexes not supported in the current version of Azure SQL DB.

SQL Server example:

```
CREATE TABLE tab_index (
i int not null,
n int NOT NULL,
d datetime NULL,
v varchar(2048) not NULL,
PRIMARY KEY (i))

CREATE INDEX idx_tab_index ON tab_index (n)
CREATE INDEX key_tab_index ON tab_index (d)
CREATE FULLTEXT INDEX ON tab_index (v) key index pk__tab_index__72e607db
```

Issue: FOREIGN KEY Constraint Indexes

Unlike SQL Server, in MySQL, for FOREIGN KEY constraints the index is created automatically.

Solution:

Create the index manually if needed.

Issue: RESTRICT Keyword in Reference Options

MySQL supports the RESTRICT keyword in reference options.

Solution:

Replace the RESTRICT keyword in reference options with the NO ACTION keyword.

Issue: KEY Keyword in Column Definitions

MySQL supports the KEY keyword in a column definition.

Solution:

Replace the KEY keyword in a column definition with the PRIMARY KEY keyword.

Issue: AUTO_INCREMENT Column Option and AUTO_INCREMENT Table Option

MySQL supports the AUTO_INCREMENT column option and AUTO_INCREMENT table option.

MySQL example:

```
create table auto_a (
i int not null auto_increment primary key,
d datetime null
)
auto_increment = 1000;

insert auto_a values (null,now()),(null,now()),(null,now());
select * from auto_a;
-- 1000 2009-02-16 17:41:43
-- 1001 2009-02-16 17:41:43
-- 1002 2009-02-16 17:41:43
```

Solution:

The MySQL AUTO_INCREMENT column option should be replaced by using the SQL Server IDENTITY column property. The MySQL AUTO_INCREMENT table option value should be converted to SQL Server as a seed parameter of the IDENTITY property. Also in some cases you can use CREATE SEQUENCE statement to emulate of AUTO_INCREMENT columns.

SQL Server example:

```
CREATE TABLE auto_a (
i int NOT NULL identity(1000,1) PRIMARY KEY,
d datetime NULL
)

INSERT auto_a VALUES (getdate())
INSERT auto_a VALUES (getdate())
INSERT auto_a VALUES (getdate())
```

Issue: PARTITION BY clause in Table Definitions

MySQL supports the partiting of a table on separate parts using different ways (HASH, KEY, RANGE and LIST).

Solution:

Create a partiting table using CREATE PARTITION FUNCTION, CREATE PARTITION SCHEME statements and ON partition_scheme_name (partition_column_name) clause of CREATE TABLE statement.

Issue: MERGE Tables

MERGE tables (storage engine) are a collection of identical MyISAM tables that can be used as one.

MySQL example:

```
create table merge_a (a int not null) engine=myisam ;
create table merge_b (b int not null) engine=myisam;

create table merge_m (m int not null)
engine=merge union=(merge_a,merge_b);

insert merge_a values (1),(2),(3);
insert merge_b values (4),(5),(6),(7);
```

```
select * from merge_a; -- 1 2 3
select * from merge_b; -- 4 5 6 7
select * from merge_m; -- 1 2 3 4 5 6 7

update merge_m set m=m+10 where m % 2 = 0;

select * from merge_a; -- 1 12 3
select * from merge_b; -- 14 5 16 7
```

Solution:

You can use partitioned views or tables.

Issue: CREATE TABLE...SELECT Syntax

MySQL allows you to create one table from another by adding a SELECT statement at the end of the CREATE TABLE statement.

MySQL creates new columns for all elements in the SELECT list that have a unique name.

CREATE TABLE...SELECT adds the result data into a new table.

MySQL example:

```
create table sel_a select 1 as id from dual;
show create table sel_a;
-- create table sel_a (id bigint not null)
select * from sel_a; -- 1

create table sel_b (id bigint not null) select 2 as id from dual;
show create table sel_b;
-- create table sel_b (id bigint not null)
select * from sel_b; -- 2

create table sel_c (id bigint not null) select 'ABC' as val, 1000 as id
from dual;
show create table sel_c;
-- create table sel_c (val varchar(3) not null, id bigint not null)
select * from sel_c; -- ABC 1000

create table sel_d (id bigint not null) select 'ABC' as val, 1000
```

```
from dual;
show create table sel_d;
-- create table sel_d (id bigint not null, val varchar(3) not null,
   `1000` bigint not null)
select * from sel_d; -- 0 ABC 1000
```

Solution:

To add new columns, the CREATE TABLE table definition must be changed manually.

Issue: CREATE TABLE...LIKE Syntax

MySQL supports LIKE to create an empty table based on the definition of another table, including any column attributes and indexes defined in the original table.

MySQL example:

```
create table like_a (
i int not null auto_increment primary key,
d datetime not null unique,
v varchar(1024) null,
fulltext (v)
) engine = myisam;

create table like_b like like_a;

show create table like_b;
/*
create table `like_b` (
  `i` int not null auto_increment,
  `d` datetime not null,
  `v` varchar(1024) collate latin1_general_ci default null,
  primary key  (`i`),
  unique key `d` (`d`),
  fulltext key `v` (`v`)
) engine=myisam default charset=latin1 collate=latin1_general_ci
*/
```

Solution:

Create such tables manually.

Issue: Foreign Key References to Other Databases

MySQL allows the creation of foreign key references to other databases.

Solution:

Use triggers instead of foreign keys, or organize data by schemas instead of databases.

ALTER TABLE Statement

Issue: IGNORE Keyword

The IGNORE extension controls how ALTER TABLE works if there are duplicates on unique keys in the new table or if warnings occur when strict mode is enabled. If IGNORE is not specified, the copy is stopped and rolled back if duplicate-key errors occur. If IGNORE is specified, only the first row in rows that have duplicates on a unique key is used. The other conflicting rows are deleted.

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Issue: FIRST | AFTER Keywords

In MySQL, to add a column at a specific position within a table row, the FIRST or AFTER *col_name* clause is used. The default is to add the column last. You can also use the FIRST and AFTER clauses in CHANGE or MODIFY column operations.

Solution:

Drop table and re-create it with a specific column position.

Issue: ALTER Clause

ALTER...SET DEFAULT specifies a new default value for a column. ALTER...DROP DEFAULT removes the old default value.

MySQL example:

```
create table alter_def (i int not null);
```

```
insert alter_def values ();
alter table alter_def alter i set default 99;
insert alter_def values ();
alter table alter_def alter i drop default;
insert alter_def values ();
select * from alter_def; -- 0 99 0
```

Solution:

Convert ALTER...SET DEFAULT and ALTER...DROP DEFAULT clauses to ADD/DROP DEFAULT constraint clauses.

SQL Server example:

```
CREATE TABLE alter_def (i int not null primary key);
INSERT alter_def default values; -- Cannot insert the value NULL into 'i'
ALTER TABLE alter_def add constraint i_def default 99 for i;
INSERT alter_def default values;
ALTER TABLE alter_def DROP CONSTRAINT i_def;
INSERT alter_def default values; -- Cannot insert the value NULL into 'i'
SELECT * FROM alter_def; -- 99
```

Issue: CHANGE Clause

MySQL supports a CHANGE clause in ALTER TABLE statements.

MySQL example:

```
create table change_a (i int not null);
alter table change_a change i i varchar(64) null;
show create table change_a;
alter table change_a change i d datetime not null;
show create table change_a;
```

Solution:

The CHANGE clause is converted to a SQL Server ALTER COLUMN clause. If CHANGE renames a column, use an additional call to **sp_rename**.

SQL Server example:

```
CREATE TABLE change_a (id int NOT NULL identity(1, 1) primary key,
```

```
i int not null);
ALTER TABLE change_a ALTER COLUMN i varchar(64) NULL;
sp_help 'change_a';
ALTER TABLE change_a ALTER COLUMN i datetime NOT NULL;
exec sp_rename 'change_a.i', 'd', 'COLUMN';
sp_help 'change_a';
```

Issue: MODIFY Clause

MySQL supports the MODIFY clause in ALTER TABLE statements.

MySQL example:

```
create table modify_a (i int not null);
alter table modify_a modify i varchar(64) null;
show create table modify_a;
alter table modify_a modify i datetime not null;
show create table modify_a;
```

Solution:

Convert the MODIFY clause to an ALTER COLUMN clause.

SQL Server example:

```
CREATE TABLE modify_a (id int NOT NULL identity(1, 1) primary key,
i int not null);
ALTER TABLE modify_a ALTER COLUMN i varchar(64) NULL;
sp_help 'modify_a';
ALTER TABLE modify_a ALTER COLUMN i datetime NOT NULL;
sp_help 'modify_a';
```

Issue: DROP [COLUMN] *col_name* Clause

In MySQL, ALTER TABLE can contain a DROP clause, which has the same meaning as DROP COLUMN.

Solution:

Add any missing COLUMN keywords.

Issue: DROP PRIMARY KEY Clause

MySQL supports the DROP PRIMARY KEY clause in ALTER TABLE statements.

MySQL example:

```
create table alter_c (id int not null, v varchar(64) not null);
alter table alter_c add constraint pk_alter_c primary key (id);
alter table alter_c drop primary key;
```

Solution:

The DROP PRIMARY KEY clause should be replaced with a DROP CONSTRAINT pk_constraint_name clause.

SQL Server example:

```
CREATE TABLE alter_c (id int not null, v varchar(64) NOT NULL)
ALTER TABLE alter_c ADD CONSTRAINT pk_alter_c primary key (id)
ALTER TABLE alter_c DROP CONSTRAINT pk_alter_c
```

Issue: DROP {INDEX|KEY} *index_name* Clause

MySQL supports the DROP INDEX clause in ALTER TABLE statements.

MySQL example:

```
create table alter_i (
i int not null,
n int not null,
primary key (i),
index idx_tab_index (n));
alter table alter_i drop index idx_tab_index;
```

Solution:

DROP INDEX clauses in ALTER TABLE statements should be converted to separate DROP INDEX statements.

SQL Server example:

```
CREATE TABLE alter_i (
```

```
i int not null,
n int NOT NULL,
PRIMARY KEY (i))
CREATE INDEX idx_tab_index ON alter_i (n)
DROP INDEX idx_tab_index ON alter_i
```

Issue: DROP FOREIGN KEY Clause

MySQL supports the DROP FOREIGN KEY clause in ALTER TABLE statements.

MySQL example:

```
create table alter_f (f_id int not null, c_id int not null);
alter table alter_f add constraint fk_alter_f foreign key (c_id)
                        references alter_c (id);
alter table alter_f drop foreign key fk_alter_f;
```

Solution:

Replace the DROP FOREIGN KEY clause with a DROP CONSTRAINT *fk_constraint_name* clause.

SQL Server example:

```
CREATE TABLE alter_f (f_id int not null, c_id int NOT NULL)
ALTER TABLE alter_f ADD CONSTRAINT fk_alter_f foreign key (c_id)
                        references alter_c (id)
ALTER TABLE alter_f DROP CONSTRAINT fk_alter_f
```

Issue: DISABLE KEYS and ENABLE KEYS Clauses

The ALTER TABLE...DISABLE KEYS clauses tell MySQL to stop updating nonunique indexes for a MyISAM table. You then use ALTER TABLE...ENABLE KEYS to re-create missing indexes. MySQL does this with a special algorithm that is much faster than inserting keys one by one, so disabling keys before performing bulk insert operations should speed up the operation significantly.

Solution:

Disable nonclustered indexes with the ALTER INDEX command.

Issue: RENAME Clause

MySQL supports a RENAME clause in ALTER TABLE statements.

MySQL example:

```
create table rename_a (i int not null);
insert rename_a values (1);
select * from rename_a; -- 1
alter table rename_a rename to rename_b;
select * from rename_a; -- Table 'rename_a' doesn't exist
select * from rename_b; -- 1
```

Solution:

Convert the RENAME clause to a separate **sp_rename** call.

SQL Server example:

```
CREATE TABLE rename_a (i int not null primary key);
INSERT rename_a values (1);
SELECT * FROM rename_a; -- 1
EXEC sp_rename 'rename_a', 'rename_b';
SELECT * FROM rename_a; -- Invalid object name 'rename_a'
SELECT * FROM rename_b; -- 1
```

Issue: ORDER BY Clause

In MySQL, the ORDER BY clause in an ALTER TABLE statement enables you to create a new table with the rows in a specific order.

Solution:

Use clustered indexes and ordered queries to order data.

RENAME DATABASE Statement

Issue: RENAME DATABASE Statements

MySQL supports the RENAME DATABASE statement.

Solution:

Convert RENAME DATABASE statements to a **sp_renamedb** call.

RENAME TABLE Statement

Issue: RENAME TABLE Statements

The MySQL RENAME TABLE statement renames one or more tables or views.

MySQL example:

```
create table rename_a (i int not null);
insert rename_a values (1);
create table rename_b (d datetime not null);
insert rename_b values (now());
rename tables rename_a to rename_c, rename_b to rename_a,
              rename_c to rename_b;
select * from rename_a; -- 2007-02-20 15:03:37
select * from rename_b; -- 1
select * from rename_c; -- Table 'ATest.rename_c' doesn't exist

create view rename_view_a as select 'view_string' as vs;
rename table rename_view_a to rename_view_b;
select * from rename_view_a; -- Table 'ATest.rename_view_a' doesn't exist
select * from rename_view_b; -- 'view_string'
```

Solution:

Convert each RENAME TABLE operation into a separate **sp_rename** call.

SQL Server example:

```
CREATE TABLE rename_a (i int not null primary key);
INSERT rename_a values (1);
CREATE TABLE rename_b (d datetime NOT NULL primary key);
INSERT rename_b values (getdate());
EXEC sp_rename 'rename_a', 'rename_c'
EXEC sp_rename 'rename_b', 'rename_a'
EXEC sp_rename 'rename_c', 'rename_b'
SELECT * FROM rename_a; -- 2007-02-20 15:06:48.967
SELECT * FROM rename_b; -- 1
SELECT * FROM rename_c; -- Invalid object name 'rename_c'

CREATE VIEW rename_view_a AS SELECT 'view_string' AS vs;
EXEC sp_rename 'rename_view_a', 'rename_view_b';
```

```
SELECT * FROM rename_view_a; -- Invalid object name 'rename_view_a'
SELECT * FROM rename_view_b; -- 'view_string'
```

Issue: Moving Tables Between Databases with the RENAME TABLE Statement

The MySQL RENAME TABLE statement can be used to move a table from one database to another.

MySQL example:

```
create table world.rename_table (v varchar(8) null);
insert world.rename_table values ('ABC');
rename table world.rename_table to sakila.rename_table;
select * from world.rename_table;
-- Table 'world.rename_table' doesn't exist
select * from sakila.rename_table; -- 'ABC'
```

Solution:

Move tables manually using CREATE TABLE, INSERT INTO, and DROP TABLE commands.

CREATE VIEW, ALTER VIEW, and DROP VIEW Statements

Issue: Database Name Prefix at View Name

Unlike MySQL, in SQL Server CREATE/ALTER/DROP VIEW does not allow the specification of the database name as a prefix to the object name.

MySQL example:

```
create view sakila.view_a as select 'ABCDE' as s;
select * from sakila.view_a; -- ABCDE
drop view sakila.view_a;
```

Solution:

Remove such prefixes and issue a USE command to set the active database.

In Azure SQL DB the *database* parameter can only refer to the current database. The USE statement does not switch between databases. To change databases, you must directly connect to the database.

SQL Server example:

```
USE sakila;
GO
CREATE VIEW view_a AS SELECT 'ABCDE' AS s;
GO
USE MASTER;
GO
SELECT * FROM sakila.view_a; -- ABCDE
GO
USE sakila;
GO;
DROP VIEW view_a;
```

Issue: LOCAL Keyword in WITH CHECK OPTION Clause

In a WITH CHECK OPTION clause for an updatable view, the LOCAL and CASCADED keywords determine the scope of check testing when the view is defined in terms of another view. The LOCAL keyword restricts CHECK OPTION to only the view that is being defined. The CASCADED keyword causes the checks for underlying views to be evaluated as well. When neither keyword is given, the default is CASCADED.

MySQL Example:

```
create table t1 (a int);
create view v1 as select * from t1 where a < 2
with check option;
create view v2 as select * from v1 where a > 0
with local check option;
create view v3 as select * from v1 where a > 0
with cascaded check option;

insert into v1 values (2); -- CHECK OPTION failed 'ATest.v1'
insert into v2 values (2);
insert into v3 values (2); -- CHECK OPTION failed 'ATest.v3'
select * from t1; -- 2
```

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Issue: Unnamed Columns in View Select List

MySQL automatically generates names for unnamed columns in a view's select list.

MySQL example:

```
create table table_name (a int not null, b int not null);
insert table_name values (1,2);
create view view_name as
select a, b, a+b, a*b, now() from table_name;
select * from view_name where `a+b`=3;
```

Solution:

Generate column names based on the MySQL select list.

SQL Server example:

```
CREATE TABLE table_name (a int NOT NULL primary key, b int NOT NULL);
```

```
INSERT table_name values (1,2);
CREATE VIEW view_name as
SELECT a, b, a+b AS [a+b], a*b AS [a*b], getdate() AS [now()]
FROM table_name;
SELECT * FROM view_name where [a+b]=3;
```

CREATE EVENT, ALTER EVENT, DROP EVENT Statements

Issue: MySQL Events

MySQL *events* are tasks that run according to a schedule. When you create an event, you are creating a named database object containing one or more SQL statements to be executed at one or more regular intervals, beginning and ending at a specific date and time.

Solution:

Use SQL Server Agent jobs.

SQL Azure does not support SQL Server Agent or jobs.

CREATE/ALTER/DROP PROCEDURE/FUNCTION Statements

Issue: Database Name Prefix at Procedure/Function Name

Unlike MySQL, in SQL Server, the CREATE/ALTER/DROP PROCEDURE/FUNCTION statements do not allow the specification of the database name as a prefix to the object name.

MySQL example:

```
create function sakila.func_drop () returns float
begin
declare s float;
set s:=3.14;
return s;
end

drop function sakila.func_drop;
```

Solution:

Remove such prefixes and issue a USE command to set the active database.

In Azure SQL DB the *database* parameter can only refer to the current database. The USE statement does not switch between databases. To change databases, you must directly connect to the database.

Issue: SQL SECURITY Characteristic

In MySQL, the SQL SECURITY characteristic can be used to specify whether the routine should be executed by using the permissions of the user who creates the routine or the user who invokes it. The default value is DEFINER.

MySQL example:

```
root user:
create table table_access (i int not null);
insert table_access values (1),(2),(3),(4),(5);

create procedure proc_access ()
sql security definer
begin
select * from table_access;
end

grant execute on ATest.* to abc;

abc user:
select * from ATest.table_access;
-- SELECT command denied to user 'abc'
call ATest.proc_access();
-- 1 2 3 4 5

root user:
drop procedure proc_access;
create procedure proc_access ()
sql security invoker
begin
select * from table_access;
end

abc user:
select * from ATest.table_access;
```

```
-- SELECT command denied to user 'abc
call ATest.proc_access();
-- SELECT command denied to user 'abc'
```

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Issue: Routine Parameter Names

Unlike SQL Server, MySQL does not require an at sign (@) prefix for the names of a routine's variables.

MySQL example:

```
create function func_pi (p int) returns float
begin
declare s float;
set s:=p*pi();
return s;
end
```

Solution:

Convert parameter names in routines to SQL Server procedure or function parameter names and use an at sign (@) as the first character.

SQL Server example:

```
CREATE FUNCTION func_pi (@p int) returns float
BEGIN
DECLARE @s float;
SET @s=@p*pi();
RETURN @s;
END
```

Issue: INOUT Procedure Parameters

MySQL supports INOUT specification for procedure parameters.

MySQL example:

```
create procedure proc_inout (a int, inout b int)
begin
set b=b+a;
end

set @b=0;
call proc_out(7,@b);
select @b; -- 7
call proc_out(7,@b);
select @b; -- 14
```

Solution:

Convert MySQL INOUT procedure parameters to SQL Server OUT procedure parameters.

SQL Server example:

```
CREATE PROCEDURE proc_inout (@a int, @b int out)
AS
BEGIN
SET @b=@b+@a;
END

DECLARE @b int
SET @b=0;
EXEC proc_inout 7, @b out
SELECT @b; -- 7
EXEC proc_inout 7, @b out
SELECT @b; -- 14
```

Issue: OUT Procedure Parameters

MySQL supports an OUT specification for procedure parameters.

An OUT parameter passes a value from the procedure back to the caller. Its initial value is NULL within the procedure, and its value is visible to the caller when the procedure returns.

MySQL example:

```
create procedure proc_out (a int, out b int)
```

```
 begin
 set b=isnull(b)+a;
 end

set @b=99;
call proc_out(7,@b);
select @b; -- 8
call proc_out(7,@b);
select @b; -- 8
```

Solution:

Set variable values to NULL at the beginning of the procedure.

Example:

```
create procedure proc_out @a int,  @b int output
as
 begin
 set @b = null;
 set @b = isnull(@b, 1) + @a;
 end
```

Issue: AS Keyword Before Procedure Body

MySQL does not use the AS keyword before the procedure body.

MySQL example:

```
create procedure proc_as (a int, b int)
begin
select a+b;
end
```

Solution:

Add the AS keyword before the procedure body.

SQL Server example:

```
CREATE PROCEDURE proc_as (@a int, @b int)
AS
BEGIN
SELECT @a+@b;
END
```

Data Manipulation Statements

This section describes differences between the MySQL and SQL Server 2014 Data Manipulation Languages (DMLs) and provides common solutions for typical migration issues.

It also covers SELECT, INSERT, UPDATE, and DELETE statements, discusses the conversion of a number of MySQL specific clauses such as LIMIT, and explains the differences between join syntax.

LIMIT Clause

Issue: LIMIT Clause in SELECT Statements

The MySQL SELECT result can be limited by using the LIMIT clause.

If a LIMIT clause has two arguments, the first argument specifies the offset of the first row to return, and the second specifies the maximum number of rows to return.

If a LIMIT clause has one argument, the value specifies the number of rows to return from the beginning of the result set.

MySQL example:

```
select id, data from t1 order by id limit 3, 2;
```

Solution:

Emulate a LIMIT clause with one argument by using the TOP clause of a SELECT statement.

A LIMIT clause with two arguments can be emulated by using a subquery with the ROW_NUMBER() function or new OFFSET-FETCH clause.

SQL Server example:

```
SELECT id, data
FROM
( SELECT id, data, row_number () over (order by id) - 1 as rn
  FROM t1 ) rn_subquery
WHERE rn between 3 and (3+2)-1
ORDER BY id

SELECT id, data
FROM t1
ORDER BY id
OFFSET 3 ROWS
  FETCH NEXT 2 ROWS ONLY
```

Issue: LIMIT and ORDER BY Clauses in a Single-Table DELETE Statement

The LIMIT clause places a limit on the number of rows that can be deleted.

If the DELETE statement includes an ORDER BY clause, the rows are deleted in the order specified in the clause. This is really useful only in conjunction with LIMIT.

MySQL example:

```
delete from t3 where data<ascii(id) limit 2;
```

Solution:

Emulate the LIMIT clause by using the TOP clause in a DELETE statement.

Review the logic of DELETE statements with LIMIT and ORDER BY clauses.

SQL Server example:

```
DELETE TOP (2) from t3 WHERE data<ascii(id);
```

Issue: LIMIT and ORDER BY Clauses in Single-Table UPDATE Statements

You can use LIMIT row_count to restrict the scope of an UPDATE statement.

If an UPDATE statement includes an ORDER BY clause, rows are updated in the order specified by the clause.

MySQL example:

```
create table t_upd (id int not null primary key, v varchar(8) not null);
insert t_upd values (1,'A'),(2,'B'),(3,'C');
update t_upd set id=id+1; -- Error Code : 1062 Duplicate entry '2' for key 1
update t_upd set id=id+1 order by id desc; -- 3 row(s)affected
update t_upd set v=concat(v,'+',v) limit 1; -- 1 row(s)affected
```

Solution:

Emulate the LIMIT clause by using the TOP clause of the UPDATE statement.

The logic of UPDATE statements with LIMIT and ORDER BY clauses should be reviewed.

SQL Server example:

```
CREATE TABLE t_upd (id int NOT NULL PRIMARY KEY, v varchar(8) NOT NULL)
```

```
INSERT t_upd values (1,'A')
INSERT t_upd values (2,'B')
INSERT t_upd values (3,'C')
UPDATE t_upd SET id=id+1 -- 3 row(s) affected
UPDATE TOP (1) t_upd SET v=v+'+'+v -- 1 row(s) affected
```

DELETE Statement

Issue: Multiple-Table DELETE

For multiple-table syntax, DELETE deletes from each *tbl_name* the rows that satisfy the conditions.

MySQL example:

```
create table del_a (id int not null, v varchar(8) not null);
create table del_b (id int not null, v varchar(8) not null);
insert del_a values (1,'A'),(2,'B'),(3,'C');
insert del_b values (1,'C'),(2,'B'),(3,'A');

-- deletes row with id=3 from both tables
delete a, b from del_a a, del_b b where a.id=b.id and a.v>b.v;
-- deletes row with id=3 from table del_a only
delete a from del_a a, del_b b where a.id=b.id and a.v>b.v;
```

Solution:

Multiple-table DELETE can be emulated by using separate DELETE statements for each table in aggregate with a table variable (or temporary table) for saving intermediate data.

SQL Server example:

```
CREATE TABLE del_a (id int NOT NULL primary key, v varchar(8) NOT NULL)
CREATE TABLE del_b (id int NOT NULL primary key, v varchar(8) NOT NULL)
INSERT del_a SELECT 1,'A' UNION SELECT 2,'B' UNION SELECT 3,'C'
INSERT del_b SELECT 1,'C' UNION SELECT 2,'B' UNION SELECT 3,'A'

-- deletes row with id=3 from both tables
DECLARE @temp table (id int, v varchar(8))

DELETE a OUTPUT deleted.id, deleted.v INTO @temp
FROM del_a a, del_b b WHERE a.id=b.id AND a.v>b.v
```

```
DELETE b
FROM @temp a, del_b b WHERE a.id=b.id AND a.v>b.v

-- deletes row with id=3 from table del_a only
DELETE a FROM del_a a, del_b b WHERE a.id=b.id AND a.v>b.v
```

UPDATE Statement

Issue: Multiple-Table UPDATE

For multiple-table syntax, UPDATE updates rows in each table named in *table_references* that satisfy the conditions.

MySQL example:

```
create table upd_a (id int not null, v varchar(32) not null);
create table upd_b (id int not null, v varchar(32) not null);
insert upd_a values (1,'A'),(2,'B'),(3,'C');
insert upd_b values (1,'C'),(2,'B'),(3,'A');

update upd_b b, upd_a a
set a.v=concat('Z',b.v,'+',b.v),
    b.v=concat('Z',a.v,'+',a.v)
where a.id=b.id and a.v>=b.v;
```

Solution:

Multiple-table UPDATEs can be emulated by using separate UPDATE statements for each table in aggregate with a table variable (or temporary table) for saving intermediate data.

SQL Server example:

```
CREATE TABLE upd_a (id int NOT NULL primary key, v varchar(32) NOT NULL)
CREATE TABLE upd_b (id int NOT NULL primary key, v varchar(32) NOT NULL)
INSERT upd_a SELECT 1,'A' UNION SELECT 2,'B' UNION SELECT 3,'C'
insert upd_b SELECT 1,'C' UNION SELECT 2,'B' UNION SELECT 3,'A'

DECLARE @temp table (id int, v_old varchar(32), v_new varchar(32))

UPDATE b
SET b.v='Z'+a.v+'+'+a.v
OUTPUT deleted.id, deleted.v, inserted.v into @temp
FROM upd_b b, upd_a a
```

```
WHERE a.id=b.id AND a.v>=b.v;

UPDATE a
SET a.v='Z'+b.v_new+'+'+b.v_new
FROM @temp b, upd_a a
WHERE a.id=b.id AND a.v>=b.v_old;
```

INSERT Statement

Issue: INSERT...ON DUPLICATE KEY UPDATE Syntax

In MySQL, if you specify ON DUPLICATE KEY UPDATE, and a row is inserted that would cause a duplicate value in a UNIQUE index or PRIMARY KEY, an UPDATE of the old row is performed.

Also, you can use the VALUES(col_name) function in the UPDATE clause to refer to column values from the INSERT portion of the INSERT...UPDATE statement.

MySQL example:

```
create table ins_upd
       (a int not null primary key, b int not null, c int not null);
insert ins_upd values (1,2,3),(2,3,4),(1,20,30)
       on duplicate key update c=values(c);
select * from ins_upd; -- 1 2 30, 2 3 4
```

Solution:

Check for the presence of added keys in the index and execute an INSERT statement for new keys and an UPDATE statement for existing keys.

Replace the VALUES function with its value.

SQL Server example:

```
CREATE TABLE ins_upd
       (a int NOT NULL PRIMARY KEY, b int NOT NULL, c int NOT NULL)

DECLARE c CURSOR FORWARD_ONLY STATIC READ_ONLY
   FOR SELECT 1,2,3 UNION ALL SELECT 2,3,4 UNION ALL SELECT 1,20,30

DECLARE @a int, @b int, @c int
```

```
OPEN c
FETCH c INTO @a, @b, @c
WHILE @@fetch_status=0
BEGIN
IF NOT EXISTS (SELECT TOP 1 0 from ins_upd WHERE a=@a)
   INSERT ins_upd values (@a, @b, @c)
   ELSE UPDATE ins_upd set c=@c where a=@a
FETCH c INTO @a, @b, @c
END
CLOSE c DEALLOCATE c

SELECT * FROM ins_upd; -- 1 2 30, 2 3 4
```

Issue: INSERT and AUTO_INCREMENT Fields

The MySQL INSERT statement allows inserting into AUTO_INCREMENT fields.

MySQL example:

```
create table table_autoinc
(id int not null auto_increment, v varchar(32) null, key (id));

insert into table_autoinc (v) values ('Value_1');
insert into table_autoinc (v) values ('Value_2');
insert into table_autoinc (v) values ('Value_3');
insert into table_autoinc (id, v) values (40,'Value_4');
insert into table_autoinc (id, v) values (50,'Value_5');
insert into table_autoinc (id, v) values (40,'Value_4_2');
insert into table_autoinc (id, v) values (40,'Value_4_3');
insert into table_autoinc (v) values ('Value_6');
insert into table_autoinc (v) values ('Value_7');

select * from table_autoinc; -- id: 1,2,3,40,50,40,40,51,52
```

Solution:

This functionality can be emulated by using SET IDENTITY_INSERT statements.

SQL Server example:

```
CREATE TABLE table_autoinc
(id int NOT NULL identity(1, 1) primary key, v varchar(32) NULL)
create index idx_table_autoinc ON table_autoinc (id)

INSERT INTO table_autoinc (v) VALUES ('Value_1');
INSERT INTO table_autoinc (v) VALUES ('Value_2');
INSERT INTO table_autoinc (v) VALUES ('Value_3');
SET identity_insert table_autoinc ON
INSERT INTO table_autoinc (id, v) VALUES (40,'Value_4');
INSERT INTO table_autoinc (id, v) VALUES (50,'Value_5');
INSERT INTO table_autoinc (id, v) VALUES (40,'Value_4_2');
INSERT INTO table_autoinc (id, v) VALUES (40,'Value_4_3');
SET identity_insert table_autoinc OFF
INSERT INTO table_autoinc (v) VALUES ('Value_6');
INSERT INTO table_autoinc (v) VALUES ('Value_7');

SELECT * FROM table_autoinc; -- id: 1,2,3,40,50,40,40,51,52
```

Issue: Expression in INSERT VALUES Syntax

An expression in INSERT VALUES syntax can refer to any column that was set earlier in a value list.

MySQL example:

```
create table ins_expr (a float not null, b float not null);
insert ins_expr values (sin(4),abs(a));
```

Solution:

Replace the column reference with its value.

SQL Server example:

```
CREATE TABLE ins_expr (a float NOT NULL primary key, b float NOT NULL);
INSERT ins_expr VALUES (sin(4),abs(sin(4)));
```

Issue: Inserting Explicit and Implicit Default Values

If both the column list and the VALUES list are empty, INSERT creates a row with each column set to its default value: INSERT INTO tbl_name () VALUES();

MySQL example:

```
create table ins_def (a int null, b int not null, c int default 1);

insert ins_def values (); -- insert ins_def () values ();
select * from ins_def; -- NULL 0 1
```

Solution:

Use the INSERT … DEFAULT VALUES statement.

SQL Server example:

```
CREATE TABLE ins_def (id int NOT NULL identity(1, 1) primary key, a int NULL,
b int NOT NULL DEFAULT 0, c int DEFAULT 1);

INSERT ins_def DEFAULT VALUES; -- INSERT ins_def () VALUES ();
SELECT * FROM ins_def; -- 1 NULL 0 1
```

REPLACE Statement

Issue: REPLACE Statement

A MySQL REPLACE statement works exactly like an INSERT statement, except that if an old row in the table has the same value as a new row for a PRIMARY KEY or a UNIQUE index, the old row is deleted before the new row is inserted.

MySQL example:

```
create table tab_repl
       (a int not null primary key, b int not null, c int not null);
replace tab_repl values (1,2,3),(2,3,4),(1,20,30);
select * from tab_repl; -- 1 20 30, 2 3 4
```

Solution:

Check for the presence of added keys in the index and execute an INSERT statement for new keys and an UPDATE statement for existing keys, or delete existing keys before inserting.

SQL Server example:

```
CREATE TABLE tab_repl
       (a int NOT NULL PRIMARY KEY, b int NOT NULL, c int NOT NULL)

DECLARE c CURSOR FORWARD_ONLY STATIC READ_ONLY
   FOR SELECT 1,2,3 UNION ALL SELECT 2,3,4 UNION ALL SELECT 1,20,30

DECLARE @a int, @b int, @c int

OPEN c
FETCH c INTO @a, @b, @c
WHILE @@fetch_status=0
BEGIN
 DELETE FROM tab_repl WHERE a=@a;
 INSERT tab_repl values (@a, @b, @c)
 FETCH c INTO @a, @b, @c
END
CLOSE c DEALLOCATE c

SELECT * FROM tab_repl; -- 1 20 30, 2 3 4
```

SELECT Statement

Issue: DISTINCTROW Keyword

MySQL supports the DISTINCTROW keyword in SELECT statements.

Solution:

Replace the MySQL DISTINCTROW keyword with the SQL Server DISTINCT keyword.

Issue: References in ORDER BY Clause

An ORDER BY clause can refer to fields missed in the SELECT list when the DISTINCT clause is used.

MySQL example:

```
create table tab_dist (a int, b int);
insert tab_dist values (1,30),(2,20),(3,40),(4,20),(5,10);
select distinct b from tab_dist order by a desc; -- 10 40 20 30
```

Solution:

This behavior can be emulated using the GROUP BY clause.

SQL Server example:

```
CREATE TABLE tab_dist (id int NOT NULL identity(1, 1) primary key,
a int,
b int);
INSERT tab_dist VALUES (1,30),(2,20),(3,40),(4,20),(5,10);
SELECT b FROM tab_dist GROUP BY b ORDER BY min(a) DESC;
```

Issue: DUAL Table

In MySQL, you can specify DUAL as a dummy table name in situations where no tables are referenced.

MySQL example:

```
select curdate();
select curdate() from dual;
```

```
select count(*) from dual; -- 1
```

Solution:

Usually you can ignore FROM DUAL clauses. However, a SELECT COUNT(*) FROM DUAL statement should be converted to SELECT 1.

Issue: SELECT...FROM...PROCEDURE Syntax

In MySQL, you can define a procedure in the Microsoft Visual C++® development system that can access and modify the data in a query before it is sent to the client.

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

SELECT…INTO and LOAD DATA INFILE Statements

Issue: SELECT…INTO *variable* statement

MySQL supports SELECT…INTO variable statements.

MySQL example:

```
select max(host), max(user) into @h, @u from mysql.user;
```

Solution:

Define variable assignment in the field list.

SQL Server example:

```
SELECT @h=max(host), @u=max([user]) FROM mysql.dbo.[user];
```

Issue: SELECT...INTO OUTFILE and LOAD DATA INFILE Statements

MySQL uses SELECT...INTO OUTFILE statements to write data from a table to a file and LOAD DATA INFILE statements to read the file back into a table.

MySQL example:

```
select * into outfile "d:\\in.out\\table_gh.dat" from gh;
delete from gh;
load data infile "d:/in.out/table_gh.dat" into table gh;
```

Solution:

The SQL Server **bcp** utility allows both writing tables to a file and loading the contents of a file into tables.

SQL Server example:

```
EXEC xp_cmdshell 'bcp "SELECT * FROM ATest.dbo.gh" queryout "d:\table_gh.dat"
-c -T'
DELETE FROM gh
EXEC xp_cmdshell 'bcp ATest.dbo.gh in "d:\table_gh.dat" -k -E -c -T'
```

GROUP BY, HAVING, and ORDER BY Clauses

Issue: Column References in GROUP BY and HAVING Clauses

In MySQL, columns from the SELECT list can be referenced in a GROUP BY or HAVING clause by using column aliases or by positions (only in GROUP BY).

MySQL example:

```
create table tab_alias (field_a int, field_b int);
insert tab_alias values (1,1),(1,2),(1,3),(2,1),(2,2);
select field_a as a, count(*) from tab_alias group by a order by a desc;
-- 2 2, 1 3
select field_b as b, count(*) from tab_alias group by 1 order by 1 desc;
-- 3 1, 2 2, 1 2
```

Solution:

Alias and column position references in GROUP BY and HAVING clauses should be changed with referenced fields.

SQL Server example:

```
CREATE TABLE tab_alias (id int NOT NULL identity(1, 1) primary key, field_a
int, field_b int)
INSERT tab_alias SELECT 1,1 UNION ALL SELECT 1,2 UNION ALL SELECT 1,3 UNION
ALL SELECT 2,1 UNION ALL SELECT 2,2
SELECT field_a AS a, count(*) FROM tab_alias GROUP BY field_a ORDER BY a DESC
SELECT field_b AS b, count(*) FROM tab_alias GROUP BY field_b ORDER BY 1 DESC
```

Issue: GROUP BY Sorting

In MySQL, if you use GROUP BY, output rows are sorted according to the GROUP BY columns as if you had an ORDER BY for the same columns.

MySQL extends the GROUP BY clause so that you can also specify ASC and DESC after columns named in the clause.

MySQL example:

```
select help_category_id, count(*)
  from mysql.help_topic group by help_category_id desc
```

Solution:

In SQL Server, add the ORDER BY clause for sorting.

Issue: ORDER BY NULL Syntax

To avoid the overhead of sorting that GROUP BY produces in MySQL, the ORDER BY NULL clause is used.

MySQL example:

```
select host, count(*) from mysql.user group by host order by null
```

Solution:

This syntax can be ignored.

Issue: SELECT and ORDER BY Clauses Can Have Fields Without Aggregation but Are Not Presented in GROUP BY

MySQL extends the use of GROUP BY to allow the selection of fields that are not mentioned in the GROUP BY clause.

A similar MySQL extension applies to the HAVING clause. The SQL standard does not allow the HAVING clause to name any column that is not found in the GROUP BY clause if it is not enclosed in an aggregate function. MySQL allows the use of these columns to simplify calculations.

This extension assumes that the nongrouped columns have the same group-wise values. Otherwise, the result is indeterminate.

MySQL example:

```
create table customer (custid int, name varchar(32));
insert customer values (1,'Customer_A');
insert customer values (2,'Customer_B');
insert customer values (3,'Customer_C');
create table order (custid int, payments numeric(19,2));
insert order values (1,50.80);
insert order values (1,140.84);
insert order values (2,32.80);

select order.custid, customer.name, max(payments)
from order, customer
where order.custid = customer.custid
group by order.custid;
```

Solution:

SQL Server does not support queries whose fields are present in the SELECT list or ORDER BY clause without aggregation, but are missing in GROUP BY clause. Modify the query by including these fields in the GROUP BY clause.

Issue: HAVING Clause Without GROUP BY Clause

In MySQL you can use a HAVING clause without a GROUP BY clause.

MySQL example:

```
create table tab_hav (class varchar(128), amount int, date datetime);
insert tab_hav values ('PRINTER',2,'20061215');
insert tab_hav values ('SCANNER',3,'20070123');
insert tab_hav values ('FAX',5,'20070918');
insert tab_hav values ('PRINTER',1,'20070921');
insert tab_hav values ('PHONE',4,'20070308');
```

```
insert tab_hav values ('SCANNER',2,'20070514');
insert tab_hav values ('PRINTER',3,'20071011');

select * from tab_hav having sum(amount)=20; -- 1 row
select * from tab_hav having max(amount)=20; -- 0 row
select * from tab_hav having max(amount)=5;  -- 1 row
```

Solution:

To emulate this functionality, convert the HAVING clause to a WHERE clause and use a subquery to calculate table aggregate functions.

JOINs

Issue: JOIN...USING Syntax

The USING (column_list) clause names a list of columns that must exist in both tables. If tables a and b both contain columns c1, c2, and c3, the following join compares corresponding columns from the two tables: a LEFT JOIN b USING (c1,c2,c3).

MySQL example:

```
create table tab_value
  (key_a char(8), key_b char(8), key_c char(8), value int);
create table tab_subvalue
  (key_a char(8), key_b char(8), key_c char(8), subvalue int);
insert tab_value values ('A','A','A',1),('B','D','E',2),('X','Y','Z',3);
insert tab_subvalue
  values ('A','A','A',100),('A','A','A',120),('B','D','M',200),
         ('X','Y','Z',318),('X','Y','Z',350);

select value, subvalue
from tab_value v join tab_subvalue sv using (key_a,key_b,key_c)
order by value, subvalue;

select value, subvalue
from tab_value v join tab_subvalue sv using (key_a,key_b)
order by value, subvalue;
```

Solution:

Replace the USING clause with ON and set the condition by all joined fields.

SQL Server example:

```
SELECT value, subvalue
FROM tab_value v
     JOIN tab_subvalue sv ON v.key_a=sv.key_a AND v.key_b=sv.key_b
                             AND v.key_c=sv.key_c
ORDER BY value, subvalue
```

```
SELECT value, subvalue
FROM tab_value v
     JOIN tab_subvalue sv ON v.key_a=sv.key_a AND v.key_b=sv.key_b
ORDER BY value, subvalue
```

Issue: CROSS JOIN and INNER JOIN

In MySQL, CROSS JOIN is the syntactic equivalent of INNER JOIN (they can replace each other).

MySQL example:

```
select value, subvalue
from tab_value v inner join tab_subvalue sv
order by value, subvalue;
```

```
select value, subvalue
from tab_value v cross join tab_subvalue sv on v.key_a=sv.key_a
order by value, subvalue;
```

Solution:

MySQL INNER join can be used without join conditions (ON …). In this case it works as CROSS JOIN.

MySQL CROSS join can be used with join conditions (ON …). In this case it works as INNER JOIN.

SQL Server example:

```
SELECT value, subvalue
FROM tab_value v CROSS JOIN tab_subvalue sv
```

```
ORDER BY value, subvalue

SELECT value, subvalue
FROM tab_value v INNER JOIN tab_subvalue sv ON v.key_a=sv.key_a
ORDER BY value, subvalue
```

Issue: STRAIGHT_JOIN

In MySQL, STRAIGHT_JOIN is identical to JOIN, except that the left table is always read before the right table. This can be used for those (few) cases in which the join optimizer puts the tables in the wrong order.

MySQL example:

```
select value, subvalue
from tab_value v straight_join tab_subvalue sv
order by value, subvalue;

select value, subvalue
from tab_value v straight_join tab_subvalue sv on v.key_a=sv.key_a
order by value, subvalue;
```

Solution:

MySQL STRAINT_JOIN is an optimization issue and can be replaced with a SQL Server INNER or CROSS join in most cases.

SQL Server example:

```
SELECT value, subvalue
FROM tab_value v CROSS JOIN tab_subvalue sv
ORDER BY value, subvalue

SELECT value, subvalue
FROM tab_value v INNER JOIN tab_subvalue sv ON v.key_a=sv.key_a
ORDER BY value, subvalue
```

Issue: NATURAL JOIN

The NATURAL [LEFT] JOIN of two tables is defined to be semantically equivalent to an INNER JOIN or a LEFT JOIN with a USING clause that names all columns that exist in both tables.

MySQL example:

```
select value, subvalue
from tab_value v natural join tab_subvalue sv
order by value, subvalue;
```

Solution:

If the joined tables have columns with the same names, convert the NATURAL join to an INNER join by these columns. Otherwise, convert a NATURAL join as a CROSS join.

SQL Server example:

```
SELECT value, subvalue
FROM tab_value v
     join tab_subvalue sv ON v.key_a=sv.key_a AND v.key_b=sv.key_b
                             and v.key_c=sv.key_c
ORDER BY value, subvalue
```

Subqueries

Issue: Row Subqueries

MySQL allows row subqueries. A row subquery is a subquery variant that returns a single row and can thus return more than one column value.

MySQL example:

```
A: select * from gh where (id, value) = row(1, 'A');

B: select * from gh
   where (id, value) = (select subid, value from gj where gj.id=gh.id);
```

Solution:

Rewrite MySQL statements that have row subqueries by using the logical AND operator and the EXISTS condition.

SQL Server example:

```
A: SELECT * FROM gh WHERE id = 1 AND value = 'A'

B: SELECT * FROM gh
   WHERE EXISTS (SELECT 1 FROM gj
                 WHERE gj.id=gh.id AND gj.subid=gh.id AND gj.value=gh.value)
```

Prepared Statements

Issue: Server-Side Prepared Statements

Starting from version 5.1 MySQL supports server-side prepared statements. The scope of a prepared statement is the client session within which it is created.

```
PREPARE stmt_name FROM preparable_stmt
EXECUTE stmt_name [USING @var_name [, @var_name] ...]
{DEALLOCATE | DROP} PREPARE stmt_name
```

MySQL example:

```
 create procedure ProcPrepare ()
 begin
 execute prep_stmt using @a, @b;
 end

prepare prep_stmt from 'select sqrt(pow(?,2) + pow(?,2)) as hypotenuse';
set @a = 3, @b = 4;
call procprepare();
deallocate prepare prep_stmt;
```

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

DO Command

Issue: DO Syntax

The MySQL DO command executes expressions but does not return any results. In most respects, DO is shorthand for SELECT *expr*, ..., but it has the advantage that it is slightly faster when you do not care about the result.

MySQL example 1:

```
select @a:=200; -- sets @a and returns 200
select @a; -- returns 200
do @a:=300; -- sets @a
select @a; -- returns 300
```

MySQL example 2:

```
create table TableDO (d int not null);

create function func_do (par_d int) returns int
begin
delete from TableDO where d=par_d;
return row_count();
end

insert TableDO values (1), (2), (3);

do func_do(2);
select d from TableDO; -- 1, 3

do @r:=func_do(1);
select @r, d from TableDO; -- 1 3
```

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

HANDLERs

Issue: HANDLER Interface

MySQL supports the HANDLER interface for reading table data.

```
HANDLER tbl_name OPEN [ AS alias ]
HANDLER tbl_name READ index_name { = | >= | <= | < } (value1,value2,...)
    [ WHERE where_condition ] [LIMIT ... ]
HANDLER tbl_name READ index_name { FIRST | NEXT | PREV | LAST }
    [ WHERE where_condition ] [LIMIT ... ]
HANDLER tbl_name READ { FIRST | NEXT }
```

```
    [ WHERE where_condition ] [LIMIT ... ]
HANDLER tbl_name CLOSE
```

The HANDLER statement provides direct access to table storage engine interfaces. It is available for MyISAM and InnoDB tables.

MySQL example:

```
HANDLER TableA OPEN;
HANDLER TableA READ FIRST LIMIT 100;
HANDLER TableA CLOSE;
```

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

MODIFIERs

Issue: Modifiers (LOW_PRIORITY, DELAYED, HIGH_PRIORITY, QUICK, IGNORE) in DML Statements

These MySQL modifiers enable you to apply something similar to an isolation level to each statement separately and to manage error raising.

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Transactional and Locking Statements

This section discusses the main differences between MySQL and SQL Server 2014 locking and transaction control statements—starting, committing and rolling back, table locking, working with isolation levels, and AUTOCOMMIT mode.

BEGIN TRANSACTION Statements

Issue: Different Begin Transaction Syntax

MySQL and SQL Server use different syntax to start a transaction.

Solution:

Replace MySQL START TRANSACTION statements in batches and routines and BEGIN/BEGIN WORK statements in batches by using the SQL Server BEGIN TRANSACTION statement.

Issue: Begin Transaction Statements Implicitly Commit Current Transaction

The MySQL begin transaction statement implicitly commits the current transaction.

Solution:

Add a commit transaction statement with check @@TRANCOUNT state before the begin transaction statement.

Issue: Statements That Cause an Implicit Commit

Each of the following MySQL statements (and any synonyms for them) implicitly ends a transaction, as if you had done a COMMIT before executing the statement:

ALTER FUNCTION, ALTER PROCEDURE, ALTER TABLE, BEGIN, CREATE DATABASE, CREATE FUNCTION, CREATE INDEX, CREATE PROCEDURE, CREATE TABLE, DROP DATABASE, DROP FUNCTION, DROP INDEX, DROP PROCEDURE, DROP TABLE, LOAD MASTER DATA, LOCK TABLES, LOAD DATA INFILE, RENAME TABLE, SET AUTOCOMMIT=1, START TRANSACTION, TRUNCATE TABLE, UNLOCK TABLES.

Solution:

Add a commit transaction statement with check @@TRANCOUNT state before statements that cause an implicit commit.

Issue: Statements That Cannot Be Rolled Back

Some MySQL statements cannot be rolled back. In general, these include DDL statements, such as those that create or drop databases, and those that create, drop, or alter tables or stored routines.

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Issue: Storage Engines That Do Not Support Transactions

Transactions do not influence operations with tables that are based on storage engines that do not support transactions.

MySQL example:

```
create table tran_x (i int not null) engine = innodb;
create table tran_y (i int not null) engine = myisam;

start transaction;
insert tran_x values (7);
insert tran_y values (7);
rollback;

select * from tran_x; --
select * from tran_y; -- 7
```

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

END TRANSACTION Statements

Issue: End Transaction Statement Without a Begin Transaction Statement

End transaction statements can be executed without prior begin transaction statements.

Solution:

Add IF (@@TRANCOUNT>0) condition before COMMIT/COMMIT WORK/ROLLBACK/ROLLBACK WORK statements.

Issue: CHAIN Clause

The MySQL AND CHAIN clause causes a new transaction to begin as soon as the current one ends. The new transaction has the same isolation level as the just-terminated transaction.

Solution:

Replace the MySQL CHAIN clause with a SQL Server BEGIN TRANSACTION statement after the END TRANSACTION statement.

Issue: RELEASE Clause

The RELEASE clause causes the server to disconnect the current client connection after terminating the current transaction.

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Named Transaction SAVEPOINT Statements

Issue: Different Savepoint Syntax

MySQL and SQL Server have different syntax for savepoint.

MySQL example:

```
create table tran_a (i int not null);
```

```
begin;
insert tran_a values (1);
start transaction;
insert tran_a values (2);
rollback and chain;
insert tran_a values (3);
savepoint spoint;
insert tran_a values (4);
rollback to savepoint spoint;
commit;

select * from tran_a; -- 1 3
```

Solution:

Replace MySQL SAVEPOINT savepoint_name statements with SQL Server SAVE TRANSACTION savepoint_name statements.

Replace MySQL ROLLBACK [WORK] TO SAVEPOINT savepoint_name statements with SQL Server ROLLBACK TRANSACTION savepoint_name statements.

SQL Server example:

```
CREATE TABLE tran_a (id int NOT NULL identity(1, 1) primary key, i int NOT
NULL)

BEGIN TRANSACTION
INSERT tran_a values (1)
IF (@@trancount>0) COMMIT
BEGIN TRANSACTION
INSERT tran_a values (2)
IF (@@trancount>0) ROLLBACK
BEGIN TRANSACTION
INSERT tran_a VALUES (3)
SAVE TRANSACTION spoint
INSERT tran_a values (4)
ROLLBACK TRANSACTION spoint
IF (@@trancount>0) COMMIT
```

```
SELECT * FROM tran_a -- 1 3
```

Issue: RELEASE SAVEPOINT Statement

The MySQL RELEASE SAVEPOINT statement removes the named savepoint from the set of savepoints of the current transaction.

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

SET AUTOCOMMIT Statements

Issue: SET AUTOCOMMIT Statement

MySQL supports the SET AUTOCOMMIT statement.

MySQL Example:

```
create table tran_auto (i int not null);

set autocommit = 1;
insert tran_auto values (10);
insert tran_auto values (20);
rollback;
set autocommit = 0;
insert tran_auto values (30);
insert tran_auto values (40);
rollback;
insert tran_auto values (50);
commit;

select * from tran_auto; -- 10 20 50
```

Solution:

Replace SET AUTOCOMMIT = 1 with SET IMPLICIT_TRANSACTIONS OFF.

Replace SET AUTOCOMMIT = 0 with SET IMPLICIT_TRANSACTIONS ON.

SQL Server example:

```
CREATE TABLE tran_auto (id int NOT NULL identity(1, 1) primary key, i int NOT
NULL)

SET IMPLICIT_TRANSACTIONS OFF
INSERT tran_auto VALUES (10)
INSERT tran_auto VALUES (20)
IF (@@trancount>0) ROLLBACK
SET IMPLICIT_TRANSACTIONS ON
INSERT tran_auto VALUES (30)
INSERT tran_auto VALUES (40)
IF (@@trancount>0) ROLLBACK;
INSERT tran_auto VALUES (50)
IF (@@trancount>0) COMMIT

SELECT i FROM tran_auto -- 10 20 50
```

LOCK TABLES and UNLOCK TABLES Statements

Issue: LOCK TABLES and UNLOCK TABLES Syntax

LOCK TABLES locks base tables (but not views) for the current thread.

UNLOCK TABLES releases any locks held by the current thread.

If a thread obtains a READ lock on a table, that thread (and all other threads) can only read from the table. If a thread obtains a WRITE lock on a table, only the thread holding the lock can write to or read the table. Other threads are blocked from writing to or reading the table until the lock is released.

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

SET TRANSACTION ISOLATION LEVEL Statement

Issue: MySQL Default Transaction Isolation Level

In MySQL the default transaction isolation level is REPEATABLE READ. In SQL Server the default transaction isolation level is READ COMMITTED.

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

XA Transaction Statements

Issue: XA Transactions

The MySQL XA transactions implementation is based on the X/Open CAE document Distributed Transaction Processing: The XA Specification
(http://www.opengroup.org/public/pubs/catalog/c193.htm).

The XA interface to a MySQL server consists of SQL statements that begin with the XA keyword.

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Database Administration Statements

This section discusses converting MySQL administrative statements, including account management and table maintenance.

Account Management Statements

Issue: CREATE USER, DROP USER, GRANT, RENAME USER, REVOKE, and SET PASSWORD Statements

MySQL and SQL Server have different syntax for account management statements.

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Table Maintenance Statements

Issue: ANALYZE TABLE, BACKUP TABLE, CHECK TABLE, CHECKSUM TABLE, OPTIMIZE TABLE, REPAIR TABLE, and RESTORE TABLE Statements

SQL Server does not have the identical statements.

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

SET Statement

Issue: Multiple Variable Assignments in SET Statements

In MySQL a SET statement can contain multiple variable assignments, separated by commas. A SQL Server SET statement can contain only one assignment to a variable.

MySQL example:

```
create procedure proc_set_var ()
begin
declare a, b int;
set a=10, b=20;
```

```
select a+b;
end

call proc_set_var () -- 30
```

Solution:

Convert each variable assignment to a separate SET statement or use a single SELECT statement.

SQL Server example:

```
 CREATE PROCEDURE proc_set_var
 AS
 BEGIN
 DECLARE @a int, @b int
 SET @a=10
 SET @b=20
 SELECT @a+@b
 END

EXEC proc_set_var -- 30
```

Issue: Server System Variables (Global Variables)

The MySQL server maintains many system variables that indicate how it is configured. Each system variable has a default value. System variables can be set at server startup by using options on the command line or in an option file. Most of them can be changed dynamically while the server is running by means of the SET statement, which enables you to modify the operation of the server without having to stop and restart it. You can refer to system variable values in expressions.

System variable values can be set globally at server startup by using options on the command line or in an option file.

Many system variables are dynamic and can be changed while the server runs by using the SET statement. To change a system variable with SET, refer to it as *var_name*, optionally preceded by a modifier:

- To indicate explicitly that a variable is a global variable, precede its name by GLOBAL or @@global.
- To indicate explicitly that a variable is a session variable, precede its name by SESSION, @@session., or @@ .

- LOCAL and @@local. are synonyms for SESSION and @@session.

If no modifier is present, SET changes the session variable.

A SET statement can contain multiple variable assignments, separated by commas.

If you change a session system variable, the value remains in effect until your session ends or until you change the variable to a different value. The change is not visible to other clients.

If you change a global system variable, the value is remembered and used for new connections until the server restarts. (To make a global system variable setting permanent, set it in an option file.)

To set a SESSION variable to the GLOBAL value or a GLOBAL value to the compiled-in MySQL default value, use the DEFAULT keyword.

See also: SQL Mode

MySQL example:

```
create table table_inc
(id int not null auto_increment, unique key (id), v varchar(8) null);

insert table_inc (v) values ('A');
insert table_inc (v) values ('B');
insert table_inc (v) values ('C');

set session auto_increment_increment = 7;

insert table_inc (v) values ('D');
insert table_inc (v) values ('E');

set session auto_increment_increment = default;

insert table_inc (v) values ('F');
insert table_inc (v) values ('G');

select id from table_inc; -- 1 2 3 8 15 16 17
```

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

SHOW Statement

Issue: SHOW Syntax

SHOW has many forms that provide information about databases, tables, columns, or status information about the server.

See also: DESCRIBE Syntax

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Issue: DESCRIBE Syntax

DESCRIBE provides information about the columns in a table. It is a shortcut for SHOW COLUMNS FROM. These statements also display information for views.

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Other Administrative Statements

Issue: CACHE INDEX, LOAD INDEX INTO CACHE, FLUSH, RESET, and KILL Statements

SQL Server does not have the identical statements.

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Stored Procedures and Functions (Routines)

This section discusses differences between the SQL procedural extension language in MySQL and Microsoft SQL Server. This includes the creation and calling of stored procedures and functions, working with local variables, cursors, and the control of flow statements.

CALL Statements

Issue: Syntax for Calling Procedures

MySQL uses the CALL statement to invoke a procedure.

MySQL supports expressions as call parameters.

MySQL example:

```
create procedure proc_case (s varchar(64),
       out s_low varchar(64), out s_up varchar(64))
begin
set s_low:=lower(s), s_up:=upper(s);
end

call proc_case (date_format(now(),'%D %M %Y'),@low,@up);
select @low, @up; -- 23rd october 2007, 23RD OCTOBER 2007
```

Solution:

Convert MySQL CALL statements to Transact-SQL EXEC statements. Expressions in call parameters should be calculated with temporary variables before the statements. Include the OUTPUT keyword for output parameters.

SQL Server example:

```
CREATE PROCEDURE proc_case (@s varchar(64),
       @s_low varchar(64) OUT, @s_up varchar(64) OUT) AS
BEGIN
SELECT @s_low=lower(@s), @s_up=upper(@s)
END

DECLARE @s varchar(64), @low varchar(64), @up varchar(64)
SET @s=convert(varchar(64),getdate(),106)
EXEC proc_case @s, @low OUTPUT, @up OUTPUT
SELECT @low, @up; -- 23 oct 2007, 23 OCT 2007
```

Compound Statements Block

Issue: Empty Compound Statements

The empty compound statement (BEGIN END) is valid in MySQL but not in Transact-SQL.

MySQL example:

```
create procedure empty_block(i int)
begin
select sin(i);
 begin
 end;
select cos(i);
end
```

Solution:

Ignore these statements.

Issue: Labeled Compound Statements

MySQL compound statements can be labeled. You cannot use end_label unless begin_label is also present. If both are present, they must have the same name.

MySQL example:

```
create procedure lab_comp()
begin
 s: begin
    select 'STEP 1'; -- displayed
    leave s;
    select 'STEP 2'; -- ignored
    end;
 select 'STEP 3'; -- displayed
end
```

Solution:

Emulate LEAVE behavior in a labeled compound statement by using the Transact-SQL GOTO statement.

SQL Server example:

```
CREATE PROCEDURE lab_comp as
BEGIN
    BEGIN
    SELECT 'STEP 1'; -- displayed
    GOTO s;
    SELECT 'STEP 2'; -- ignored
    END;
 s: SELECT 'STEP 3'; -- displayed
END
```

Local Variables

Issue: Declaring Variables of the Same Type

MySQL allows declaring several variables of one type in a single statement.

MySQL example:

```
declare x, y, z int;
```

Solution:

Declare the type of each variable in SQL Server.

SQL Server example:

```
DECLARE @x int, @y int, @z int
```

Issue: Scope of Local Variables

In MySQL, the scope of a local variable is within the BEGIN...END block in which it is declared. The variable can be referred to in blocks nested within the declaring block, except those blocks that declare a variable of the same name.

MySQL example:

```
create procedure var_scope()
begin
declare a, b int;
set a=5, b=7;
select a, b; -- 5 7
  begin
  declare a int;
  set a=9;
  select a, b; -- 9 7
  end;
select a, b; -- 5 7
end
```

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Issue: SQL Variable Names Can Be the Same As Column Names

If an SQL statement contains a reference to a column and a declared local variable with the same name, MySQL interprets the reference as the name of a variable.

MySQL example:

```
create procedure var_field()
begin
create temporary table if not exists vf (a int, b int);
insert vf values (1,1),(1,2),(1,3);
select a, b from vf; -- 1 1, 1 2, 1 3
  begin
  declare a int default 7;
  select a, b from vf; -- 7 1, 7 2, 7 3
  end;
end
```

Solution:

Interpret dual references during conversion as the name of a variable.

SQL Server example:

```
CREATE PROCEDURE var_field as
BEGIN
CREATE TABLE #vf (a int, b int)
INSERT #vf values (1,1)
INSERT #vf values (1,2)
INSERT #vf values (1,3)
SELECT a, b FROM #vf -- 1 1, 1 2, 1 3
  BEGIN
  DECLARE @a int
  SET @a=7
  SELECT @a, b FROM #vf -- 7 1, 7 2, 7 3
  END;
END
```

Conditions and Handlers

Issue: MySQL Condition Handling

MySQL manages conditions by defining handlers.

The DECLARE CONDITION statement specifies conditions that need specific handling.

The DECLARE HANDLER statement specifies handlers. Each handler handles one or more conditions. If one of these conditions occurs, the statement (compound statement) specified in the handler is executed.

MySQL example:

```
create table TableCondition_A (c_a int not null);
create table TableCondition_B (c_b int not null);

create procedure ProcCondition (in par_value int, inout par_null_error int)
begin
declare cond_a condition for sqlstate value '23000';
-- Error: 1048 SQLSTATE: 23000 (ER_BAD_NULL_ERROR)
-- Message: Column '%s' cannot be null
declare continue handler for cond_a
        begin set par_null_error=par_null_error+1; end;

set par_null_error=0;

insert TableCondition_A values (par_value);
insert TableCondition_B values (par_value);
end

call ProcCondition (null, @err);
select @err; -- 2

call ProcCondition (100, @err);
select @err; -- 0
```

Solution:

Use TRY…CATCH exception handling.

SQL Server example:

```
CREATE TABLE TableCondition_A (id int NOT NULL identity(1, 1) primary key,
c_a INT NOT NULL);
CREATE TABLE TableCondition_B (id int NOT NULL identity(1, 1) primary key,
c_b INT NOT NULL);
go
CREATE PROCEDURE ProcCondition  @par_value INT, @par_null_error INT OUTPUT
AS
-- Error: 1048 SQLSTATE: 23000 (ER_BAD_NULL_ERROR)
-- Message: Column '%s' cannot be null
SET @par_null_error = 0;

BEGIN TRY
INSERT TableCondition_A VALUES (@par_value);
END TRY
BEGIN CATCH
      IF ERROR_NUMBER() = 515
            SET @par_null_error += 1;
END CATCH

BEGIN TRY
INSERT TableCondition_B VALUES (@par_value);
END TRY
BEGIN CATCH
      IF ERROR_NUMBER() = 515
            SET @par_null_error += 1;
END CATCH
GO
DECLARE @err INT
EXEC ProcCondition null, @err OUTPUT;
SELECT @err; -- 2

EXEC ProcCondition 100, @err OUTPUT;
SELECT @err; -- 0
```

Cursors

Issue: "No Data" Cursor State

MySQL requires a handler for conditions with SQLSTATE value 02000 to detect "No Data" cursor state.

MySQL example:

```
create table t1 (id char(16), data int);
create table t2 (i int);
create table t3 (id char(16), data int);
insert t1 values ('A',65),('K',75),('Q',81),('S',83),('W',87);
insert t2 values (10),(100),(20),(200),(30);

create procedure curdemo()
begin
  declare done int default 0;
  declare a char(16);
  declare b, c int;
  declare cur1 cursor for select id, data from t1;
  declare cur2 cursor for select i from t2;
  declare continue handler for sqlstate '02000' set done = 1;

  open cur1;
  open cur2;

  repeat
    fetch cur1 into a, b;
    fetch cur2 into c;
    if not done then
       if b < c then
          insert into t3 values (a,b);
       else
          insert into t3 values (a,c);
       end if;
    end if;
  until done end repeat;

  close cur1;
  close cur2;
end

call curdemo();
select * from t3; -- A 10, K 75, Q 20, S 83, W 30
```

Solution:

Use the Transact-SQL @@FETCH_STATUS variable to detect the status of the last cursor FETCH statement.

SQL Server example:

```
CREATE PROCEDURE curdemo AS
BEGIN
  DECLARE @done int SET @done=0
  DECLARE @a char(16)
  DECLARE @b int, @c int
  DECLARE cur1 CURSOR FORWARD_ONLY STATIC READ_ONLY
          FOR SELECT id, data from t1;
  DECLARE cur2 CURSOR FORWARD_ONLY STATIC READ_ONLY
          FOR SELECT i FROM t2;

  OPEN cur1;
  OPEN cur2;

  WHILE @done=0
  BEGIN
    FETCH cur1 INTO @a, @b;
    IF @@fetch_status<>0 SET @done = 1
    FETCH cur2 INTO @c;
    IF @@fetch_status<>0 SET @done = 1
    IF @done<>1
       BEGIN
       IF @b < @c
          BEGIN INSERT INTO t3 VALUES (@a,@b); END
       ELSE
          BEGIN INSERT INTO t3 VALUES (@a,@c); END
       END
  END;

  CLOSE cur1 DEALLOCATE cur1
  CLOSE cur2 DEALLOCATE cur2
END
```

Flow Control Constructs

Issue: IF Statement

MySQL and SQL Server have different syntax for the IF statement.

MySQL example:

```
if (1>2)
   then select 'A'; select 'B';
   elseif (2>3) then select 'C'; select 'D';
   elseif (3>4) then select 'E'; select 'F';
   else select 'G'; select 'H';
end if;
```

Solution:

The MySQL IF statement can be easily emulated in SQL Server.

SQL Server example:

```
IF (1>2)
   BEGIN SELECT 'A' SELECT 'B' END
   ELSE IF (2>3) BEGIN SELECT 'C' SELECT 'D' END
   ELSE IF (3>4) BEGIN SELECT 'E' SELECT 'F' END
   ELSE BEGIN SELECT 'G' SELECT 'H' END
```

Issue: CASE Statement

MySQL and SQL Server have different syntax for the CASE statement.

MySQL example:

```
case int_value
when 1 then select 'A'; select 'AA';
when 2 then select 'B';
when 1 then select 'A1'; select 'A2'; -- ignored
when 3 then select 'C';
else select 'NULL';
end case;
```

Solution:

CASE statements can be emulated by using SQL Server IF statements.

SQL Server example:

```
IF @int_value=1 BEGIN SELECT 'A' SELECT 'AA' END
   ELSE IF @int_value=2 BEGIN SELECT 'B' end
   ELSE IF @int_value=1 BEGIN SELECT 'A1' SELECT 'A2' end
   ELSE IF @int_value=3 BEGIN SELECT 'C' END
   ELSE BEGIN SELECT 'NULL' END
```

Issue: LOOP and REPEAT Statements

SQL Server does not have the identical statements.

MySQL example:

```
declare i int;
set i=0;
m: loop
     set i:=i+1;
     if (sin(i) - cos(i) < 0) then leave m; end if;
   end loop;
select i; -- 4

set i=0;
repeat
  set i:=i+1;
until (sin(i) - cos(i) < -1)
end repeat;
select i; -- 5
```

Solution:

MySQL LOOP and REPEAT statements can be easily emulated by using WHILE statements in SQL Server.

SQL Server example:

```
DECLARE @i int;
SET @i=0;
WHILE 1=1
BEGIN
  SET @i=@i+1;
  IF (sin(@i) - cos(@i) < 0) BREAK;
END;
SELECT @i; -- 4

SET @i=0;
WHILE 1=1
BEGIN
  SET @i=@i+1;
  IF (sin(@i) - cos(@i) < -1) BREAK;
END
SELECT @i; -- 5
```

Issue: LEAVE and ITERATE Statements

SQL Server does not have the identical statements.

MySQL example:

```
create procedure proc_goto(s varchar(64), a int, b int)
begin
 m1: loop
     if (a>b) then leave m1; end if;
     set s:=concat(substring(s,1,a-1),
                   upper(substring(s,a,1)),substring(s,a+1));
     set a:=a+1;
     if (a>b) then iterate m1; end if;
     set a:=a+1;
     end loop;
 select s;
end

call proc_goto ('abcdefghijklmnopqrstuvwxyz',5,10)
-- abcdEfGhIjklmnopqrstuvwxyz
```

Solution:

Emulate this behavior by using Transact-SQL BREAK and CONTINUE statements.

SQL Server example:

```
CREATE PROCEDURE proc_goto (@s varchar(64), @a int, @b int) AS
BEGIN
 WHILE 1=1
 BEGIN
   IF (@a>@b) BREAK
   SET @s=substring(@s,1,@a-1)+
          UPPER(substring(@s,@a,1))+substring(@s,@a+1,len(@s));
   SET @a=@a+1;
   IF (@a>@b) CONTINUE
   SET @a=@a+1;
 END
   SELECT @s
END

EXEC proc_goto 'abcdefghijklmnopqrstuvwxyz',5,10
-- abcdEfGhIjklmnopqrstuvwxyz
```

Routines

Issue: DML Statements in Functions

MySQL functions can contain DML statements. This is not supported in SQL Server.

MySQL example:

```
create table TableFuncA (a int not null);
create table TableFuncB (b int not null);

create function new_func_a (par_int int) returns int
begin
delete from TableFuncA where a=par_int;
return row_count();
end

insert TableFuncA values (10), (20), (20), (30), (30), (30),
(40), (40), (40), (40);
insert TableFuncB values (20), (40), (50);

select new_func_a(b)
from TableFuncB; -- 2 4 0

select * from TableFuncA; -- 10 30 30 30
```

Solution:

Convert function with DML to procedure.

Triggers

This section explains how to convert MySQL triggers to SQL Server 2014 triggers.

Issue: FOR EACH ROW Triggers

MySQL supports only FOR EACH ROW triggers, which are not supported in SQL Server.

MySQL example:

```
create table t_data (
id int not null primary key,
v varchar(128) not null, log_date datetime not null);
create table t_log (
id int null, action varchar(6) null,
v_old varchar(128) null, v_new varchar(128) null,
log_date_old datetime null, log_date_new datetime null);

create trigger trg_data_ins
after insert
on t_data
for each row
begin
 declare a varchar(6);
 if (new.v!='') then set a:='INSERT'; else set a:='EMPTY'; end if;
 insert t_log (id,action,v_old,v_new,log_date_old,log_date_new)
 values (new.id,a,null,new.v,null,new.log_date);
end

insert t_data
values (1,'A',now()),(2,'B',now()),(3,'',now()),(4,'C',now());
```

Solution:

FOR EACH ROW trigger, functionality can be emulated by using a SQL Server cursor.

SQL Server example:

```
CREATE TRIGGER trg_data_ins
ON t_data
AFTER INSERT
```

```
AS
BEGIN
DECLARE @id int, @v varchar(128), @log_date datetime

DECLARE for_each_row CURSOR FORWARD_ONLY STATIC READ_ONLY
  FOR SELECT id, v, log_date FROM INSERTED

DECLARE @a varchar(6);

OPEN for_each_row

FETCH for_each_row INTO @id, @v, @log_date

WHILE @@fetch_status = 0
BEGIN
  IF (@v!='') SET @a='INSERT' ELSE SET @a='EMPTY';
  INSERT t_log (id,action,v_old,v_new,log_date_old,log_date_new)
  VALUES (@id,@a,null,@v,null,@log_date);
  FETCH for_each_row INTO @id, @v, @log_date
END

close for_each_row
deallocate for_each_row
end
```

Issue: BEFORE Triggers

MySQL supports BEFORE triggers. In MySQL triggers, the BEFORE keyword indicates that the trigger is invoked before the execution of the triggering statement.

Inside a trigger, you can refer to columns in the subject table (the table associated with the trigger) by using the aliases OLD and NEW. OLD.col_name refers to a column in an existing row before it is updated or deleted. NEW.col_name refers to the column of a new row to be inserted or an existing row after it is updated.

MySQL example:

```
create trigger trg_data_upd
before update
on t_data
for each row
begin
```

```
 set new.log_date:=now();
 if (old.v='') then set new.v:=''; end if;
 insert t_log (id,action,v_old,v_new,log_date_old,log_date_new)
 values (old.id,'UPDATE',old.v,new.v,old.log_date,new.log_date);
end

update t_data set v=concat(v,'+',v);
```

Solution:

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

SQL Server example:

```
CREATE TRIGGER trg_data_upd
ON t_data
INSTEAD OF UPDATE
AS
BEGIN
DECLARE @id_old int, @v_old varchar(128), @log_date_old datetime
DECLARE @id_new int, @v_new varchar(128), @log_date_new datetime

DECLARE for_each_row CURSOR FORWARD_ONLY STATIC READ_ONLY
  FOR SELECT id, v, log_date FROM DELETED

OPEN for_each_row

FETCH for_each_row INTO @id_old, @v_old, @log_date_old

WHILE @@fetch_status = 0
BEGIN

  SELECT @id_new=id, @v_new=v, @log_date_new=log_date
  FROM INSERTED WHERE id=@id_old

  SET @log_date_new=getdate();
  IF (@v_old='') SET @v_new='';
  INSERT t_log (id,action,v_old,v_new,log_date_old,log_date_new)
  VALUES (@id_old,'UPDATE',@v_old,@v_new,@log_date_old,@log_date_new);
```

```
  -- INSTEAD OF ----------------------------------------
  UPDATE t_data
  SET v=@v_new, log_date=@log_date_new
  WHERE id=@id_old
  ------------------------------------------------------

  FETCH for_each_row into @id_old, @v_old, @log_date_old
END

close for_each_row
deallocate for_each_row
end
```

SQL Mode (SQL_MODE System Variable)

Issue: Applying and Operating in Different SQL Modes

The MySQL server can operate in different SQL modes, and can apply these modes differently for different clients. Modes define what SQL syntax MySQL supports and the kind of data validation checks it performs.

You change and retrieve the SQL mode in MySQL by using the **sql_mode** system variable.

MySQL example:

```
SET sql_mode = '';
SELECT NOT 1 BETWEEN -5 AND 5; -- 0
SET sql_mode = 'HIGH_NOT_PRECEDENCE';
SELECT NOT 1 BETWEEN -5 AND 5; -- 1 -- (NOT 1) BETWEEN -5 AND 5
```

Solution:

Custom solutions can be created to emulate this issue in SQL Server 2014 using native Transact-SQL code.

Data Migration

This section describes the recommended procedure of transferring MySQL data into SQL Server tables.

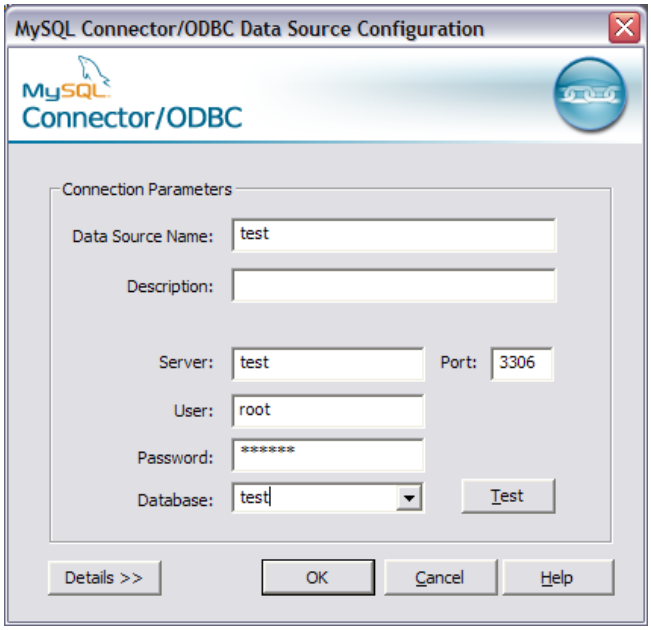
Migration Steps

The fastest way to copy the data from MySQL tables to SQL Server tables is to use the SQL Server Import and Export Data Wizard. For connectivity with MySQL server, install MySQL ODBC Connector, which can be downloaded from:

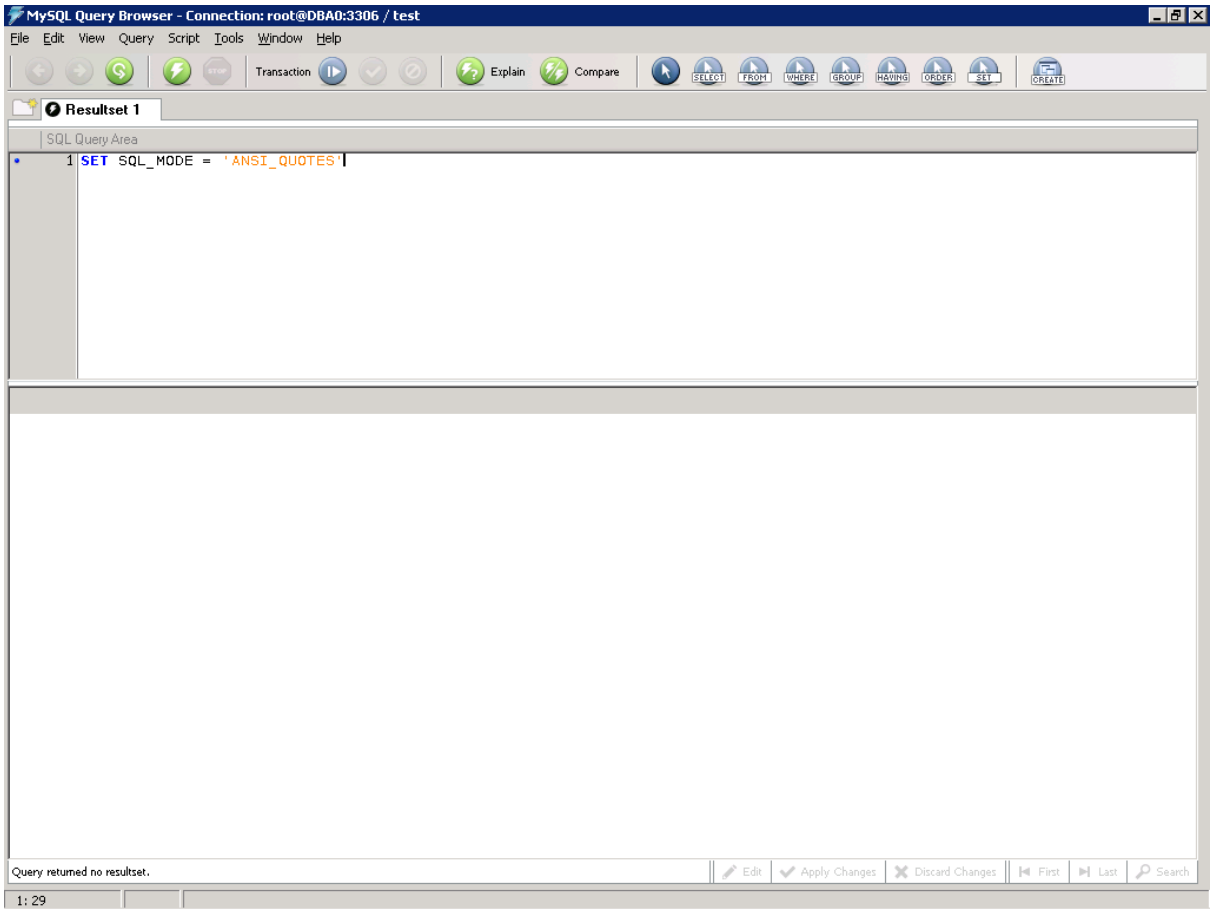
http://dev.mysql.com/downloads/connector/odbc

Here are the steps to perform the data migration using the wizard interface:

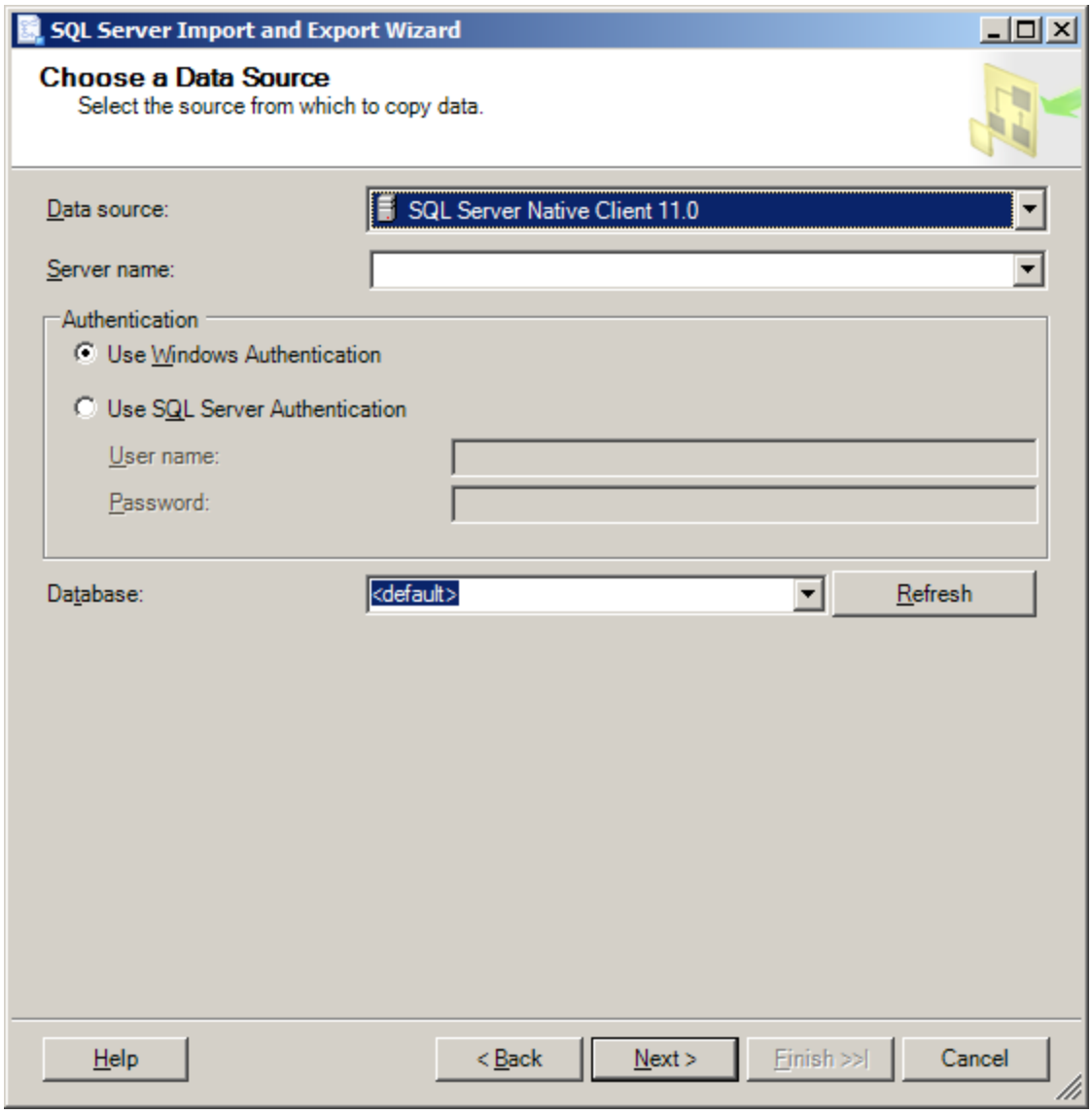
1. On the SQL Server machine, create system ODBC data source for MySQL database using MySQL ODBC Connector.



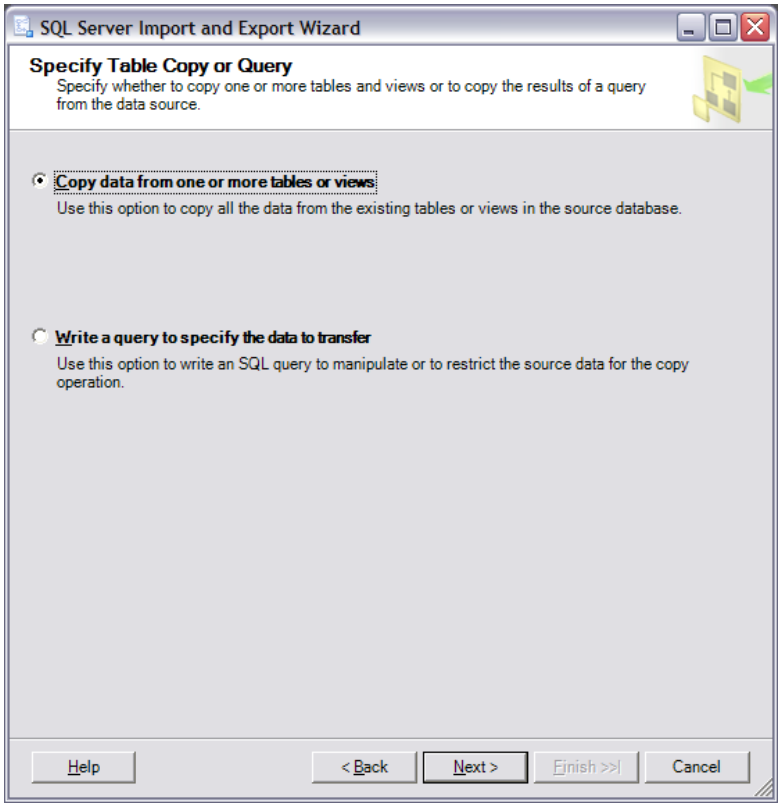
2. Set SQL_MODE to ANSI_QUOTES on MySQL Server.



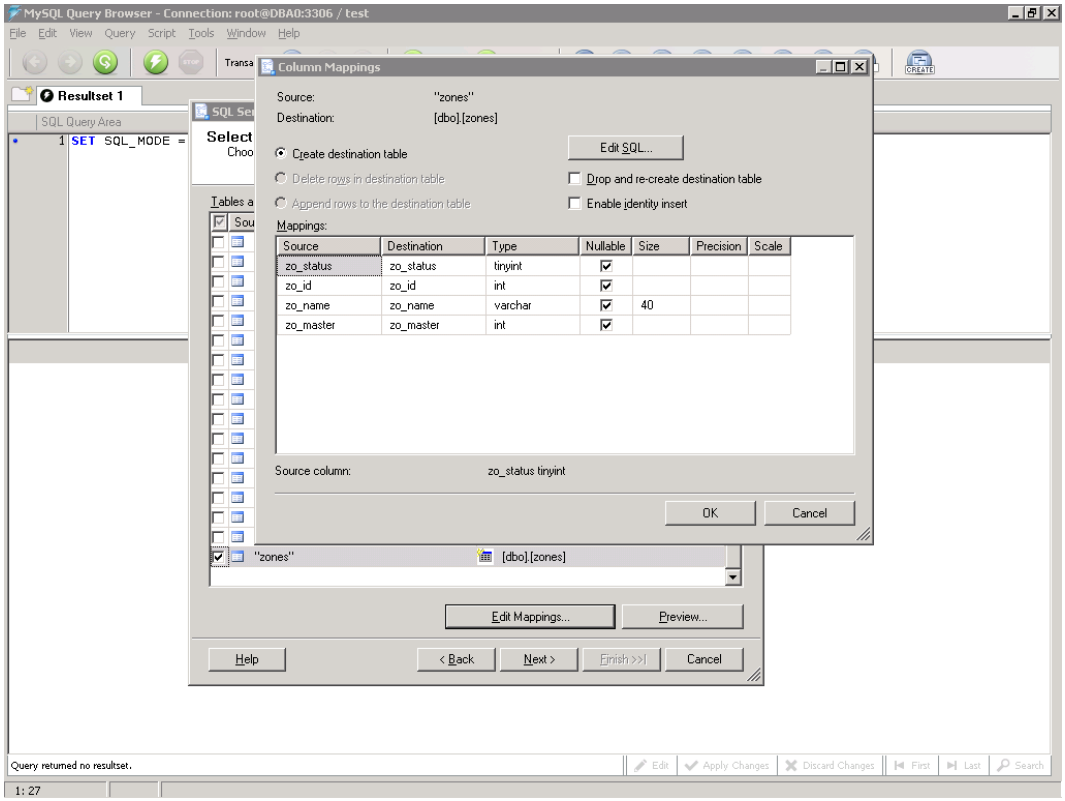
3. In SQL Server Management Studio, right-click the target database in Object Explorer, click **Tasks**, and then click **Import Data**. The SQL Server Import and Export Wizard appears.
4. On the next page, the wizard requests the data source. Select the.NET Data Provider for ODBC and specify the DSN created at the step 1.
5. Next, set up the destination. Select **SQL Server Native Client 11**, specify your instance of SQL Server and your database, and then click **Next**.



6. Click **Copy data from one or more tables or views**, and then click **Next**.



7. Select source and target tables for the migration. If a target table has an identity field, you can enable identity insert using the **Column Mappings** dialog box.



8. Execute the created package. If necessary, the package can be saved and reused later.

Validating Migration Results

After the data transfer is complete, you might want to verify that all the data have been migrated correctly. It can be done using the SQL Server linked server mechanism. The following commands illustrate this method. They should be executed in Query window of SQL Server Management Studio:

1. First, you need to create the linked server pointing to MySQL database.

   ```
   EXEC sp_addlinkedserver '<ServerName>','MySQL','MSDASQL', '<DSN>'
   ```

   where <ServerName> is name of linked server and <DSN> is the ODBC data source name.

2. Specify the login for this linked server.

   ```
   EXEC master.dbo.sp_addlinkedsrvlogin @rmtsrvname = N'<ServerName>',
   @locallogin = NULL , @useself = N'False', @rmtuser = N'<User>',
   @rmtpassword = N'<Password>'
   ```

   Here <ServerName> is name of the linked server, and <User> and <Password> are MySQL credentials that provide read access to the databases being transferred.

3. The following query returns the invalid rows where differences between MySQL and SQL Server tables exist. Change <database>, <schema>, and <table> so that they specify the location of the table for which you want the migration to be verified.

   ```
   (SELECT * FROM <database>.<schema>.<table>
   EXCEPT
   SELECT * FROM OPENQUERY(<ServerName>,'SELECT * FROM <schema>.<table>'))
   UNION
   (SELECT * FROM OPENQUERY(<ServerName>,'SELECT * FROM <schema>.<table>')
   EXCEPT
   SELECT * FROM <database>.<schema>.<table>)
   ```

Migrating MySQL System Functions

This section describes how to map MySQL system functions to equivalent SQL Server 2014 functions and provides solutions for converting MySQL functions.

Equivalent Functions

The following MySQL system functions are usable as is in SQL Server code:

ASCII, LEFT, LOWER, LTRIM, REPLACE, REVERSE, RIGHT, RTRIM, SOUNDEX, SPACE, SUBSTRING, UPPER, ABS, ACOS, ASIN, ATAN, ATAN2, CEILING, COS, COT, DEGREES, EXP, FLOOR, LOG, LOG10, PI, POWER, RADIANS, RAND, ROUND, SIGN, SIN, SQRT, TAN, DAY, MONTH, COALESCE, NULLIF, CAST, CONVERT.

Nonsupported Functions

The following MySQL functions cannot be easily emulated in SQL Server because of logical and physical organization and security model differences:

BENCHMARK, CHARSET, COERCIBILITY, COLLATION, CRC32, DATE_ADD with INTERVAL, DATE_SUB with INTERVAL, GET_FORMAT, PERIOD_ADD, PERIOD_DIFF, SUBTIME, TIMESTAMP, TIMESTAMPADD, TIMESTAMPDIFF, MATCH, EXTRACTVALUE, UPDATEXML, GET_LOCK, IS_FREE_LOCK, MASTER_POS_WAIT, RELEASE_LOCK.

Emulated Functions

Issue: Functions That Have a Variable Parameter Count

The following functions have a variable parameter count in MySQL:

GREATEST(value1, value2, …)

LEAST(value1, value2,...)

INTERVAL(N, N1, N2, N3, ...)

CHAR(N, ... [USING charset_name])

ELT(N, str1, str2, str3,...)

FIELD(str, str1, str2, str3, ...)

MAKE_SET(bits, str1, str2,...)

Solution:

Functions that have a variable parameter count can be emulated by using the Transact-SQL CASE function. Or, you can try to use the **xml** data type to pass data into an emulation function, but you must do an additional data conversion to and from XML format.

Issue: IF(expr1, expr2, expr3)

If expr1 is TRUE (expr1 <> 0 and expr1 <> NULL), IF() returns expr2; otherwise it returns expr3.

MySQL example:

```
if(@a>@b, @a, @b-@a)
```

Solution:

Emulate this function by using the Transact-SQL CASE function or new IIF function IIF.

SQL Server example:

```
CASE WHEN @a > @b THEN @a else @b - @a END
```

```
IIF(@a > @b, @a, @b-@a)
```

Issue: BIN(N)

Returns a string representation of the binary value of N.

Solution:

Emulate this function in Transact-SQL by using string functions and bitwise operators.

Issue: BIT_LENGTH(str)

Returns the length of the string *str* in bits.

Solution:

Emulate this function in Transact-SQL by using the DATALENGTH function.

Issue: CONCAT(str1, str2, ….). CONCAT_WS(separator, str1, str2, ...)

Returns the string that results from concatenating the arguments.

MySQL example:

```
CONCAT('A','B','C'), CONCAT_WS('#','A','B','C')
```

Solution:

Use the SQL Server plus operator (+) or new CONCAT function for string concatenation.

SQL Server example:

```
'A'+'B'+'C',  'A'+'#'+'B'+'#'+'C'

CONCAT('A', 'B', 'C'), CONCAT('A', '#', 'B', '#', 'C')
```

Issue: CONV(N, from_base, to_base)

Converts numbers between different number bases.

Solution:

Use Transact-SQL mathematical functions and bitwise operators to emulate this function.

Issue: EXPORT_SET(bits, on, off [, separator [, number_of_bits]])

Returns a string such that for every bit set in the value bits, you get an on string, and for every reset bit, you get an off string.

Solution:

Use Transact-SQL mathematical functions and bitwise operators to emulate this function.

Issue: FIND_IN_SET(str, strlist)

Returns a value in the range of 1 to *N* if the string *str* is in the string list *strlist* consisting of *N* substrings.

Solution:

Use the Transact-SQL CHARINDEX function to emulate this function.

Issue: FORMAT(X, D)

Formats the number *X* to a format like '#,###,###.##', rounded to *D* decimal places, and returns the result as a string.

Solution:

Use the Transact-SQL FORMAT, ROUND and CONVERT functions to emulate this function.

Issue: HEX(N_or_S)

If N_or_S is a number, returns a string representation of the hexadecimal value of N, where N is a longlong (BIGINT) number. If N_or_S is a string, returns a hexadecimal string representation of N_or_S where each character in N_or_S is converted to two hexadecimal digits. UNHEX(S) performs the inverse operation of HEX(S).

Solution:

Emulate HEX(N_or_S) functionality by using Transact-SQL string functions, convert functions, and bitwise operators.

Issue: INSERT(str, pos, len, newstr)

Returns the string *str*, with the substring that begins at position *pos* and is *len* characters long replaced by the string *newstr*.

Solution:

Use the Transact-SQL REPLACE or SUBSTRING functions to emulate this functionality.

Issue: LOAD_FILE(file_name)

Reads the file and returns the file contents as a string. SQL Server cannot read data from an external file into a variable.

Solution:

Emulate LOAD_FILE(file_name) by using bulk load statements or an extended stored procedure.

Issue: NOW()

Returns the current date and time.

MySQL example:

```
NOW()
```

Solution:

Use the similar Transact-SQL function, GETDATE.

SQL Server example:

```
GETDATE()
```

Issue: REPEAT(str, count)

Returns a string consisting of the string *str* repeated *count* times.

MySQL example:

```
REPEAT('A', 10)
```

Solution:

Use the similar Transact-SQL function, REPLICATE.

SQL Server example:

```
REPLICATE('A', 10)
```

Issue: ISNULL(expr)

If expr is NULL, ISNULL() returns 1; otherwise it returns 0.

MySQL example:

```
ISNULL(@a)
```

Solution:

Use the Transact-SQL CASE function and IS NULL clause to emulate this functionality.

SQL Server example:

```
CASE WHEN @a IS NULL THEN 1 ELSE 0 END
```

Issue: STRCMP(expr1, expr2)

Compares two strings.

Solution:

Try using Transact-SQL comparison operators to emulate STRCMP(*expr1*, *expr2*).

Issue: CONVERT_TZ(dt, from_tz, to_tz)

Converts a datetime value *dt* from the time zone given by *from_tz* to the time zone given by *to_tz* and returns the resulting value. SQL Server does not have time zone functionality.

Solution:

Time zone functionality can be emulated by using SQL Server CLR or extended stored procedures.

Issue: DATE_FORMAT(date, format)

Formats the date value according to the *format* string. Transact-SQL has similar FORMAT function but the format models differ significantly.

Solution:

You can use Transact-SQL FORMAT function or others date, string, and convert functions to emulate DATE_FORMAT(date, format) functionality.

Issue: FROM_DAYS(N)

Given a day number N, returns a DATE value.

Solution:

Use the Transact-SQL CONVERT function to emulate FROM_DAYS(N).

Issue: MAKEDATE(year, dayofyear)

Returns a date, given *year*, and *dayofyear* values.

Solution:

Use the Transact-SQL DATEADD function to emulate this function.

Issue: SEC_TO_TIME(seconds)

Returns the *seconds* argument, converted to hours, minutes, and seconds.

Solution:

Use Transact-SQL arithmetic operators and convert functions to emulate this function.

Issue: TIME_TO_SEC(time)

Returns the time argument, converted to seconds.

Solution:

Use Transact-SQL arithmetic operators and string functions to emulate this function.

Issue: TO_DAYS(date)

Given a date, returns the day number (the number of days since year 0).

Solution:

Use the Transact-SQL CONVERT function to emulate this function.

Issue: BIT_COUNT(N)

Returns the number of bits that are set in the argument N.

Solution:

Emulate the MySQL BIT_COUNT(N) function in Transact-SQL by using string functions and bitwise operators.

Issue: Encryption and Compression Functions

AES_ENCRYT. AES_DECRYPT. COMPRESS. UNCOMPRESS. ENCODE. DECODE. DES_ENCRYPT. DES_DECRYPT. ENCRYPT. MD5. OLD_PASSWORD. PASSWORD. SHA. SHA1. UNCOMPRESSED_LENGTH.

Solution:

Emulate this functionality by using SQL Server security and cryptographic functions.

Issue: LAST_INSERT_ID()

Returns the first automatically generated value that was set for an AUTO_INCREMENT column by the most recent INSERT or UPDATE statement to affect the column.

Solution:

Use the Transact-SQL @@IDENTITY or SCOPE_IDENTITY functions to emulate LAST_INSERT_ID().

Issue: DEFAULT(column)

Returns the default value for a table *column*.

Solution:

Use a system view of the data to emulate DEFAULT(column).

Issue: INET_ATON(expr)

Given the dotted-quad representation of a network address as a string, returns an integer that represents the numeric value of the address. INET_NTOA(expr). Given a numeric network address (4 or 8 byte), returns the dotted-quad representation of the address as a string.

Solution:

Use Transact-SQL arithmetic operators and string functions to emulate these functions.

Issue: GROUP_CONCAT(expr)

This function returns a string result with the concatenated non-NULL values from a group.

Solution:

This function can be emulated by using Transact-SQL code as in the following example:

```
declare @v varchar(max)
set @v=''
select @v=@v+','+isnull(field_a,'') from table_1
select substring(@v,2,len(@v))
```

Issue: INSTR(str, substr), POSITION(substr IN str)

Returns the position of the first occurrence of substring *substr* in string *str*. LOCATE(substr, str [, pos]). Returns the position of the first occurrence of substring substr in string *str*, starting at position *pos*.

Solution:

Use the CHARINDEX function to emulate this functionality.

Conclusion

From this migration guide you learned the differences between MySQL and SQL Server 2014 database platforms, and the steps necessary to convert a MySQL database to SQL Server.

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.