Microsoft SQL Server

Guide to Migrating from Oracle to SQL Server 2014 and Azure SQL Database

SQL Server Technical Article

Writers: Yuri Rusakov (DB Best Technologies), Igor Yefimov (DB Best Technologies), Anna Vynograd (DB Best Technologies), Galina Shevchenko (DB Best Technologies)

Technical Reviewer: Dmitry Balin (DB Best Technologies)

Published: November 2014

Applies to: SQL Server 2014

Summary: This white paper explores challenges that arise when you migrate from an Oracle 7.3 database or later to SQL Server 2014. It describes the implementation differences of database objects, SQL dialects, and procedural code between the two platforms. The entire migration process using SQL Server Migration Assistant (SSMA) v6.0 for Oracle is explained in depth, with a special focus on converting database objects and PL/SQL code.

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

Migrating from an Oracle database to Microsoft® SQL Server® 2014 frequently gives organizations benefits that range from lowered costs to a more feature-rich environment. The free Microsoft SQL Server Migration Assistant (SSMA) for Oracle speeds the migration process. SSMA for Oracle V6.0 converts Oracle database objects (including stored procedures) to SQL Server database objects, loads those objects into SQL Server, migrates data from Oracle to SQL Server, and then validates the migration of code and data.

This white paper explores the challenges that arise during migration from an Oracle database to SQL Server 2014. It describes the implementation differences of database objects, SQL dialects, and procedural code between the two platforms.

Overview of Oracle-to-SQL Server 2014 Migration

This section explains the entire SSMA for Oracle migration process, with a special focus on converting database objects and PL/SQL code.

Main Migration Steps

The first migration step is to decide on the physical structure of the target SQL Server database. In the simplest case, you can map the Oracle tablespaces to SQL Server filegroups. However, because the files in the filegroups and the information stored in the files are usually different, this is not usually possible.

The next step is to choose how to map the Oracle schemas to the target. In SQL Server, schemas are not necessarily linked to a specific user or a login, and one server contains multiple databases.

You can follow one of two typical approaches to schema mapping:

- By default in SSMA, every Oracle schema becomes a separate SQL Server database. The target SQL Server schema in each of these databases is set to dbo—the predefined name for the database owner. Use this method if there are few references between Oracle schemas.
- Another approach is to map all Oracle schemas to one SQL Server database. In this case, an Oracle schema becomes a SQL Server schema with the same name. To use this method, you change the SSMA default settings. Use this method if different source schemas are deeply linked with each other (for instance if there are cross-references between Oracle tables in different schemas, when trigger is on the table and the tables itself are in different schemas…).

SSMA applies the selected schema-mapping method consistently when it converts both database objects and the references to them.

After you chose your optimal schema mapping, you can start creating the target SQL Server database and its required schemas. Because the SQL Server security scheme is quite different from Oracle's, we chose not to automate the security item migration in SSMA. That way, you can consider all possibilities and make the proper decisions yourself.

The typical SSMA migration includes connecting to the source Oracle server, selecting the server that is running SQL Server as the target, and then performing the Convert Schema command. When the target objects are created in the SSMA workspace, you can save them by using the Load to Database command. Finally, execute the Migrate Data command, which transfers the data from the source to the target tables, making the necessary conversions. The data migration process is executed on the server that is running SQL Server. The internal implementation of this feature is described in Data Migration Architecture of SSMA for Oracle.

Conversion of Database Objects

Not all Oracle database objects have direct equivalents in SQL Server. In many cases, SSMA creates additional objects to provide the proper emulation. General conversion rules are as follows:

- Each Oracle table is converted to a SQL Server table. During the conversion, all indexes, constraints, and triggers defined for a table are also converted. When determining the target table's structure, SSMA uses type mapping definitions. Data type conversion is described in Migrating Oracle Data Types. Now, conversion to memory-optimized tables is supported.

- An Oracle view is converted to a SQL Server view. This also concerns materialized views which are migrated to indexed ones. SSMA creates emulations for commonly used Oracle system views. For more information about system view conversion, see Emulating Oracle System Objects.

- Oracle stored procedures are converted to SQL Server stored procedures. Note that Oracle procedures can use *nested subprograms*, which means that another procedure or function can be declared and called locally within the main procedure. SSMA can convert inline subprograms automatically, see Converting Nested PL/SQL Subprograms.

- Oracle user-defined functions are converted to SQL Server functions if the converted function can be compatible with SQL Server requirements. Otherwise, SSMA creates two objects: one function and one stored procedure. The additional procedure incorporates all the logic of the original function and is invoked in a separate process. For more information, see Migrating Oracle User-Defined Functions. SSMA emulates most of the Oracle standard functions. See the complete list in Emulating Oracle System Objects.

- Oracle DML triggers are converted to SQL Server triggers, but because the trigger functionality is different, the number of triggers and their types can be changed. See a description of trigger conversion in Migrating Oracle Triggers.

- Some Oracle object categories, such as packages, do not have direct SQL Server equivalents. SSMA converts each packaged procedure or function into separate target subroutines and applies rules for stand-alone procedures or functions. Other issues related to package conversion, such as converting packaged variables, cursors, and types are explained in Emulating Oracle Packages. In addition, SSMA can emulate some commonly used Oracle system packages. See their description in Emulating Oracle System Objects.

- SQL Server 2014 has a sequences mechanism, though some features of Oracle sequences (e.g. CURRVAL) are not supported in SQL Server, but you can find methods to manually convert them in Sequences Conversion.

- Oracle private synonyms are converted to SQL Server synonyms stored in the target database. SSMA converts public synonyms to synonyms defined in the **ssma_oracle** schema.

Differences in SQL Languages

Oracle and SQL Server use different dialects of the SQL language, but SSMA can solve most of the problems introduced by this difference. For example, Oracle uses CONNECT BY statements for hierarchical queries, while SQL Server implements hierarchical queries by using common table expressions. The syntax of common table expressions does not resemble the Oracle format, and the order of tree traversal is different. To learn how SSMA converts hierarchical queries, see Migrating Hierarchical Queries.

Or consider how SSMA handles another nonstandard Oracle feature: the special outer join syntax with the (+) qualifier. SSMA converts these queries by transforming them into ANSI format.

Oracle pseudocolumns, such as ROWID or ROWNUM, present a special problem. When converting ROWNUM, SSMA emulates it with the TOP keyword of the SELECT statement if this pseudocolumn is used only to limit the size of the result set. If the row numbers appear in a SELECT list, SSMA uses the ROW_NUMBER( ) function. The ROWID problem can be solved by an optional column named ROWID, which stores a unique identifier in SQL Server.

SSMA does not convert dynamic SQL statements because the actual statement is not known until execution time and, in most cases, it cannot be reconstructed at conversion time. There is a workaround: The Oracle metabase tree displayed in SSMA contains a special node named Statements in which you can create and convert ad hoc SQL statements. If you can manually reproduce the final form of a dynamic SQL command, you can convert it as an object in the Statements node.

PL/SQL Conversion

The syntax of Oracle's PL/SQL language is significantly different from the syntax of SQL Server's procedural language, Transact-SQL. This makes converting PL/SQL code from stored procedures, functions, or triggers a challenge. SSMA, however, can resolve most of the problems related to these conversions. SSMA also allows establishing special data type mappings for PL/SQL variables.

Some conversion rules for PL/SQL are straightforward, such as converting assignment, IF, or LOOP statements. Other SSMA conversion algorithms are more complicated. Consider one difficult case: converting Oracle exceptions, which is described in Emulating Oracle Exceptions. The solution detailed there allows emulating Oracle behavior as exactly as possible, but you may need to review the code in order to eliminate dependencies on Oracle error codes and to simplify the processing of such conditions as NO_DATA_FOUND.

Oracle cursor functionality is not identical to cursor functionality in SQL Server. SSMA handles the differences as described in Migrating Oracle Cursors.

Oracle transactions are another conversion issue, especially autonomous transactions. In many cases you must review the code generated by SSMA to make the transaction implementation best suited to your needs. For instructions, see Simulating Oracle Transactions in SQL Server 2014 and Simulating Oracle Autonomous Transactions.

Finally, many PL/SQL types do not have equivalents in Transact-SQL. Records and collections are examples of this. SSMA can process most cases of PL/SQL record and collections usage. We also propose several approaches to the manual emulation of PL/SQL collections in Migrating Oracle Collections and Records.

Data Migration Architecture of SSMA for Oracle

This section describes SSMA for Oracle V6.0 components and their interaction during data migration. The components execute on different computers and use Microsoft SQL Server 2014 database objects for communication. This architecture produces the best migration performance and flexibility. Understanding this mechanism can help you set up the proper environment for SSMA data migration. It also helps you to better control, monitor, and optimize the process.

Implementation in SSMA

We based the SSMA for Oracle V6.0 implementation on the **SqlBulkCopy** class, defined in the .NET Framework 2.0. **SqlBulkCopy** functionality resembles the **bcp** utility, which allows transferring large amounts of data quickly and efficiently. Access to the source database is established by the .NET Framework Data Provider for Oracle, which uses the Oracle Call Interface (OCI) from Oracle client software. Optionally, you can use .NET Framework Data Provider for OLE DB, which requires an installed Oracle OLE DB provider.

We considered the following when designing SSMA for Oracle data migration:

- The data transfer process must run on SQL Server. That limits the number of installed Oracle clients and reduces network traffic.
- The client application controls the process by using SQL Server stored procedures. Therefore, you do not need any additional communication channels with the server and can reuse the existing server connection for this purpose.
- All tables that are selected for migration are transferred by a single execution command from the SSMA user.
- The user monitors the data flow progress and can terminate it at any time.

Solution Layers

Four layers participate in the data migration process:

- Client application, an SSMA executable
- Stored procedures that serve as interfaces to all server actions
- The database layer, which comprises two tables:
    - The package information table
    - The status table
- The server executable, which starts as part of a SQL Server job, executes the data transfer, and reflects its status

Client Application
SSMA lets users choose an arbitrary set of source tables for migration. The batch size for bulk copy operations is a user-defined setting.

When the process starts, the program displays the progress bar and a **Stop** button. If any errors are found, SSMA shows the appropriate error message and terminates the transfer. In addition, the user can click **Stop** to terminate the process. If the transfer is completed normally, SSMA compares the number of rows in each source with the corresponding target table. If they are equal, the transfer is considered to be successful.

As the client application does not directly control the data migration process, SSMA uses a Messages table to receive feedback about the migration status.

Stored Procedures Interface
The following SQL Server stored procedures control the migration process:

- **bcp_save_migration_package** writes the package ID and XML parameters into the **bcp_migration_packages** table.
- **bcp_start_migration_process** creates the SQL Server job that starts the migration executable and returns the ID of the job created.
- **bcp_read_new_migration_messages** returns the rows added by the migration executable, filtered by known job ID.
- **stop_agent_process** stops the migration job, including closing the original connections and killing the migration executable. The data will be migrated partially.
- **bcp_clean_migration_data** is a procedure that cleans up a migration job.
- **bcp_post_process** is a procedure that runs all post-processing tasks related to the single migrated table.

Database Layer
SSMA uses a Packages table, named [ssma_oracle].[bcp_migration_packages], to store information about the current package. Each row corresponds to one migration run. It contains package GUID and XML that represents RSA-encrypted connection strings and the tables that should be migrated.

A Messages table, named [ssma_oracle].[ssmafs_bcp_migration_messages] accumulates messages coming from migration executables during their work.

Migration Executable
The migration application, SSMA for Oracle Data Migration Assistant.exe, is executed on a SQL Server host. The executable's directory is determined during the Extension Pack installation. When **bcp_start_migration_package** starts the application, it uses hard-coded file names and retrieves the directory name from a server environment variable.

When it starts, the migration application gets the package ID from the command string and reads all other package-related information from the Packages table. That information includes source and destination connection strings, and a list of the tables to migrate. Then the tables are processed one at a time. You get source rows via the **IDataReader** interface and move them to the target table with the **WriteToServer** method.

The **BatchSize** setting defines the number of rows in a buffer. When the buffer is full, all rows in it are committed to the target.

To notify you about the progress of a bulk copy operation, the data migration executable uses the **SqlRowsCopied** event and **NotifyAfter** property. When a **SqlRowsCopied** event is generated, the application inserts new rows, sending information about the progress to the Messages table. The **NotifyAfter** property defines the number of rows that are processed before generating a **SqlRowsCopied** event. This number is 25 percent of the source table's row count.

Another type of output record—the termination message—is written to the Messages table when the application terminates either successfully or because of an exception. In the latter case, the error text is included. If **BatchSize** = 1, additional information about the columns of the row where the problem occurred is extracted, so that you can locate the problematic row.

Message Handling

The client application receives feedback from the migration executable by means of the Messages table. During migration, the client is in the loop, polling this table and verifying that new rows with the proper package ID appear there. If there are no new rows during a significant period of time, this may indicate problems with the server executable and the process terminates with a time-out message.

When the table migration completes, the server executable writes a successful completion message. If the table is large enough, you may see many intermediate messages, which show that the next batch was successfully committed. If an error occurs, the client displays the error message that was received from the server process.

Validation of the Results

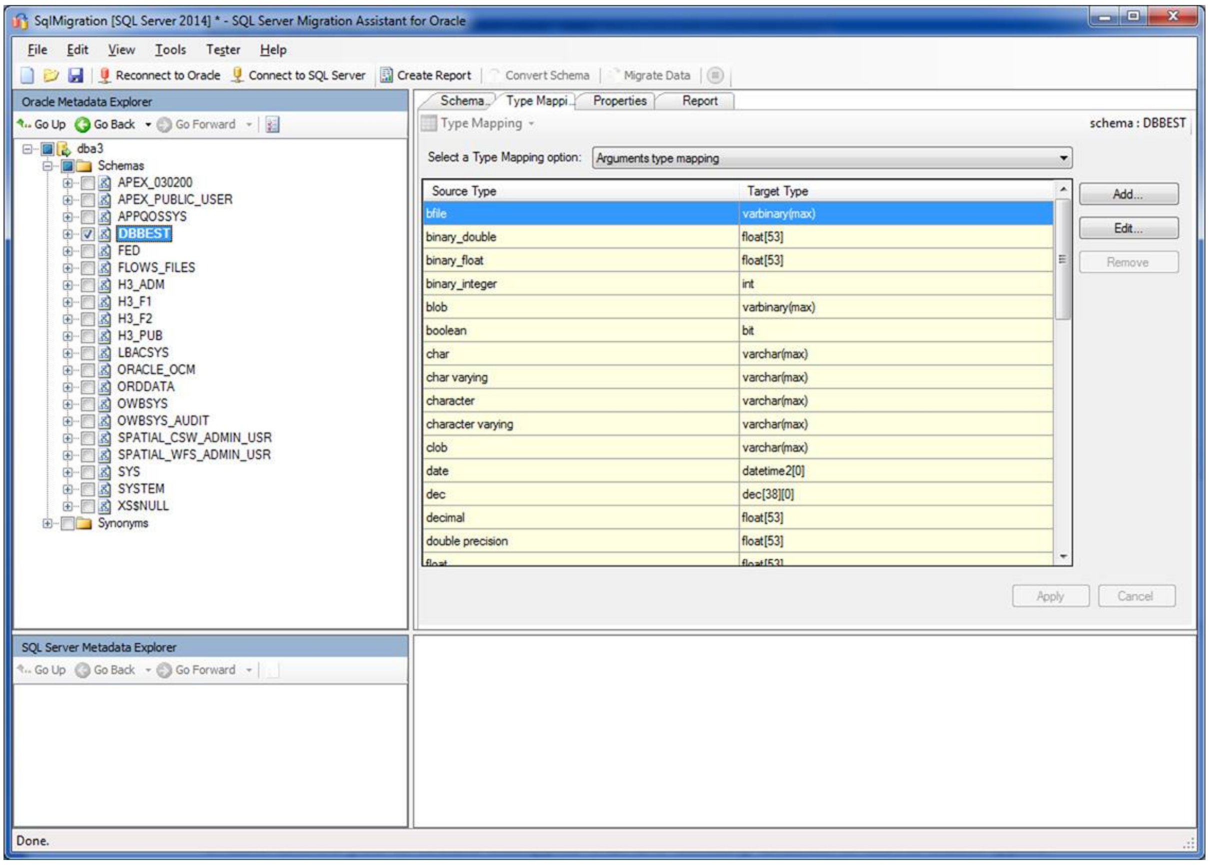
Before the migration starts, the client application calculates the number of rows in each table that will be migrated. With this data, you can evaluate the correct progress position.

After the migration completes, the client must calculate the target table's row counts. If they are equal, the overall migration result is considered to be successful. Otherwise, the user is notified of the discrepancy and can view the source and destination counts.

Migrating Oracle Data Types

Most data types used in Oracle do not have exact equivalents in Microsoft SQL Server 2014. They differ in scale, precision, length, and functionality. This section explains the data type mapping implemented in SSMA for Oracle V6.0, and it includes remarks about conversion issues.

SSMA supports all built-in Oracle types. SSMA type mapping is applied to table columns, subprogram arguments, a function's returned value, and to local variables. Usually the mapping rules are the same for all these categories, but in some cases there are differences. In SSMA, you can adjust mapping rules for some predefined limits. You can establish custom mappings for the whole schema, for specific group of objects, or to a single object on the Oracle view pane's **Type Mapping** tab (Figure 1).



schema : DBBEST

Select a Type Mapping option: Arguments type mapping

Source Type	Target Type
bfile	varbinary(max)
binary_double	float[53]
binary_float	float[53]
binary_integer	int
blob	varbinary(max)
boolean	bit
char	varchar(max)
char varying	varchar(max)
character	varchar(max)
character varying	varchar(max)
clob	varchar(max)
date	datetime2[0]
dec	dec[38][0]
decimal	float[53]
double precision	float[53]
float	float[53]

Figure 1: The Type Mapping tab in Oracle

This section does not describe migrating complex data types such as object types, collections, or records. It does not cover ANY types and some specific structures, such as spatial or media types.

Oracle allows you to create subtypes that are actually aliases of some basic types. SSMA does not process subtypes, but you can emulate that functionality manually if you can convert the basic type. Generally it is enough to replace the Oracle declaration:

```
SUBTYPE <type-name> IS <basic-type> [NOT NULL]
```

With the SQL Server 2014 declaration:

```
CREATE TYPE <type-name> FROM <basic-type-converted> [NOT NULL]
```

You may need to change the target `<type-name>` if the subtype is defined in the Oracle package. To establish the scope of this name, add a package prefix such as PackageName$<type-name>.

Numeric Data Types

The basic fixed point numeric type in Oracle is NUMBER(<precision>, <scale>). Its variation for integer numbers is NUMBER(<precision>), and a floating point value can be stored in NUMBER.

By default, SSMA maps NUMBER(<precision>, <scale>) to **numeric**(<precision>, <scale>) and NUMBER(<precision>) to **numeric**(<precision>). NUMBER becomes float(53), which has the maximum precision from SQL Server floating-point numbers.

In Oracle, INTEGER(<precision>) and INTEGER types are treated like NUMBER(<precision>, 0). Because SQL Server has a special **int** type that stores integers more efficiently, SSMA maps INTEGER to **int**. PL/SQL types such as BINARY_INTEGER and PLS_INTEGER are also mapped to **int** by default.

You may want to customize the default mapping of numeric types if you know the exact range of actual values. In fact, you can choose any SQL Server numeric type as the target for the mapping. Be cautious when mapping a source type to a type that has less precision, such as NUMBER -> smallint or NUMBER(20) -> int. Doing so could create overflows or loss of precision during data migration or during code execution. In some cases, you may want to set the precision to larger than the default, such as when mapping INTEGER to **bigint**.

You may find another reason to change default number mappings: when you convert a NUMBER field to a SQL Server identity column. Because SQL Server does not support float numbers as identities, change it to an **int** or numeric type.

SSMA recognizes various synonyms of NUMBER types such as NUMERIC, DECIMAL, NATURAL, POSITIVE, DOUBLE_PRECISION, REAL, BINARY_FLOAT, and BINARY_DOUBLE and applies the proper mapping for each one.

SIGNTYPE is mapped to **smallint** to allow storing -1 as a possible value.

Character Data Types

SSMA converts the basic character types VARCHAR2 and CHAR to SQL Server **varchar** and **char**, correspondingly preserving their length. If a PL/SQL variable is declared with a constant size greater than 8,000, SSMA maps to **varchar(max)**.

If some formal parameter of a procedure or a function has a character type, Oracle does not require that its length be explicitly declared. Meanwhile, SQL Server always wants to know the exact size of **varchar** or **char** parameters. As a result, SSMA has no other choice than to apply the maximum length by default. That means that VARCHAR2 or CHAR parameters are automatically declared as **varchar(max)** in the target code. If you know the exact length of the source data, you can change the default mapping.

Use customized mappings when Oracle is configured to store multibyte strings in VARCHAR2/CHAR columns or variables. In that case, map the character types to Unicode types in SQL Server. For example:

```
VARCHAR2    -> nvarchar
CHAR        -> nchar
```

Otherwise, non-ASCII strings can be distorted during data migration or target code execution. Note that source strings declared as national (NVARCHAR2 and NCHAR) are automatically mapped to **nvarchar** and **nchar**.

A similar approach is applied to Oracle RAW strings. This type can be mapped to binary or **varbinary** (the default), but if their size exceeds the 8,000-byte limit, map them to **varbinary(max)**.

SSMA recognizes various synonyms of these types, namely VARCHAR, CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, and STRING.

Date and Time

The default conversion target for DATE is **datetime2**[0]. Note that the SQL Server **datetime** type can store dates from 01/01/1753 to 12/31/9999 and **datetime2** type can store dates from 01/01/0001 to 12/31/9999. This range is not as wide as Oracle's DATE, which starts from 4712 BC. This can create problems if these early dates are used in the application. However, SQL Server can store contemporary dates more efficiently with the **smalldatetime** type, which supports dates from 01/01/1900 to 06/06/2079. To customize the mapping, in SSMA choose **smalldatetime** as the target type.

Another Oracle type that holds the date and time is TIMESTAMP. It resembles DATE except that it has greater precision (up to nanoseconds). The SQL Server **timestamp** is a completely different type not related to a moment in time. Thus, the best way to convert TIMESTAMP is to use the default SSMA mapping to **datetime2**. The accuracy of **datetime2** is 100 nanoseconds. In most cases, the loss of precision caused by this conversion is acceptable. The SQL Server 2014 can store time zone information in dates. This is supported by the datetimeoffset data type.

The Oracle INTERVAL data type does not have a corresponding type in SQL Server, but you can emulate any operations with intervals by using the SQL Server functions DATEADD and DATEDIFF. The syntax of DATEADD is quite different from the syntax of DATEDIFF, and as of this writing SSMA does not perform these conversions automatically.

Boolean Type

SQL Server does not have a Boolean type. Statements containing Boolean values are transformed by SSMA to replace the value with conditional expressions. SSMA emulates stored Boolean data by using the SQL Server **bit** type.

Large Object Types

The best choice for migrating Oracle large object types (LOBs) are SQL Server variable-length types with maximum storage size: **varchar(max)**, **nvarchar(max)**, and **varbinary(max)**.

Oracle	SQL Server 2014
LONG, CLOB	**varchar(max)**
NCLOB	**nvarchar(max)**
LONG RAW, BLOB, BFILE	**varbinary(max)**

You can change SSMA mapping to use the older-style **text**, **ntext**, and **image** types, but this is not recommended. SQL Server 2014 operations over the variable-length types with maximum storage size are simple compared to the approaches in both Oracle and SQL Server 2014. Currently, SSMA does not automatically convert operations on large types. Still, it can migrate the data of all the above types. The BFILE type is somewhat different; because SSMA does not convert the Oracle concept of saving data out of the database, the result of the data migration is that the file contents are loaded into a SQL Server table in binary format. You may consider converting that result into a **varchar** format if the file is a text file. If you need to store large binary fields in file system, you can manually convert them by using new SQL Server FILESTREAM attribute with the **varbinary(max)** data type. New SQL Server FileTable table type which builds on the FILESTREAM functionality will allow access through Windows to the properties of files stored on the NT file system. You can apply this new table type in manual conversion. Note that FILESTREAM data is not supported by Azure SQL DB.

If the Oracle server supports multibyte encoding of characters, map LONG and CLOB types to **nvarchar(max)** to preserve the Unicode characters.

XML Type

The default mapping of the Oracle XMLType is to SQL Server **xml**. All XML data in XMLType columns can be successfully migrated by using SSMA. Note that XQuery operations on these types are similar in Oracle and SQL Server, but differences exist and you should handle them manually.

ROWID Types

The ROWID and UROWID types are mapped to **uniqueidentifier**, which is a GUID that could be generated for each row. Before you convert any code that relies on the ROWID pseudocolumn, ensure that SSMA added the ROWID column (see option **Generate ROWID column** in the SSMA project settings). You can migrate data in columns of ROWID type to SQL Server as is, but their correspondence with the SSMA-generated ROWID column will be broken because **uniqueidentifier** no longer represents the physical address of a row like it was in Oracle.

Migrating Oracle Spatial Data

Oracle Spatial is an Oracle subsystem which provides SQL functions to facilitate the handling of spatial features in an Oracle database. The geometric description of a spatial object is stored in a single row, in a column of dedicated object type MDSYS.SDO_GEOMETRY.

SQL Server 2014 also supports spatial data. They are implemented as SQL CLR types named **geography** and **geometry**. The **geography** type allows you to store objects defined by coordinates on Earth's surface, and the **geometry** type is used for planar objects. SQL Server 2014 spatial data types implement methods for importing and exporting data in Well Known Text (WKT) and Well Known Binary (WKB) formats that are defined by Open Geospatial Consortium (OGC) specification. Spatial functionality is supported in all editions of SQL Server 2014, including Express.

SSMA for Oracle V6.0 does not support migration of table columns that have SDO_GEOMETRY type. Straightforward use of SQL Server Integration Services (SSIS) does not help much, because the Oracle Spatial types are not recognized by existing OLE DB, ADO.NET or ODBC providers.

The proposed solution is based on the fact that both Oracle Spatial and SQL Server 2014 support conversion to WKT format. Next, we are assuming that the source SDO_GEOMETRY column is mapped to SQL Server column of the geography type. Before transferring the data, we should create a SQL Server linked server pointing at the source Oracle instance. To perform the migration, we need to convert the source column value into WKT format, which makes it a plain text, and insert the result into the target geography column using OPENQUERY statement.

Example:

Suppose we have an Oracle table defined as:

```
CREATE TABLE geoinfo (id NUMBER(10) NOT NULL, geo MDSYS.SDO_GEOMETRY);
```

Its SQL Server counterpart will be:

```
CREATE TABLE geoinfo (id NUMERIC(10) NOT NULL, geo geography);
```

In this case, the following INSERT statement will correctly copy the spatial data.

```
INSERT INTO geoinfo (id, geo)
SELECT id, geography::STGeomFromText(CAST(geo as nvarchar(max)), srid)
FROM OPENQUERY(ORACLE_LS,
’SELECT id, SDO_UTIL.TO_WKTGEOMETRY(g.geo) geo, g.geo.sdo_srid srid
            FROM geoinfo g’)
```

Here ORACLE_LS is the name of linked server referencing the source Oracle instance. The Oracle function TO_WKTGEOMETRY returns a Well Known Text representation of the Spatial geometry object. The spatial reference ID (srid) is necessary to define the way the WKT string is interpreted by SQL Server.

Azure SQL DB supports spatial data and allows storing and processing it, but you should use another way to retrieve the data from Oracle spatial table as Azure SQL DB doesn’t support OPENQUERY function.

Emulating Oracle System Objects

This section describes how SSMA for Oracle V6.0 converts Oracle system objects including views, standard functions, and packaged subroutines. You will also find hints about how to convert packages that are currently unsupported.

Converting Oracle System Views

SSMA for Oracle V6.0 can convert Oracle system views, which are frequently used. It does not convert columns that are too closely linked with Oracle physical structures or have no equivalent in SQL Server 2014. The following views can be migrated automatically to SQL Server views:

- ALL_INDEXES
- DBA_INDEXES
- ALL_OBJECTS
- DBA_OBJECTS
- ALL_SYNONYMS
- DBA_SYNONYMS
- ALL_TAB_COLUMNS
- DBA_TAB_COLUMNS
- ALL_TABLES
- DBA_TABLES
- ALL_CONSTRAINTS
- DBA_ CONSTRAINTS
- ALL_SEQUENCES
- DBA_SEQUENCES
- ALL_VIEWS
- DBA_VIEWS
- ALL_USERS
- DBA _USERS
- ALL_SOURCE
- DBA_SOURCE

- GLOBAL_NAME
- ALL_JOBS
- DBA_ JOBS
- V$SESSION

In this section, we describe ways to manually convert the following views:

- ALL_EXTENTS
- V$LOCKED_OBJECT
- DBA_FREE_SPACE
- DBA_SEGMENTS

Location of Generated System View Emulations for SSMA for Oracle V6.0

Views emulating Oracle DBA_* views and ALL_* views are created in `<target_db>.ssma_oracle.DBA_*` and `<target_db>.ssma_oracle.ALL_*`, correspondingly.

USER_* views are created in each scheme where these views are used, and they have additional WHERE conditions with the format:

```
OWNER = <target_schema>
```

Note that SSMA creates only those target views that are actually referenced in the generated code.

Note In the following code we assume that SSMA creates DBA_* and USER_* views based on ALL_* and therefore we do not describe DBA_* and USER_*in this document.

Example:

```
CREATE VIEW ssma_oracle.ALL_TRIGGERS

AS

select

  UPPER(t.name) as TRIGGER_NAME,

  UPPER(s.name) as TABLE_OWNER,

  UPPER(o.name) as TABLE_NAME,

  CASE

    WHEN t.is_disabled = 0 THEN 'ENABLED'
    ELSE 'DISABLED'
  END as STATUS
 from sys.triggers t, sys.tables o, sys.schemas AS s
where t.parent_id = o.object_id
  and o.schema_id = s.schema_id
GO

CREATE VIEW USER1.USER_TRIGGERS
AS
SELECT * FROM ssma_oracle.ALL_TRIGGERS v
  WHERE v.OWNER = N'TEST_USER'


CREATE SYNONYM ssma_oracle.DBA_TRIGGERS
FOR TEST_DATABASE.ssma_oracle.ALL_TRIGGERS
```

ALL_INDEXES System View
SSMA converts owner, index_name, index_type, table_owner, table_name, table_type, uniqueness, compression, and prefix_length columns.

ALL_OBJECTS System View
SSMA converts owner, object_name, object_type, created, last_ddl_time, and generated columns.

ALL_SYNONYMS System View
SSMA converts all columns for this view.

ALL_TAB_COLUMNS System View
SSMA converts OWNER, table_name, column_name, DATA_TYPE, data_length, data_precision, data_scale, nullable, and column_id columns.

ALL_TABLES System View
SSMA for Oracle V6.0 converts owner and table_name columns.

ALL_CONSTRAINTS System View
SSMA converts owner, constraint_name, constraint_type, table_name, search_condition, r_owner, r_constraint_name, delete_rule, status, deferable, and generated columns.

ALL_SEQUENCES System View

SSMA converts sequence_owner, sequence_name, minvalue, increment_by, cycle_flag, order_flag, cache_size, and last_number columns.

ALL_VIEWS System View

SSMA converts owner, view_name, text_length, and text columns.

ALL_USERS System View

SSMA converts all columns for this view.

ALL_SOURCE System View

SSMA converts owner, name, and text columns.

GLOBAL_NAME System View

SSMA converts all columns for this view.

ALL_JOBS System View

SSMA converts job, last_date, last_sec, next_date, next_sec, total_time, broken, and what columns.

V$SESSION System View

SSMA converts sid, username, status, schemaname, program, logon_time, and last_call_et columns.

DBA_EXTENTS System View

SSMA does not automatically convert DBA_EXTENTS. You can emulate owner, segment_name, segment_type, bytes, and blocks.

The following code produces the result similar to DBA_EXTENTS:

```
insert #extentinfo

exec( '

dbcc extentinfo ( 0 ) with tableresults

' )



select

   UPPER(s.name) AS owner,

   UPPER(t.name) AS object_name,

   'TABLE' AS segment_type,

   ext_size*8192 as bytes,

   ext_size as blocks

  from #extentinfo AS e, sys.tables AS t, sys.schemas AS s
```

```
WHERE t.schema_id = s.schema_id

  AND e.obj_id = t.object_id

UNION ALL

select

   UPPER(s.name) AS owner,

   UPPER(i.name) AS object_name,

   'INDEX' AS segment_type,

   ext_size*8192 as bytes,

   ext_size as blocks

  from #extentinfo AS e, sys.indexes AS i,

       sys.tables AS t, sys.schemas AS s

WHERE t.schema_id = s.schema_id

  AND i.object_id = t.object_id

  AND e.obj_id = t.object_id
```

Note that this emulation cannot be applied to Azure SQL DB as DBCC command 'extentinfo' is not supported in this version of SQL Server.

V$LOCKED_OBJECT System View

SSMA does not automatically convert V$LOCKED_OBJECT. You can emulate V$LOCKED_OBJECT data by using the following columns in SQL Server 2014: os_user_name, session_id, oracle_username, locked_mode.

The following view provides the emulation:

```
CREATE VIEW ssma_oracle.V$LOCK_OBJECT AS

SELECT

    s.hostname as OS_USER_NAME,

    s.spid as SESSION_ID,

    UPPER(u.name) as ORACLE_USERNAME,

    CASE

      WHEN d.request_mode = 'IX' THEN 3

      WHEN d.request_mode = 'IS' THEN 2
```

```
          WHEN d.request_mode = 'X' THEN 6
          WHEN d.request_mode = 'S' THEN 4
          ELSE 0
        END as LOCKED_MODE
      FROM sys.dm_tran_locks as d LEFT OUTER JOIN
           (master..sysprocesses as s LEFT OUTER JOIN sysusers as u
           ON s.uid = u.uid) ON d.request_session_id = s.spid
     WHERE resource_type = 'OBJECT' and request_mode NOT IN ('Sch-M',
    'Sch-S')
```

Note that this emulation cannot be applied to Azure SQL DB as reference to master..sysprocesses is not supported in this version of SQL Server.

DBA_FREE_SPACE System View

SSMA does not automatically convert DBA_FREE_SPACE. You can emulate it in SQL Server 2014 in the following columns: file_id, bytes, blocks.

The following code performs the emulation:

```
CREATE VIEW DBA_FREE_SPACE AS
SELECT
    a.data_space_id as FILE_ID,
    SUM(a.total_pages - a.used_pages)*8192 as BYTES,
    SUM(a.total_pages - a.used_pages) as BLOCKS
  FROM sys.allocation_units as a
 GROUP BY a.data_space_id
```

Note that this emulation cannot be applied to Azure SQL DB as reference to sys.allocation_units is not supported in this version of SQL Server.

DBA_SEGMENTS System View

SSMA does not automatically convert the DBA_SEGMENTS view. You can emulate it in SQL Server 2014 with the following columns: owner, segment_name, segment_type, bytes.

We propose the following emulation:

```
CREATE VIEW ssma_oracle.DBA_SEGMENTS AS

SELECT

   UPPER(s.name) AS owner,

   UPPER(o.name) AS SEGMENT_NAME,

   'TABLE' AS SEGMENT_TYPE,

   SUM(a.used_pages*8192) as BYTES

  FROM sys.tables AS o INNER JOIN

       sys.schemas AS s ON s.schema_id = o.schema_id left join

       (sys.partitions as p join sys.allocation_units a on
p.partition_id = a.container_id

           left join sys.internal_tables it on p.object_id =
it.object_id)

       on o.object_id = p.object_id

WHERE     (o.is_ms_shipped = 0)

GROUP BY s.name, o.name

UNION ALL

SELECT

   UPPER(s.name) AS owner,

   UPPER(i.name) AS SEGMENT_NAME,

   'INDEX' AS OBJECT_TYPE,

   SUM(a.used_pages*8192) as BYTES

FROM sys.indexes AS i INNER JOIN

     sys.objects AS o ON i.object_id = o.object_id and

                         o.type = 'U' INNER JOIN

     sys.schemas AS s ON o.schema_id = s.schema_id left join

       (sys.partitions as p join sys.allocation_units a on
p.partition_id = a.container_id

           left join sys.internal_tables it on p.object_id =
it.object_id)
```

```
        on o.object_id = p.object_id
GROUP BY s.name, i.name
```

Note that this emulation cannot be applied to Azure SQL DB as reference to sys.allocation_units, sys.partitions and sys.internal_tables is not supported in this version of SQL Server.

Converting Oracle System Functions

SSMA converts Oracle system functions to either SQL Server system functions or to user-defined functions from the Microsoft Extension Library for SQL Server. The library is created in the ssma_oracle schema when you convert your database. The following table lists the Oracle system functions and SQL Server mappings.

Function conversion status (S)	Type of conversion (T)
Y: The function is fully converted.	M: Using standard Transact-SQL mapping.
P: The function is partially converted.	F: Using database user-defined functions.

Note: The prefix [ssma_oracle] is placed before functions in the ssma_oracle schema, as required for SQL Server functions that are part of the SSMA conversion.

Oracle System Function	S	T	Conversion to SQL Server	Comment
ABS(p1)	Y	M	ABS(p1)	
ACOS(p1)	Y	M	ACOS(p1)	
ADD_MONTHS(p1, p2)	Y	M	DATEADD(m, p2, p1)	
ASCII(p1)	Y	M	ASCII(p1)	
ASIN(p1)	Y	M	ASIN(p1)	
AVG(p1)	Y	M	AVG(p1)	
ATAN(p1)	Y	M	ATAN(p1)	
BITAND(p1, p2)	Y	F	ssma_oracle.BITAND(p1, p2)	
CAST(p1 AS t1)	Y	M	CAST(p1 AS t1)	
CEIL(p1)	Y	M	CEILING(p1)	

Oracle System Function	S	T	Conversion to SQL Server	Comment
CHR(p1 [USING NCHAR_CS])	P	M	CHAR(p1)	USING NCHAR_CS is currently not supported.
COALESCE(p1, …)	Y	M	COALESCE(p1, …)	
CONCAT(p1, p2)	Y	M	Into expression (p1 + p2) or CONCAT(p1, p2)	CONCAT function performs a concatenation of values, allowing for NULL values
COS(p1)	Y	M	COS(p1)	
COSH(p1)	Y	F	ssma_oracle.COSH(p1) no spaces are allowed in ssma_oracle user name.	
COUNT(p1)	Y	M	COUNT(p1)	
CUME_DIST()	Y	M	CUME_DIST( ) OVER ([partition_by_clause] order_by_clause)	
CURRENT_DATE	P	M	SYSDATETIME()	Limitation: CURRENT_DATE returns date in the time zone of DB session, but SYSDATETIME() returns date on SQL Server instance machine
DECODE(p1, p2, p3 [, p4])	Y	M	CASE p1 WHEN p2 THEN p3 [ELSE p4] END	
DENSE_RANK()	Y	M	DENSE_RANK()	
EXP(p1)	Y	M	EXP(p1)	
EXTRACT(p1 FROM p2)	P	M	DATEPART(part-p1, p2)	Only p1 = (YEAR, MONTH, DAY, HOUR, MINUTE, SECOND) is converted. For p1 = (TIMEZONE_HOU

Oracle System Function	S	T	Conversion to SQL Server	Comment
				R, TIMEZONE_MINUTE, TIMEZONE_REGION, TIMEZONE_ABBR) a message is generated saying that it is impossible to convert.
FIRST_VALUE()	Y	M	FIRST_VALUE ([scalar_expression]) OVER ([partition_by_clause] order_by_clause [rows_range_clause])	
FLOOR(p1)	Y	M	FLOOR(p1)	
FROM_TZ(p1, p2)	Y	M	**TODATETIMEOFFSET(p1, p2)**	
GREATEST(p1,p2 [,p3…pn])	P	F	ssma_oracle. GREATEST_DATETIME(p1, p2) GREATEST_FLOAT(p1, p2) GREATEST_INT(p1, p2) GREATEST_NVARCHAR(p1, p2) GREATEST_REAL(p1, p2) GREATEST_VARCHAR(p1, p2)	Function type is based on the p1 data type. If the Oracle source is GREATEST(p1,p2, p3), SSMA transforms it as GREATEST(p1, GREATEST(p2,p3)) and so on.
INITCAP(p1)	Y	F	ssma_oracle.	Function type is

Oracle System Function	S	T	Conversion to SQL Server	Comment
			INITCAP _VARCHAR(p1) INITCAP _NVARCHAR(p1)	based on the p1 data type. Currently supports the following argument types: CHAR, NCHAR, VARCHAR2, NVARCHAR2. For other types, a message is generated.

INSTR(p1,p2[,p3,p4])	P	F	ssma_oracle.
			
			INSTR2_CHAR(p1, p2)
			
			INSTR2_NCHAR(p1, p2)
			
			INSTR2_NVARCHAR(p1, p2)
			
			INSTR2_VARCHAR(p1, p2)
			
			INSTR3_CHAR(p1, p2, p3)
			
			INSTR3_NCHAR(p1, p2, p3)
			
			INSTR3_NVARCHAR(p1, p2, p3)
			
			INSTR3_VARCHAR(p1, p2, p3)
			
			INSTR4_CHAR(p1, p2, p3, p4)
			
			INSTR4_NCHAR(p1, p2, p3, p4)
			
			INSTR4_NVARCHAR(p1, p2, p3, p4)
			
			INSTR4_VARCHAR(p1, p2, p3, p4)
				INSTRB, INSTRC, INSTR2, INSTR4 currently not converted.

LAG()	Y	M	LAG (scalar_expression [,offset] [,default]) OVER ([partition_by_clause] order_by_clause)	
LAST_DAY(p1)	Y	M	EOMONTH (start_date [, month_to_add])	If you do not need time part in result it

Oracle System Function	S	T	Conversion to SQL Server	Comment
			ssma_oracle.LAST_DAY(p1)	is better to use this built-in function. Or
LAST_VALUE()	Y	M	LAST_VALUE ([scalar_expression) OVER ([partition_by_clause] order_by_clause rows_range_clause)	
LEAD()	Y	M	LEAD (scalar_expression [,offset] , [default]) OVER ([partition_by_clause] order_by_clause)	
LEAST(p1, p2 [, p3 … pn])	P	F	ssma_oracle. LEAST_DATETIME (p1, p2) LEAST_FLOAT (p1, p2) LEAST_INT (p1, p2) LEAST_NVARCHAR (p1, p2) LEAST_REAL (p1, p2) LEAST_VARCHAR (p1, p2)	Function type is based on the p1 data type. If Oracle source is LEAST (p1,p2,p3), SSMA transforms it as LEAST (p1, LEAST (p2,p3)) and so on.

LENGTH(p1)	P	F	ssma_oracle.
			LENGTH_CHAR(p1)
			LENGTH_NCHAR(p1)
				LENGTHB, LENGTHC, LENGTH2, LENGTH4 currently not converted.
				Function type determined based on the p1 data type.

Oracle System Function	S	T	Conversion to SQL Server	Comment

			LENGTH_NVARCHAR(p1)
			
			LENGTH_VARCHAR(p1)	

LN(p1)	Y	M	LOG(p1)	
LOCALTIMESTAMP	Y	M	SYSDATETIME()	
LOG(p1, p2)	Y	F	ssma_oracle.LOG_ANYBASE(p1, p2)	
LOWER(p1)	Y	M	LOWER(p1)	

LPAD(p1, p2)	Y	F	ssma_oracle.
			
			LPAD_VARCHAR(p1, p2, p3)
			
			LPAD_NVARCHAR(p1, p2, p3)
				Function type is based on the p1 data type. P3 = ‘ ’ (by default). Currently supports the following argument types: CHAR, NCHAR, VARCHAR2, NVARCHAR2. For other types a message is generated.

LPAD(p1, p2, p3)	Y	F	ssma_oracle.
			
			LPAD_VARCHAR(p1, p2, p3)
			
			LPAD_NVARCHAR(p1,p2,p3)
				Function type is based on the p1 data type. Currently supports the following argument types: CHAR, NCHAR, VARCHAR2, NVARCHAR2.

LTRIM(p1)	Y	M	LTRIM(p1)	

LTRIM(p1, p2)	Y	F	ssma_oracle.
			
			LTRIM2_VARCHAR(p1, p2)
			
			LTRIM2_NVARCHAR(p1, p2)
				Function type is based on the p1 data type. Currently supports the following argument types: CHAR,

Oracle System Function	S	T	Conversion to SQL Server	Comment
				NCHAR, VARCHAR2, NVARCHAR2.
MOD(p1, p2)	Y	M	Into expression (p1 % p2)	No check of parameter data types.
MONTHS_BETWEEN (p1, p2)	Y	M	DATEDIFF(MONTH, CAST(p2 AS float), CAST(DATEADD(DAY, (- CAST(DATEPART(DAY, p2) AS float(53)) + 1), p1) AS float))	
NEXT_DAY (p1, p2)	Y	F	ssma_oracle.NEXT_DAY (p1, p2)	
NEW_TIME(p1, p2, p3)	Y	F	ssma_oracle.NEW_TIME(p1, p2, p3)	
NLS_INITCAP(p1[, p2])	P	F	ssma_oracle. NLS_INITCAP_NVARCHAR(p1)	Only function calls with one argument are currently supported. The type of function is determined by the first argument data type. The following data types of the first argument are currently supported: NCHAR, NVARCHAR2. For other data types a message is generated.
NTILE()	Y	M	NTILE()	
NULLIF(p1, p2)	Y	M	NULLIF(p1, p2)	
NVL(p1, p2)	Y	M	ISNULL(p1, p2)	
PERCENTILE_DISC()	Y	M	PERCENTILE_DISC (numeric_literal) WITHIN GROUP (ORDER BY	

Oracle System Function	S	T	Conversion to SQL Server	Comment
			order_by_expression [ASC | DESC]) OVER ([<partition_by_clause>])	
PERCENT_RANK()	Y	M	PERCENT_RANK( ) OVER ([partition_by_clause] order_by_clause)	
PERCENTILE_CONT( )	Y	M	PERCENTILE_CONT (numeric_literal) WITHIN GROUP (ORDER BY order_by_expression [ASC | DESC]) OVER ([<partition_by_clause>])	
POWER(p1,p2)	Y	M	POWER(p1,p2)	
RANK()	Y	M	RANK()	
RAWTOHEX (p1)	Y	F	ssma_oracle.RAWTOHEX_VARCHAR (p1)	varchar is supported as the returned value type.
REMINDER (n2, n1)	Y	F		n2 - (n1*round(cast(n2 as float)/cast(n1 as float), 0))

REPLACE(p1, p2)
 	
REPLACE(p1, p2, p3)
	P	M	REPLACE(p1, p2 , ‘’)
			
			REPLACE(p1, p2 , p3)	

ROUND(p1) [p1 date] ROUND(p1, p2) [p1 date]	Y	F	ssma_oracle.ROUND_DATE (p1, NULL) ssma_oracle.ROUND_DATE (p1, p2)	
ROUND(p1) [p1 numeric]	Y	F	ssma_oracle.ROUND_NUMERIC_0 (p1)	
ROUND (p1, p2) [p1	Y	M	ROUND (p1, p2)	

Oracle System Function	S	T	Conversion to SQL Server	Comment
numeric]				
ROW_NUMBER()	Y	M	ROW_NUMBER()	

RPAD(p1, p2)	Y	F	ssma_oracle.
			
			RPAD_VARCHAR(p1, p2, p3)
			
			RPAD_NVARCHAR(p1, p2, p3)
				The type of function is determined by the first argument data type. P3 = ‘ ’ (by default). The following data types of the first argument are currently supported: CHAR, NCHAR, VARCHAR2, NVARCHAR2. For other data types a message is generated.

RPAD(p1, p2, p3)	Y	F	ssma_oracle.
			
			RPAD_VARCHAR(p1, p2, p3)
			
			RPAD_NVARCHAR(p1,p2,p3)
				The type of function is determined by the first argument data type. The following data types of the first argument currently supported: CHAR, NCHAR, VARCHAR2, NVARCHAR2. For other data types a message is generated

RTRIM(p1)	Y	M	RTRIM(p1)	

RTRIM(p1,p2)	Y	F	ssma_oracle.
			
			RTRIM2_VARCHAR(p1,p2)
			
			RTRIM2_NVARCHAR(p1,p2)
				The function type is based on the p1 data type. Currently supported following argument types are: CHAR, NCHAR,

Oracle System Function	S	T	Conversion to SQL Server	Comment
				VARCHAR2, NVARCHAR2.
SIGN(p1)	Y	M	SIGN(p1)	
SIN(p1)	Y	M	SIN(p1)	
SINH(p1)	Y	F	ssma_oracle.SINH(p1)	
SQRT(p1)	Y	M	SQRT (p1)	

SUBSTR(p1, p2[, p3])	P	F	ssma_oracle.
			
			SUBSTR2_CHAR(p1,p2)
			
			SUBSTR2_NCHAR(p1,p2)
			
			SUBSTR2_NVARCHAR(p1,p2)
			
			SUBSTR2_VARCHAR(p1,p2)
			
			SUBSTR3_CHAR(p1,p2,p3)
			
			SUBSTR3_NCHAR(p1,p2,p3)
			
			SUBSTR3_NVARCHAR(p1,p2,p3)
			
			SUBSTR3_VARCHAR(p1,p2,p3)
				The function type is based on the p1 data type.

SUM()	Y	M	SUM()	
SYS_GUID()	P	M	NEWID()	Not guaranteed to work correctly. For example, SELECT SYS_GUID() from dual differs from SELECT NEWID().
SYSDATE	Y	M	SYSDATETIME()	
SYSTIMESTAMP	Y	M	SYSDATETIMEOFFSET()	
TAN(p1)	Y	M	TAN(p1)	
TANH(p1)	Y	F	ssma_oracle.TANH(p1)	
TO_CHAR(p1)	Y	M	CAST(p1 AS CHAR)	Not guaranteed to

Oracle System Function	S	T	Conversion to SQL Server	Comment
				work correctly.
TO_CHAR(p1, p2)	Y	M	FORMAT (value, format [, culture]) ssma_oracle. TO_CHAR_DATE (p1, p2) TO_CHAR_NUMERIC (p1, p2)	For the overwhelming majority of cases it is better to use the built-in function. Or p1 can have date or numeric type. Formats currently not supported are E, EE, TZD, TZH, TZR. Allowable numeric formats are comma, period, ‘0’, ‘9,’ and ‘fm.’ Character value of p1 is not supported.

TO_DATE(p1)
 	
TO_DATE(p1, p2)
	P	F	CAST(p1 AS datetime)
			
			ssma_oracle.TO_DATE2 (p1, p2)
				Only 1- or 2-argument format is converted.

TO_NUMBER(p1[, p2[, p3]])	P	M	CAST(p1 AS NUMERIC)	Currently supported with only one argument. The conversion is not guaranteed to be fully equivalent.

TRANSLATE(p1, p2, p3)	Y	F	ssma_oracle.
			
			TRANSLATE_VARCHAR(p1, p2, p3)
			
			TRANSLATE_NVARCHAR(p1, p2, p3)
				The type of function is determined by the first argument data type. The following data types of the first argument are currently supported: CHAR, NCHAR, VARCHAR2, NVARCHAR2. For other data types a

Oracle System Function	S	T	Conversion to SQL Server	Comment
				message is generated.

TRUNC(p1[, p2])	Y	F	ssma_oracle.
			
			TRUNC(p1[, p2])
			
			TRUNC_DATE(p1)
			
			TRUNC_DATE2(p1, p2)
				Currently supported only for p1 of NUMERIC and DATE types.

TRIM	Y	F	ssma_oracle.TRIM2, ssma_oracle.TRIM3	The parameters are transformed.
UID	P	M	SUSER_SID()	The conversion is not guaranteed to be fully equivalent.
UPPER(p1)	Y	M	UPPER(p1)	
USER	Y	M	SESSION_USER	
WIDTH_BUCKET(p1, p2, p3, p4)	Y	F	ssma_oracle.WIDTH_BUCKET(p1, p2, p3, p4)	

Note that the following functions are not supported on Azure SQL DB: CUME_DIST, LAG, LEAD, FIRST_VALUE, LAST_VALUE, PERCENTILE_DISC, PERCENTILE_RANK, PERCENTILE_COST.

Converting Oracle System Packages

This section covers the migration of commonly used subroutines in Oracle standard packages. Some of the modules are migrated automatically by SSMA, and some should be handled manually. Examples illustrate our approach for the conversion.

DBMS_SQL Package

SSMA automatically covers cases where the statement is not SELECT. The dynamic SQL is processed manually.

Oracle Function or Procedure	Conversion to SQL Server	Comment
OPEN_CURSOR()	[ssma_oracle].DBMS_SQL_OPEN_CURSOR	The conversion is not guaranteed to be fully equivalent.
PARSE(p1,p2,p3)	[ssma_oracle].DBMS_SQL_PARSE p1,p2,p3	The conversion is not guaranteed to be fully equivalent.
EXECUTE(p1)	[ssma_oracle].DBMS_SQL_EXECUTE -p1	The conversion is not guaranteed to be fully equivalent.
CLOSE_CURSOR(p1)	[ssma_oracle].DBMS_SQL_CLOSE_CURSOR -p1	The conversion is not guaranteed to be fully equivalent.

Example:

Oracle

```
declare

   cur int;

   ret int;

begin

    cur :=  dbms_sql.open_cursor();

    dbms_sql.parse(cur, ' select col1 from t1', dbms_sql.NATIVE);

    ret := dbms_sql.execute(cur);

    dbms_sql.close_cursor(cur);
```

```
end;
```

SQL Server

```
Declare
  @cur numeric(38),
  @ret numeric(38)
begin
   EXECUTE ssma_oracle.dbms_sql_open_cursor @result = @cur OUTPUT
   EXECUTE ssma_oracle.dbms_sql_parse @cur, 'SELECT t1.col1 FROM
dbo.t1'
   EXECUTE ssma_oracle.dbms_sql_execute @cur,
@ssma$rows_processed = @ret  OUTPUT
   EXECUTE ssma_oracle.dbms_sql_close_cursor @cur
End
```

Conversion of DBMS_SQL Package to Azure SQL DB

There is a pequliarity of DBMS_SQL package conversion to Azure SQL DB as the latest doesn't support the usage of xp_ora2ms_exec extended stored procedure.

This procedure is used when Oracle DBMS_SQL.FETCH_ROWS function is converted to ssma_oracle.DBMS_SQL_FETCH_ROWS function. As there are DML operations are to be performed by the function and SQL Server doesn't allow DML operations in function code, it is rewritten as ssma_oracle.DBMS_SQL_FETCH_ROWS$IMPL stored procedure. Thus, conversion of DBMS_SQL.FETCH_ROWS function is implemented both as a procedure and a function. In this case, the procedure is used in a call via an extended procedure in the function body.

In Azure SQL DB, we have to get rid of this function and use implementation procedure directly.

Depending on the usage of DBMS_SQL.FETCH_ROWS function, SSMA performs two approaches in the conversion to Azure SQL DB:

- Changes calling code to use implementation procedure instead of the function;
- Marks calling code with error message.

In case SSMA marked the call to procedure with error, it is suggested to rewrite it to ssma_oracle.DBMS_SQL_FETCH_ROWS$IMPL procedure call manually where possible.

DBMS_OUTPUT Package

SSMA can handle commonly used PUT_LINE functions.

Oracle function or procedure	T	Conversion to SQL Server	Comment
PUT_LINE(p1)	M	PRINT p1	The conversion is not guaranteed to be fully equivalent.

Example:

Oracle

```
declare

    tname varchar2(255);

begin

    tname:='Hello, world!';

    dbms_output.put_line(tname);

end;
```

SQL Server

```
DECLARE

    @tname varchar(255)

BEGIN

    SET @tname = 'Hello, world!'

    PRINT @tname

END
```

UTL_FILE Package

The following table lists the UTL_FILE subprograms that SSMA processes automatically.

Oracle function or procedure	T	Conversion to SQL Server	Comment
IS_OPEN(p1)	S	UTL_FILE_IS_OPEN(p1)	
FCLOSE(p1)	S	UTL_FILE_FCLOSE p1	
FFLUSH (p1)	S	UTL_FILE_FFLUSH p1	
FOPEN (p1,p2,p3, p4)	S	UTL_FILE_FOPEN$IMPL(p1,p2,p3,p4,p5)	p5 return value
GET_LINE	S	UTL_FILE_GET_LINE(p1,p2,p3)	p2 return value
PUT	S	UTL_FILE_PUT(p1,p2)	
PUTF(p1, p2)	S	UTL_FILE_PUTF(p1,p2)	
PUT_LINE	S	UTL_FILE_PUT_LINE(p1,p2)	

Example:

Oracle

```
DECLARE

   outfile  utl_file.file_type;

   my_world varchar2(4) := 'Zork';

   V1 VARCHAR2(32767);

Begin

   outfile := utl_file.fopen('USER_DIR','1.txt','w',1280);

   utl_file.put_line(outfile,'Hello, world!');


   utl_file.PUT(outfile, 'Hello, world NEW! ');


   UTL_FILE.FFLUSH (outfile);
   IF utl_file.is_open(outfile) THEN
     Utl_file.fclose(outfile);
   END IF;

   outfile := utl_file.fopen('USER_DIR','1.txt','r');
   UTL_FILE.GET_LINE(outfile,V1,32767);
   DBMS_OUTPUT.put_line('V1= '||V1);
   IF utl_file.is_open(outfile) THEN
     Utl_file.fclose(outfile);
   END IF;
End write_log_file;
```

SQL Server

```
DECLARE
      @outfile XML,
      @my_world varchar(4),
      @V1 varchar(max)

    SET @my_world = 'Zork'
BEGIN

      EXEC ssma_oracle.UTL_FILE_FOPEN$IMPL 'USER_DIR', '1.txt',
'w', 1280, @outfile OUTPUT

      EXEC ssma_oracle.UTL_FILE_PUT_LINE @outfile, 'Hello,
world!'

      EXEC ssma_oracle.UTL_FILE_PUT @outfile, 'Hello, world NEW!
'
```

```
      EXEC ssma_oracle.UTL_FILE_FFLUSH @outfile


      IF (ssma_oracle.UTL_FILE_IS_OPEN(@outfile) != /* FALSE */
0)
        EXEC ssma_oracle.UTL_FILE_FCLOSE @outfile


      EXEC ssma_oracle.UTL_FILE_FOPEN$IMPL 'USER_DIR', '1.txt',
'r', 1024, @outfile OUTPUT


      EXEC ssma_oracle.UTL_FILE_GET_LINE @outfile, @V1 OUTPUT,
32767


      PRINT ('V1= ' + isnull(@V1, ''))


      IF (ssma_oracle.UTL_FILE_IS_OPEN(@outfile) != /* FALSE */
0)
        EXEC ssma_oracle.UTL_FILE_FCLOSE @outfile
END
```

Note that this code is not applicable on Azure SQL DB as this version of SQL Server doesn’t support working with file system.

DBMS_UTILITY Package

SSMA supports only the GET_TIME function.

Oracle function or procedure	T	Conversion to SQL Server	Comment
GET_TIME	M	SELECT CONVERT(NUMERIC(38, 0), (CONVERT(NUMERIC(38, 10), getdate()) * 8640000))	

DBMS_SESSION Package

SSMA supports only the UNIQUE_SESSION_ID function.

Oracle function or procedure	T	Conversion to SQL Server	Comment
UNIQUE_SESSION_ID	M	ssma_oracle.unique_session_id()	Return value is different

DBMS_PIPE Package

SSMA for Oracle V6.0 does not convert the DBMS_PIPE system package. To emulate it manually, follow these suggestions.

The DBMS_PIPE package has the following subprograms:

- function Create_Pipe()
- procedure Pack_Message()
- function Send_Message()
- function Receive_Message()
- function Next_Item_Type()
- procedure Unpck_Message()
- procedure Remove_Pipe()
- procedure Purge()
- procedure Reset_Buffer()
- function Unique_Session_Name()

Use a separate table to store data that is transferred via pipe.

Here's an example:

```
CREATE TABLE ssma_oracle.Pipes(

ID Bigint Not null Identity(1, 1),

PipeName Varchar(128) Not Null Default 'Default',

DataValue Varchar(8000)

);
```

```
GO

GRANT SELECT, INSERT, DELETE ON ssma_oracle.Pipes TO PUBLIC

GO
```

The pack-send and receive-unpack commands are usually used in pairs. Therefore, you can do the following replacement:

Oracle

```
      s := dbms_pipe.receive_message('<Pipe_Name>');

      if s = 0 then

         dbms_pipe.unpack_message(chr);

      end if;
```

SQL Server

```
DECLARE

          @s bigint,

          @chr varchar(8000)

BEGIN

          SET @chr = ''

          Select @s = Min(ID) from ssma_oracle.Pipes where
PipeName = '<Pipe_Name>'

          If @s is not null

          Begin

            Select @chr = DataValue From ssma_oracle.Pipes where
ID = @s

            Delete From ssma_oracle.Pipes where ID = @s

          End

END
```

Oracle

```
dbms_pipe.pack_message(info);

status := dbms_pipe.send_message('<Pipe_Name>');
```

SQL Server

```
INSERT INTO ssma_oracle.Pipes (PipeName, DataValue) Values
('<Pipe_Name>', @info)
```

Note that in order this emulation work on Azure SQL DB, you should create clustered index on ssma_oracle.Pipes table. Thus, the code for creating the pipes table will be as follow on Azure SQL DB:

```
CREATE TABLE ssma_oracle.Pipes(

ID Bigint Not null Identity(1, 1),

PipeName Varchar(128) Not Null Default 'Default',

DataValue Varchar(8000)

);

GO

CREATE CLUSTERED INDEX idxc_pipes ON ssma_oracle.Pipes (ID)

GO

GRANT SELECT, INSERT, DELETE ON ssma_oracle.Pipes TO PUBLIC

GO
```

Here are some considerations for the package conversion:

- Create_Pipe(). Can be ignored.
- Pack_Message(), Unpack_Message(). Add storage as a buffer or ignore.
- Send_Message(), Receive_Message(). Will be emulated as insert/select on the Pipes table (as shown in earlier example code).
- Next_Item_Type(). The system requires the addition of a **datatype** field to your Pipes table.
- Remove_Pipe() Emulate as Delete From Pipes where PipeName = '<PipeName>'

- Purge(). In our emulation, this means the same as Remove_Pipe().
- Reset_Buffer(). Needed if you emulate the buffer (and pack and unpack procedures).
- Unique_Session_Name(). Returns session name. It is possible to emulate it as SessionID.

DBMS_LOB Package

SSMA can automatically convert some functions of DBMS_LOB package. Their emulation is performed by SSMA procedures and functions, generated in ssma_oracle schema.

The following table lists the DBMS_LOB subprograms that SSMA processes automatically.

Oracle function or procedure	T	Conversion to SQL Server	Comment
DBMS_LOB.READ	S	ssma_oracle.dbms_lob$read_blob ssma_oracle.dbms_lob$read_clob	
DBMS_LOB.WRITE	S	ssma_oracle.dbms_lob$write_blob ssma_oracle.dbms_lob$write_clob	
DBMS_LOB.WRITEAPPEND	S	ssma_oracle.dbms_lob$writeappend blob ssma_oracle.dbms_lob$writeappend clob	
DBMS_LOB.GETLENGTH	S	ssma_oracle.dbms_lob$getlength_blob ssma_oracle.dbms_lob$getlength_clob	-
DBMS_LOB.SUBSTR	S	ssma_oracle.dbms_lob$substr_blob ssma_oracle.dbms_lob$substr_clob	-
DBMS_LOB.OPEN	S	This procedure is ignored during the conversion	
DBMS_LOB.CLOSE	S	This procedure is ignored during the conversion	

DBMS_JOB System Package

Both Oracle and SQL Server support jobs, but how they are created and executed is quite different. SSMA does not support conversion of the DBMS_JOB package, so this paper provides a description of manual conversion. The following example shows how to create the equivalent to an Oracle job in SQL Server. The subroutines are discussed below.

Submit a job to the job queue:

```
DBMS_JOB.SUBMIT (

<job_id> OUT binary_integer,

<what> IN varchar2,

<next_date> IN date DEFAULT sysdate,

<interval> IN varchar2 DEFAULT 'NULL',

<no_parse> IN boolean DEFAULT false,

<instance> IN DEFAULT any_instance,

<force> IN boolean DEFAULT false);
```

Remove a job from the queue:

```
DBMS_JOB.REMOVE (<job_id> IN binary_integer);
```

Where:

- <job_id> is the identifier of the job just created; usually it is saved by the program and used afterwards to reference this job (in a REMOVE statement).
- <what> is the string representing commands to be executed by the job process. To run it, Oracle puts this parameter into a BEGIN…END block, like this: BEGIN <what> END.
- <next_date> is the moment when the first run of the job is scheduled.
- <interval> is a string with an expression of DATE type, which is evaluated during the job run. Its value is the date + time of the next run.

The <instance> and <force> parameters are related to the Oracle clustering mechanism and we ignore them here. Also, we don’t convert the <no_parse> parameter, which controls when Oracle parses the command.

Note Convert the <what> and <interval> dynamic SQL strings independently. The important thing is to add the [database].[owner] qualifications to all object names that are referenced by this code. This is necessary because DB defaults are not effective during job execution.

Convert the SUBMIT and REMOVE routines into new stored procedures named DBMS_JOB_SUBMIT and DBMS_JOB_REMOVE, respectively. In addition, create a new special wrapper procedure _JOB_WRAPPER for implementing intime evaluations and scheduling the next run.

Note that Oracle and SQL Server use different identification schemes for jobs. In Oracle, the job is identified by sequential binary integer (job_id). In SQL Server, job identification is by **uniqueidentifier** job_id and by unique job name.

In our emulation scheme, we create three SQL Server stored procedures, which are described here.

DBMS_JOB_SUBMIT procedure

This SQL Server procedure creates a job and schedules its first execution. Find the full text of the procedure later in this section.

To submit a job in SQL Server:

1. Create a job and get its identifier by using **sp_add_job**.
2. Add an execution step to the job by using **sp_add_jobstep** (we use a single step).
3. Attach the job to the local server by using **sp_add_jobserver**.
4. Schedule the first execution by using **sp_add_jobschedule** (we use one-time execution at the specific time).

To save Oracle job information, store the Oracle <job_id> in the Transact-SQL *job_name* parameter and the <what> command as the job description. Because the job description is **nvarchar**(512), you cannot convert any command that is longer than 512 Unicode characters. The MS SQL identifier is generated automatically as job_id during execution of **sp_add_job**.

DBMS_JOB_REMOVE procedure

This procedure locates the SQL Server job ID by using the supplied Oracle job number, and it removes the job and all associated information by using **sp_delete_job**.

JOB_WRAPPER procedure

This procedure executes the job command and changes the job schedule so that the next run is set according to the <interval> parameter.

DBMS_JOB.SUBMIT

Convert a call to the SUBMIT procedure into the following SQL Server code:

```
EXEC DBMS_JOB_SUBMIT

    <job-id-ora> OUTPUT,

    <ms-command>,

    <next_date>,

    <interval>,

    <ora_command>
```

Where:

- <job-id-ora> is the Oracle-type job number; its declaration must be present in the source program.
- <ms-command> is the command in the source <what> parameter (dynamic SQL statement) that is converted to SQL Server independently. If the converted code contains several statements, divide them with semicolons (;). Because <ms-command> will run out of the current context (asynchronously inside of the_JOB_WRAPPER procedure), put all generated declarations into this string.
- <next_date> is the date of first scheduled run. Convert it as normal date expression.
- <interval> is the string with a dynamic SQL expression, which is evaluated at each job run to get the next execution date/time. Like <ms-command>, convert it to the corresponding SQL Server expression.
- <ora_command> is the parameter that is not present in Oracle format. This is the original <what> parameter without any changes. You save it for reference purposes.

Note that the <no_parse>, <instance>, and <force> parameters are not included in the converted statement. Instead the new <ora_command> item is used.

DBMS_JOB.REMOVE

Convert a call to the REMOVE procedure into the following code:

```
EXEC DBMS_JOB_REMOVE <job-id-ora>
```

<job-id-ora> is the Oracle-type number of the job that you want to delete. The source program must supply its declaration.

Example of an Oracle Job Conversion

This section contains a two-step example of a job conversion and the source of the new **ssma_oracle** procedures it references.

Step 1: Submit a job

Oracle PL/SQL

- Table the job will modify:

```
create table ticks (d date);
```

- Procedure executed at each step:

```
create or replace procedure ticker (curr_date date) as
begin
  insert into ticks values (curr_date);
  commit;
end;
```

- Job submitting:

```
declare j number;
        sInterval varchar2(50);
begin
  sInterval := 'sysdate + 1/8640';  -- 10 sec
  dbms_job.submit(job => j,
                  what => 'ticker(sysdate);',
                  next_date => sysdate + 1/8640, -- 10 sec
                  interval => sInterval);
  dbms_output.put_line('job no = ' || j);
end;
```

SQL Server

In this example, commands are executed by the **sa** user in a database called AUS:

```
USE AUS

GO
```

- Table the job will modify:

```
CREATE TABLE ticks (d datetime)

GO
```

- Procedure executed at each step:

```
CREATE PROCEDURE ticker (@curr_date datetime) AS

BEGIN

  INSERT INTO ticks VALUES (@curr_date);

END;

GO
```

- Job submitting:

```
declare @j float(53),

   @sInterval varchar(50)

begin

set @sInterval = 'getdate() + 1./8640'


/* parameter calculation is normally generated by the converter*/

declare @param_expr_0 datetime

set @param_expr_0 = getdate() + 1./8640  -- 10 sec


/* note AUS.DBO.ticker */

exec DBMS_JOB_SUBMIT

    @j OUTPUT,

    N'DECLARE @param_expr_1 DATETIME; SET @param_expr_1 =
getdate(); EXEC AUS.DBO.TICKER @param_expr_1',

    @param_expr_0,

    @sInterval,

    N'ticker(sysdate);' /* parameter to save the original command
*/

print 'job no = ' + cast (@j as varchar)

end

go
```

Step 2: Locate and remove a job

This solution uses emulation of the Oracle USER_JOBS system view, which can be generated by SSMA for Oracle V6.0.

Oracle

```
declare j number;
begin
  SELECT  job INTO  j
    FROM  user_jobs
   WHERE  (what = 'ticker(sysdate);');
  dbms_output.put_line(j);
  dbms_job.remove(j);
end;
```

SQL Server

```
declare @j float(53);

begin

  SELECT  @j = job

    FROM  USER_JOBS

   WHERE (what = 'ticker(sysdate);'); -- note Oracle expression left
here

  print @j

  exec DBMS_JOB_REMOVE @j

end
```

Source of new procedures

```
   ------------------------S U B M I T-------------------

   create procedure DBMS_JOB_SUBMIT (

      @p_job_id int OUTPUT,       -- Oracle job id

      @p_what nvarchar(4000),     -- command converted to SQL Server

      @p_next_date datetime,      -- date of the first run

      @p_interval nvarchar(4000),-- interval expression converted to
   SQL Server

      @p_what_ora nvarchar(512)  -- original Oracle command

   ) as
```

```
begin
declare @v_name nvarchar(512),
      @v_job_ora int,
      @v_job_ms  uniqueidentifier,
        @v_command nvarchar(4000),
        @v_buf varchar(40),
      @v_nextdate int,
      @v_nexttime int


-- 1. Create new job

select @v_job_ora =
  max(
   case isnumeric(substring(name,6,100))
     when 1 then cast(substring(name,6,100) as int)
     else 0
   end
 )
 from msdb..sysjobs
where substring(name,1,5)='_JOB_'

set @v_job_ora = isnull(@v_job_ora,0) + 1
set @v_name = '_JOB_' + cast(@v_job_ora as varchar(12))

exec msdb..sp_add_job
   @job_name = @v_name,
   @description = @p_what_ora,  -- saving non-converted Oracle
command for reference
   @job_id = @v_job_ms OUTPUT



-- 2. Add a job step

set @v_command = N'exec _job_wrapper  '''
         + cast(@v_job_ms as varchar(40)) + ''', N'''
         + @p_what + ''', N'''
         + @p_interval +''''
```

```
exec msdb..sp_add_jobstep
   @job_id = @v_job_ms,
   @step_name = N'oracle job emulation',
   @command = @v_command


-- 3. Attach to local server

exec msdb..sp_add_jobserver
   @job_id = @v_job_ms,
   @server_name = N'(LOCAL)'


-- 4. Make schedule for the first run

/* date format is YYYY-MM-DD hh:mm:ss */
set @v_buf = convert(varchar, @p_next_date, 20)
set @v_nextdate =
substring(@v_buf,1,4)+substring(@v_buf,6,2)+substring(@v_buf,9,2)
set @v_nexttime =
substring(@v_buf,12,2)+substring(@v_buf,15,2)+substring(@v_buf,18,2)

exec msdb..sp_add_jobschedule
   @job_id = @v_job_ms,
   @name = 'oracle job emulation',
   @freq_type = 1,
   @freq_subday_type = 1,
   @active_start_date = @v_nextdate,
   @active_start_time = @v_nexttime

end

go

-----------------------------R E M O V E----------------------------

create procedure DBMS_JOB_REMOVE (
   @p_job_id int     -- Oracle-style job id
)
```

```
as
begin
declare @v_job_id uniqueidentifier   -- SQL Server job id

select @v_job_id = job_id
  from msdb..sysjobs
where name = '_JOB_' + cast(@p_job_id as varchar(12))

if @v_job_id is not null
  exec msdb..sp_delete_job @v_job_id

end

go

--------------------------W R A P P E R----------------------------

create procedure _JOB_WRAPPER (
   @p_job_id_ms uniqueidentifier,
   @p_what nvarchar(512),
   @p_interval nvarchar(4000)
) as
begin
declare @v_command nvarchar(4000),
   @v_buf varchar(40),
   @v_nextdate int,
   @v_nexttime int


-- 1. Execute job command

execute (@p_what)

-- 2. Evaluate next run date

set @v_command =
  'set @buf = convert(varchar, ' + @p_interval + ', 20)'
```

```
exec sp_executesql @v_command, N'@buf varchar(40) output', @v_buf
output



-- 3. Redefine the schedule

/* ODBC date format: YYYY-MM-DD hh:mm:ss */
set @v_nextdate =
substring(@v_buf,1,4)+substring(@v_buf,6,2)+substring(@v_buf,9,2)
set @v_nexttime =
substring(@v_buf,12,2)+substring(@v_buf,15,2)+substring(@v_buf,18,2)

exec msdb..sp_update_jobschedule
   @job_id = @p_job_id_ms,
   @name = 'oracle job emulation',
   @enabled = 1,
   @freq_type = 1,
   @freq_subday_type = 1,
   @active_start_date = @v_nextdate,
   @active_start_time = @v_nexttime

end
```

Note that Azure SQL DB doesn’t support jobs and the above example is not subject to this version of SQL Server.

Converting Nested PL/SQL Subprograms

Oracle allows PL/SQL subprogram (procedure or function) definitions to be nested within another subprogram. These subprograms can be called only from inside the PL/SQL block or the subprogram in which they were declared. There are no special limitations for parameters or the functionality of nested procedures or functions. That means that any of these subprograms can in turn include other subprogram declarations, which makes multiple levels of nesting possible. In addition, the nested modules can be overloaded; that is, they can use the same name a few times with different parameter sets.

Microsoft SQL Server 2014 does not provide similar functionality. It is possible to create a stand-alone SQL Server procedure or function that emulates Oracle nested subprograms. But doing so presents the problem of how to handle local variables. In PL/SQL, a nested subprogram declared at level N has full access to all local variables declared at levels N, N-1, . . . 1. In SQL Server, the local declarations of other procedures are not visible.

SSMA can convert inline subprograms automatically. A **Type of local modules conversion** option is provided in Project Settings. You can adjust this option to convert local modules either inline or by creating a separate stored procedure.

Inline Substitution

If the type of local modules conversion is set to inline substitution, a nested module itself is not converted to any target object, but each call of the module is expanded to inline blocks in the outermost subprogram. The inline block is formed according to the following pattern:

```
<parameter_declaration>

<return_value_parameter_declaration>

<parameters_assignments>

<module_body>

<output_parameters_assignments>

<return_value_assignment>
```

Example 1

Oracle

```
create procedure Proc1 is

on_year int := 2000;
```

```
dept_sales int := 0;

 procedure DeptSales(dept_id int) is
 lv_sales int;
 procedure Add is
 begin
 dept_sales := dept_sales + lv_sales;
 end Add;
 procedure Add(i int) is
 begin
 dept_sales := dept_sales + i;
 end Add;

begin
select sales into lv_sales from departmentsales
where id = dept_id and year = on_year;
Add;
Add(200);
end DeptSales;
begin
DeptSales(100);
end Proc1;
```

SQL Server

```
CREATE  PROCEDURE  Proc1
CREATE PROCEDURE ATEST.PROC1
AS
   BEGIN
      DECLARE @on_year int
      SET @on_year = 2000

      DECLARE @dept_sales int

      SET @dept_sales = 0

      BEGIN

         DECLARE

            @DeptSales$dept_id int

         SET @DeptSales$dept_id = 100

         BEGIN

            DECLARE

               @DeptSales$lv_sales int

            SELECT @DeptSales$lv_sales = DEPARTMENTSALES.SALES

            FROM dbo.DEPARTMENTSALES

            WHERE DEPARTMENTSALES.ID = @DeptSales$dept_id AND
DEPARTMENTSALES.YEAR = @on_year

            BEGIN

               BEGIN

                  SET @dept_sales = @dept_sales +
@DeptSales$lv_sales

               END

            END

            BEGIN

               DECLARE @DeptSales$ADD$i int

               SET @DeptSales$ADD$i = 200

               BEGIN

                  SET @dept_sales = @dept_sales +
@DeptSales$ADD$i

               END

            END

         END

      END
```

```
   END
```

Example 2

To convert an output parameter, SSMA adds an assignment statement that saves the output value stored in the intermediate variable.

Oracle

```
create procedure Proc1 is

on_year int := 2000;

dept_sales int;

lv_out_sales int;


 procedure DeptSales(dept_id int, lv_sales out int) is

 begin

 select sales into lv_sales from departmentsales

 where id = dept_id and year = on_year;

 end DeptSales;


begin

DeptSales(dept_sales, lv_out_sales);

end Proc1;
```

SQL Server

```
CREATE PROCEDURE PROC1

AS

   BEGIN

      DECLARE @on_year int

      SET @on_year = 2000

      DECLARE

         @dept_sales int,
         @lv_out_sales int


      BEGIN

         DECLARE

            @DeptSales$dept_id int

         DECLARE

            @DeptSales$lv_sales int

         SET @DeptSales$dept_id = @dept_sales

         SET @DeptSales$lv_sales = @lv_out_sales

         BEGIN

            SET @DeptSales$lv_sales = NULL

            SELECT @DeptSales$lv_sales = DEPARTMENTSALES.SALES

            FROM dbo.DEPARTMENTSALES

            WHERE DEPARTMENTSALES.ID = @DeptSales$dept_id AND
DEPARTMENTSALES.YEAR = @on_year

         END

         SET @lv_out_sales = @DeptSales$lv_sales

      END

   END
```

Emulation by Using Transact-SQL Subprograms

If the **Type of local modules conversion** option is set to create a separate stored procedure, SSMA converts nested PL/SQL subprograms into separate stored procedures and functions with special naming rules. This is reasonable if you are working with large nested subprograms with a limited number of variables.

SSMA analyzes the original module and collects the following information:

- A list of all locally declared subroutines
- References of each nested subroutine to outer modules
- Calls of each nested module from other modules
- A list of the variables and parameters of outer modules used in each nested module

- The type of access to the external variables in a nested module—the type can be read/write or read-only

After that, SSMA creates a set of procedures that emulate Oracle nested modules and adds additional input/output parameters for access to external variables.

```
      SELECT @lv_sales = DEPARTMENTSALES.SALES

      FROM dbo.DEPARTMENTSALES

      WHERE DEPARTMENTSALES.ID = @dept_id AND
DEPARTMENTSALES.YEAR = @on_year
```

Example

In this example, the nested module calls another nested module that is defined at the same level. In this case, all external variables used in the caller module should also be passed to the called module.

Oracle

```
create procedure Proc1 is

on_year int := 2000;

dept_sales int;


 procedure DeptSales(dept_id int) is

 lv_sales int;

 begin

 select sales into lv_sales from departmentsales

 where id = dept_id and year = on_year;

 dept_sales := lv_sales;

 end DeptSales;


 procedure DeptSales_300 is

 begin

 DeptSales(300);

 end DeptSales_300;
```

```
begin
DeptSales(100);
DeptSales_300;
end Proc1;
```

SQL Server

```
CREATE PROCEDURE  Proc1$DeptSales
@dept_id int,
@on_year int,         -- Proc1.on_year
@dept_sales int OUTPUT     -- Proc1.dept_sales
AS
BEGIN
declare @lv_sales int
      SELECT @lv_sales = DEPARTMENTSALES.SALES
      FROM dbo.DEPARTMENTSALES
      WHERE DEPARTMENTSALES.ID = @dept_id AND
      DEPARTMENTSALES.YEAR = @on_year
      SET @dept_sales = @lv_sales
END
GO

CREATE  PROCEDURE  Proc1$DeptSales_300
@on_year int,         -- Proc1.on_year
@dept_sales int OUTPUT     -- Proc1.dept_sales
AS
BEGIN
Execute Proc1$DeptSales
300,
@on_year,
@dept_sales = @dept_sales OUTPUT

END

GO


CREATE  PROCEDURE  Proc1

AS

BEGIN

declare @on_year int

set @on_year = 2000

declare @dept_sales int

Execute Proc1$DeptSales

100,

@on_year,

@$dept_sales = @dept_sales OUTPUT

Execute Proc1$DeptSales_300

@on_year,

@$dept_sales = @dept_sales OUTPUT

END

GO
```

Migrating Oracle User-Defined Functions

This section describes how SSMA for Oracle V6.0 converts Oracle user-defined functions. While Oracle functions closely resemble Transact-SQL functions, significant differences do exist. The main difference is that Transact-SQL functions cannot contain DML statements and cannot invoke stored procedures. In addition, Transact-SQL functions do not support transaction-management commands. These are stiff restrictions. A workaround implements a function body as a stored procedure and invokes it within the function by means of an extended procedure. Note that some Oracle function features, such as output parameters, are not currently supported.

Conversion Algorithm

The general format of an Oracle user-defined function is:

```
FUNCTION [schema.]name [({@parameter_name [ IN | OUT | IN OUT ]

     [ NOCOPY ] [ type_schema_name. ] parameter_data_type  [:= |
DEFAULT] default_value } [ ,...n ]

 ) ]

   RETURN <return_data_type>

   [AUTHID {DEFINER | CURRENT_USER}]

   [DETERMINISTIC]

   [PARALLEL ENABLE ...]

   [AGGREGATE | PIPELINED]

{ IS | AS } { LANGUAGE { Java_declaration | C_declaration } | {


   [<declaration statements>]


BEGIN

   <executable statements>


RETURN <return statement>


[EXCEPTION

   exception handler statements]
END [ name ]; }}
```

And the proper Transact-SQL format of a scalar function is:

```
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ][ type_schema_name. ] parameter_data_type
    [ = default_value ] } [ ,...n ]
  ]
)
RETURNS <return_data_type>
    [WITH { EXEC | EXECUTE } AS { CALLER | OWNER }]
    [ AS ]
    BEGIN
        <function_body>
        RETURN <scalar_expression>
    END
[ ; ]
```

The following clauses and arguments are not supported by SSMA and are ignored during conversion:

- AGGREGATE
- DETERMINISTIC
- LANGUAGE
- PIPELINED
- PARALLEL_ENABLE

For the remaining function options, the following rules are applied during conversion:

- The OUT qualifier is used when a function is implemented as a procedure.
- The [:= | DEFAULT] option of a function parameter is converted to an equals sign (=).

- The AUTHID clause is converted to an EXECUTE AS clause.
- The CURRENT_USER argument is converted to a CALLER argument.
- The DEFINER argument is converted to an OWNER argument.

As a result of the conversion, you get one of the following:

- One Transact-SQL function body
- Two objects:
  - Implementation of a function in the form of a procedure
  - A function that is a wrapper for the procedure calling

Following are the conditions when this additional procedure is created:

- The source function is defined as an autonomous transaction by PRAGMA AUTONOMOUS_TRANSACTION.
- A function contains statements that are not valid in SQL Server user-defined functions, such as:
  - DML operations (UPDATE, INSERT, DELETE) that modify tables, except for local table variables
  - A call of a stored procedure
  - Transaction-management commands
  - The raise exception command
  - Exception-handling statements
  - FETCH statements that return data to the client
  - Cursor operations that reference global cursors

If any of these conditions are present, the function is implemented both as a procedure and a function. In this case, the procedure is used in a call via an extended procedure in the function body. The function body is implemented according to the following pattern:

```
CREATE FUNCTION [schema.] <function_name>

  (

        <parameters list>

)

RETURNS <return_type>

AS
```

```
BEGIN

        declare @spid int, @login_time datetime

  select @spid = ssma_oracle.get_active_spid(),@login_time = 
ssma_oracle.get_active_login_time()

        DECLARE

          @return_value_variable <function_return_type>

        EXEC master.dbo.xp_ora2ms_exec2_ex @@spid,@login_time, 
<database_name>, <schema_name>, 
<function_implementation_as_procedure_name>,

bind_to_transaction_flag, [parameter1, parameter2, ... ,] 
@return_value_variable OUTPUT

        RETURN @return_value_variable

END
```

The syntax of the **xp_ora2ms_exec2_ex** procedure is:

```
xp_ora2ms_exec2_ex

      <active_spid> int,

      <login_time> datetime,
      <ms_db_name> varchar,
      <ms_schema_name> varchar,
      <ms_procedure_name> varchar,
      <bind_to_transaction_flag> varchar,
      [optional_parameters_for_procedure]
```

Where:

- <active_spid> [input parameter] is the session ID of the current user process.
- <login_time> [input parameter] is the login time of the current user process.

- <ms_db_name> [input parameter] is the database name owner of the stored procedure.
- <ms_schema_name> [input parameter] is the schema name owner of the stored procedure.
- <ms_procedure_name> [input parameter] is the name of the stored procedure.
- <bind_to_transaction_flag> [input parameter] binds or unbinds a connection to the current transaction. Valid values are 'TRUE,' 'true,' 'Y,' 'y.' Other values are ignored.
- optional_parameters_for_procedure [input/output parameter] are the procedure parameters.

If PRAGMA AUTONOMOUS_TRANSACTION is used, the **xp_ora2ms_exec2_ex** procedure's bind to transaction parameter is set to true. Otherwise, it is set to false. For details about autonomous transactions, see Simulating Oracle Autonomous Transactions.

Azure SQL DB doesn't support extended stored procedures functionality and thus all the calls to the converted functions that use the **xp_ora2ms_exec2_ex** should be replaced with the calls to the corresponding implementation procedures where possible.

A function's procedure implementation is converted according to the following pattern:

```
CREATE PROCEDURE [schema.] <function_name>$IMPL

        <parameters list> ,

        @return_value_argument <function_return_type> OUTPUT

    AS

      BEGIN

     set implicit_transactions on  /*only in case of PRAGMA
AUTONOMOUS_TRANSACTION*/

      <function implementation>

      SET @return_value_argument = <return_expression>

RETURN

END
```

Where <return_expression> is an expression that a function uses in the RETURN operator. So, the RETURN statement in a function's procedure implementation is converted according to this pattern:

PL-SQL code

```
RETURN <return_expresion>;
```

Transact-SQL code

```
SET @return_value_argument = <return_expression>
RETURN
```

Convert multiple RETURNs in the same way:

PL-SQL code

```
...
IF <condition> THEN
      RETURN <return_expresion_1>;
ELSE
      RETURN <return_expresion_2>;
ENDIF
...
```

Transact-SQL code

```
...
IF <condition>
BEGIN
      SET @return_value_argument = <return_expression_1>
      RETURN
END
ELSE
BEGIN
```

```
      SET @return_value_argument = <return_expression_2>

      RETURN

END

...
```

Example

PL-SQL code

```
declare i int :=fn_test1();

begin

i:=fn_test2();

DBMS_OUTPUT.PUT_LINE(i);

end;
```

Transact-SQL code

```
DECLARE @i int
exec FN_TEST1$IMPL @i out

BEGIN

exec FN_TEST2$IMPL @i out

PRINT @i

END
```

Converting Function Calls When a Function Has Default Values for Parameters and with Various Parameter Notations

When calling functions in Oracle, you can pass parameters by using:

- Positional notation. Parameters are specified in the order in which they are declared in the procedure.
- Named notation. The name of each parameter is specified along with its value. An arrow (=>) serves as the association operator. The order of the parameters is not significant.

- Mixed notation. The first parameters are specified with positional notation, and then they are switched to named notation for the last parameters.

Because SQL Server does not support named notation for parameters that are passed to functions, the named notation is converted to the positional notation call. In addition, SQL Server functions do not support omitted parameters, so if the default parameters are omitted, the statement is converted by adding the keyword, **default**, instead of the omitted parameters.

Examples

PL-SQL code

```
CREATE OR REPLACE  FUNCTION fn_test (
p_1 VARCHAR2,
p_2 VARCHAR2 DEFAULT 'p_2',
p_3 VARCHAR2 DEFAULT 'p_3')

RETURN VARCHAR2 IS
BEGIN
  return null;
END;

/

select fn_test('p1') from dual;

declare a varchar2(50);
begin
a:= fn_test('p_1','hello','world');
a:= fn_test('p_1');
a:= fn_test('p_1',p_3=>'world');
a:= fn_test(p_2=>'hello',p_3=>'world',p_1=>'p_1');
end;
```

Transact-SQL code

```
CREATE FUNCTION fn_test (

@p_1 VARCHAR(max),

@p_2 VARCHAR(max)= 'p_2',

@p_3 VARCHAR(max)= 'p_3')

RETURNS  VARCHAR(max) as

BEGIN

  return null;

END;
```

```
GO

select dbo.fn_test('p1',default,default)

declare @a varchar(50)

begin

set @a = dbo.fn_test('p_1','hello','world')

set @a = dbo.fn_test('p_1', default, default)

set @a = dbo.fn_test('p_1',default, 'world')

set @a = dbo.fn_test('p_1','hello','world')

end;
```

PRAGMA INLINE

The INLINE pragma specifies that a subprogram call is, or is not, to be inlined. Inlining replaces a subprogram call (to a subprogram in the same program unit) with a copy of the called subprogram.

In this case it is necessary to remove this PL/SQL code.

Examples

PL-SQL code

```
create or replace function fn_Test return varchar2

as

begin

  RETURN 'fffff';

end fn_Test ;


create or replace procedure Test

as

   x varchar2(100) ;

begin

   PRAGMA INLINE(fn_TEST, 'YES');

   x:= fn_TEST() ;

   dbms_output.put_line(x);
   PRAGMA INLINE(fn_TEST, 'NO');

end Test ;
```

Transact-SQL code

```
CREATE FUNCTION dbo.FN_TEST() RETURNS varchar(max)

AS

BEGIN

   RETURN 'fffff'

END

GO

CREATE PROCEDURE dbo.TEST

AS

BEGIN

   DECLARE @x varchar(100)

   SET @x = dbo.FN_TEST()

   PRINT @x

END

GO
```

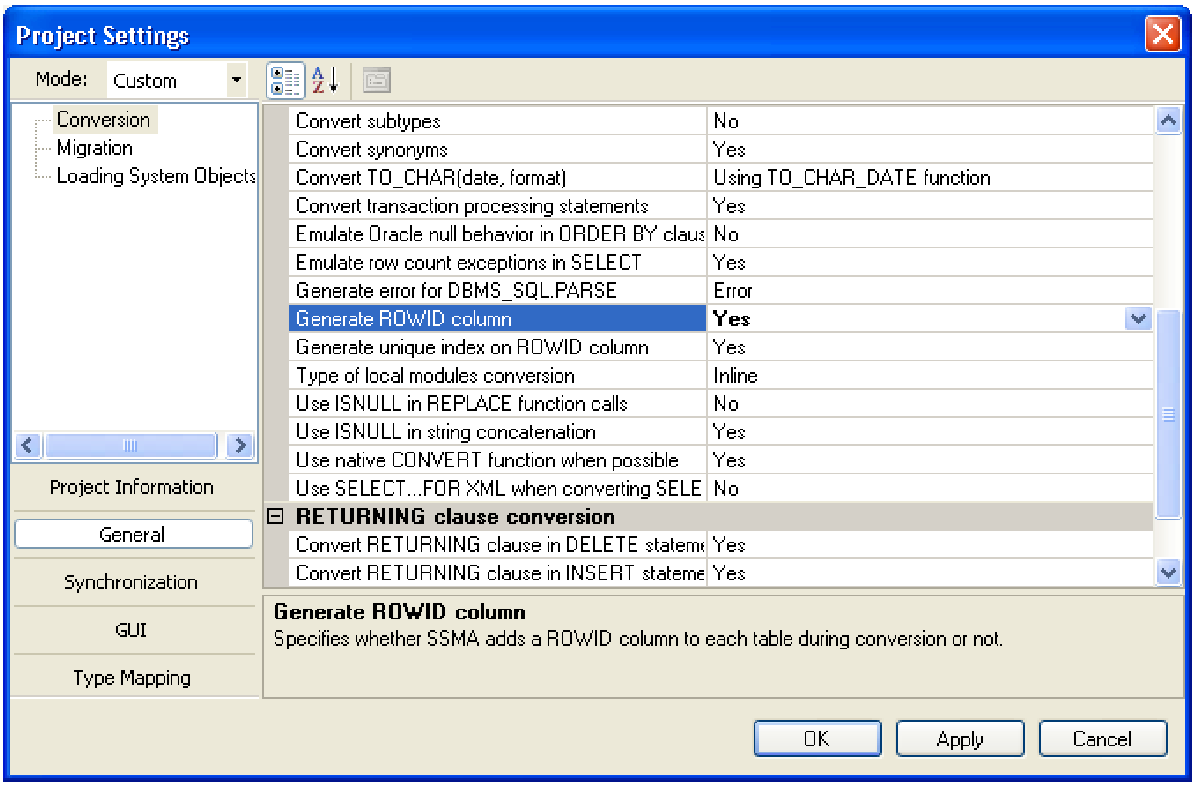
Migrating Oracle Triggers

This section describes the differences between Oracle and Microsoft SQL Server 2014 triggers, and how SSMA for Oracle V6.0 handles them when it converts Oracle triggers to SQL Server. (This section does not cover DDL or system triggers. The discussion is limited to DML triggers, that is, triggers on INSERT, UPDATE, or DELETE statements.)

The first major difference between Oracle and SQL Server triggers is that the most common Oracle trigger is a row-level trigger (FOR EACH ROW), which fires for each row of the source statement. SQL Server, however, supports only statement-level triggers, which fire only once per statement, irrespective of the number of rows affected.

In a row-level trigger, Oracle uses an *:OLD* alias to refer to column values that existed before the statement executes, and to the changed values by using a *:NEW* alias. SQL Server uses two pseudotables, **inserted** and **deleted**, which can each have multiple rows. If the triggering statement is UPDATE, a row's older version is present in **deleted**, and the newer in **inserted**. But it is not easy to tell which pair belongs to the same row if the updated table does not have a primary key or the primary key was modified.

You can resolve this problem only if SSMA generates a special ROWID column for the table. Therefore, if you are converting tables with UPDATE triggers, we recommend setting the **Generate ROWID column** option to **Yes** or **Add ROWID column for tables with triggers** in the SSMA project settings (See Figure 2). To emulate row-level triggers, SSMA processes each row in a cursor loop.



Convert subtypes	No
Convert synonyms	Yes
Convert TO_CHAR(date, format)	Using TO_CHAR_DATE function
Convert transaction processing statements	Yes
Emulate Oracle null behavior in ORDER BY claus	No
Emulate row count exceptions in SELECT	Yes
Generate error for DBMS_SQL.PARSE	Error
Generate ROWID column	**Yes**
Generate unique index on ROWID column	Yes
Type of local modules conversion	Inline
Use ISNULL in REPLACE function calls	No
Use ISNULL in string concatenation	Yes
Use native CONVERT function when possible	Yes
Use SELECT...FOR XML when converting SELE	No
RETURNING clause conversion	
Convert RETURNING clause in DELETE stateme	Yes
Convert RETURNING clause in INSERT stateme	Yes

Generate ROWID column
Specifies whether SSMA adds a ROWID column to each table during conversion or not.

Figure 2: Set up the Generate ROWID column option

The second major difference between Oracle and SQL Server triggers comes from Oracle BEFORE triggers. Because Oracle fires these triggers before the triggering statement, it is possible to modify the actual field values that will be stored in the table, or even cancel the execution of the triggering statement if it is found to be unnecessary. To emulate this in SQL Server, you must create INSTEAD OF triggers. That means you must incorporate the triggering statement into the target trigger's body. Because multiple rows can be affected, SSMA puts the statement in a separate cursor loop.

In some cases, you cannot convert Oracle triggers to SQL Server triggers with one-to-one correspondence. If an Oracle trigger is defined for several events at once (for example, INSERT or UPDATE), you must create two separate target triggers, one for INSERT and one for UPDATE. In addition, because SQL Server supports only one INSTEAD OF trigger per table, SSMA combines the logic of all BEFORE triggers on that table into a single target trigger. This means that triggers are not converted independently of each other; SSMA takes the entire set of triggers belonging to a table and converts them into another set of SQL Server triggers so that the general relation is many-to-many.

In brief, the conversion rules are:

- All BEFORE triggers for a table are converted into one INSTEAD OF trigger.
- AFTER triggers remain AFTER triggers in SQL Server.
- INSTEAD OF triggers on Oracle views remain INSTEAD OF triggers.
- Row-level triggers are emulated with a cursor loop.
- Triggers that are defined for multiple events are split into separate target triggers.

Sometimes an Oracle trigger is defined for a specific column with the UPDATE OF column [, column]...] clause. To emulate this, SSMA wraps the trigger body with the following SQL Server construction:

```
IF (UPDATE(column) [OR UPDATE(column) . . .]

BEGIN

  <trigger body>

END
```

SSMA emulates the trigger-specific functions performing INSERT, UPDATE, and DELETE operations by saving the current trigger type in a variable, and then checking that value. For example:

```
DECLARE @triggerType char(1)

SELECT @triggerType = 'I'    /* if the current type is inserting
*/

. . .

IF (@triggerType = 'I' ) . . .  /* emulation of INSERTING */

IF (@triggerType = 'U' ) . . .  /* emulation of UPDATING */

IF (@triggerType = 'D' ) . . .  /* emulation of DELETING */
```

The UPDATING function can have a column name as an argument. SSMA can convert such usage if the argument is a character literal. In this case, the Oracle expression:

```
UPDATING ('column_name')
```

Is transformed into:

```
UPDATE (columns_name)
```

Note that the original quotes are removed.

Conversion Patterns

This section illustrates the conversion algorithms SSMA uses to convert various types of Oracle triggers. Each example schematically outlines a particular type of trigger. Comments describe the typical contents of source triggers and the structure of the corresponding target triggers as generated by SSMA.

AFTER Triggers

Table-level triggers

Table-level AFTER triggers fire only once per table, resembling the behavior of SQL Server AFTER triggers. Thus, the required changes are minimal. Table-level triggers are converted according to this pattern:

```
CREATE TRIGGER [ schema. ]trigger ON <table>
    AFTER <UPDATE |INSERT | DELETE>
    AS
      /*  beginning of trigger implementation */
      SET NOCOUNT ON
------------------------------------------------------------------
      /* Oracle-trigger implementation: begin */
      BEGIN
      -- UPDATE OF CLAUSE FOR TRIGGER FOR UPDATE EVENT
      -- (UPDATE OF COLUMN[, COLUMN] ... ])
        IF (UPDATE(<COLUMN>) OR UPDATE((<COLUMN>) ...)
          BEGIN
            <TRIGGER_BODY>
          END
      END
      /* Oracle-trigger implementation: end */
-----------------------------------------------------------------------

      /*  end of trigger implementation */
```

Row-level triggers

Because Oracle Database fires a row-level trigger once for each row, emulate row-level triggers with cursor processing.

For row-level triggers, a restriction can be specified in the WHEN clause. The restriction is an SQL condition that must be satisfied for the database to fire the trigger. Also, the special variables :NEW and :OLD are available in row-level triggers to refer to new and old records respectively.

In SQL Server, the new and old records are stored in the **inserted** and **deleted** tables. So, row-level triggers are emulated in the same way as table-level ones, except for the trigger implementation wrapped into the cursor processing block.

Replace references to :OLD and :NEW values with values fetched into variables from deleted or updated tables, respectively.

Pattern for row-level AFTER INSERT triggers

```
    CREATE TRIGGER [ schema. ]trigger ON <table>
    AFTER INSERT
    AS
      /*  beginning of trigger implementation */
      SET NOCOUNT ON

      /* column variables declaration */
      DECLARE
        /* declare variables to store column values.
        if trigger has no references to :OLD or :NEW
        records then define the only uniqueidentifier type variable
        to store ROWID column value */
        @column_new_value$0 uniqueidentifier /* trigger has NO
references to :OLD or :NEW or has explicit reference to ROWID*/
```

```
      /* trigger has references to :OLD or :NEW*/

        @column_new_value$X <COLUMN_X_TYPE>,
        @column_new_value$Y <COLUMN_Y_TYPE>,
...
        @column_old_value$A <COLUMN_A_TYPE>,
        @column_old_value$B <COLUMN_B_TYPE>
...

      /* iterate for each for from inserted/updated table(s) */
      DECLARE ForEachInsertedRowTriggerCursor CURSOR LOCAL FORWARD_ONLY
READ_ONLY FOR
      /* trigger has NO references to :OLD or :NEW*/
        SELECT ROWID FROM inserted

      /* trigger has references to :OLD or :NEW* or has explicit
reference to ROWID/
        SELECT [ROWID], <COLUMN_X_NAME>,<COLUMN_Y_NAME> .. FROM
inserted

      OPEN ForEachInsertedRowTriggerCursor
      FETCH NEXT FROM ForEachInsertedRowTriggerCursor INTO
      /* trigger has NO references to :OLD or :NEW or has an explicit
reference to ROWID */
@column_new_value$0

      /* trigger has references to :NEW*/
@column_new_value$X
@column_new_value$Y
...


      WHILE @@fetch_status = 0
      BEGIN
-----------------------------------------------------------------------

        /* Oracle-trigger implementation: begin */
        BEGIN
  IF <WHILE_CLAUSE>
      BEGIN
                <TRIGGER_BODY>
            END
        END
        /* Oracle-trigger implementation: end */
-----------------------------------------------------------------------

        FETCH NEXT FROM ForEachInsertedRowTriggerCursor INTO
      /* trigger has NO references to :NEW or has an explicit reference
to ROWID */
@column_new_value$0

      /* trigger has references to :NEW*/
@column_new_value$X, @column_new_value$Y ...

      END

      CLOSE ForEachInsertedRowTriggerCursor
      DEALLOCATE ForEachInsertedRowTriggerCursor

      /*  end of trigger implementation */
```

Pattern for row-level AFTER DELETE triggers

```
CREATE TRIGGER [ schema. ]trigger ON <table>
    AFTER DELETE
    AS
      /*  beginning of trigger implementation */
      SET NOCOUNT ON

      /* column variables declaration */
      DECLARE
      /*
      Declare variables to store column values.
      If the trigger has no references to :OLD or :NEW records then
define the only uniqueidentifier type variable to store ROWID column
value. Else define variables to store old or new records. */
        @column_ old_value$0 uniqueidentifier /* trigger has NO
references to :OLD or :NEW or the trigger has explicit reference to
ROWID */


      /* trigger has references to :OLD or :NEW*/
        @column_new_value$X <COLUMN_X_TYPE>,
        @column_new_value$Y <COLUMN_Y_TYPE>,
...
        @column_old_value$A <COLUMN_A_TYPE>,
        @column_old_value$B <COLUMN_B_TYPE>,
...
      /* iterate for each for from inserted/updated table(s) */
      DECLARE ForEachDeletedRowTriggerCursor CURSOR LOCAL FORWARD_ONLY
READ_ONLY FOR
SELECT [ROWID,] [<COLUMN_A_NAME>, <COLUMN_B_NAME>..] FROM deleted

      OPEN ForEachDeletedRowTriggerCursor
      FETCH NEXT FROM ForEachDeletedRowTriggerCursor INTO
[@column_old_value$0,] [@column_old_value$A, @column_old_value$B ... ]


      WHILE @@fetch_status = 0
      BEGIN

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

        /* Oracle-trigger implementation: begin */
        BEGIN
            IF <WHERE_CLAUSE>
            BEGIN
                  <TRIGGER_BODY>
            END

        END
        /* Oracle-trigger implementation: end */
-----------------------------------------------------------------------

/*this is a trigger for delete event or a trigger for update event that
has no references both to :OLD and :NEW */
        FETCH NEXT FROM ForEachDeletedRowTriggerCursor INTO
[@column_old_value$0,] [@column_old_value$A, @column_old_value$B ... ]
      END
      CLOSE ForEachDeletedRowTriggerCursor
      DEALLOCATE ForEachDeletedRowTriggerCursor

      /*  end of trigger implementation */
```

Pattern for row-level AFTER UPDATE triggers

```
CREATE TRIGGER [ schema. ]trigger ON <table>
    AFTER UPDATE
    AS
      /*  beginning of trigger implementation */
      SET NOCOUNT ON

      /* column variables declaration */
      DECLARE
      /*
      Declare variables to store column values.
      If the trigger has no references to :OLD or :NEW records then
define the only uniqueidentifier type variable to store ROWID column
value. Else define variables to store old or new records. If the
trigger has references both to :OLD and :NEW then ALWAYS define
uniqueidentifier type variable to synchronize inserted row with deleted
row.
      */
        @column_new_value$0 uniqueidentifier /* trigger has NO
references to :OLD or :NEW or the trigger has references BOTH to :OLD
and :NEW  or the trigger has explicit reference to ROWID */



      /* trigger has references to :OLD or :NEW*/
        @column_new_value$X <COLUMN_X_TYPE>,
        @column_new_value$Y <COLUMN_Y_TYPE>,
...
        @column_old_value$A <COLUMN_A_TYPE>,
        @column_old_value$B <COLUMN_B_TYPE>,
...



/*the trigger has NO references both to :OLD and :NEW or has references
only to :OLD*/

      DECLARE ForEachDeletedRowTriggerCursor CURSOR LOCAL FORWARD_ONLY
READ_ONLY FOR
/*the trigger has NO references to :OLD and :NEW*/
SELECT ROWID FROM deleted
/*the trigger has references to :OLD*/
      SELECT <COLUMN_A_NAME>, <COLUMN_B_NAME>.. FROM deleted
/*the trigger has references to :OLD and explicit reference to ROWID */
SELECT ROWID, <COLUMN_A_NAME>, <COLUMN_B_NAME>.. FROM deleted



      OPEN ForEachDeletedRowTriggerCursor
      FETCH NEXT FROM ForEachDeletedRowTriggerCursor INTO
@column_old_value$0
```

```
/*the trigger has references to :NEW. If the trigger has references
both to :OLD and :NEW then we have to declare cursor for select ROWID
from inserted to synchronize inserted row with deleted row.
*/
      DECLARE ForEachInsertedRowTriggerCursor CURSOR LOCAL FORWARD_ONLY
READ_ONLY FOR
        SELECT [ROWID,] <COLUMN_X_NAME>, <COLUMN_Y_NAME> ... FROM
inserted
OPEN ForEachInsertedRowTriggerCursor
      FETCH NEXT FROM ForEachInsertedRowTriggerCursor INTO
[@column_new_value$0,] @column_new_value$X, @column_new_value$Y




      WHILE @@fetch_status = 0
      BEGIN

/*The trigger has references both to :OLD and :NEW. We have to
synchronize inserted row with deleted row */
      SELECT @column_old_value$A = <COLUMN_A_NAME>, @column_old_value$B
= <COLUMN_B_NAME>
          FROM deleted
          WHERE ROWID = @column_new_value$0

-------------------------------------------------------------------
        /* Oracle-trigger implementation: begin */
        BEGIN
        -- UPDATE OF CLAUSE
        -- (UPDATE OF COLUMN[, COLUMN] ... ])
        IF (UPDATE(<COLUMN>) OR UPDATE((<COLUMN>) ...)
          BEGIN
            IF <WHERE_CLAUSE>
            BEGIN
                  <TRIGGER_BODY>
            END
          END

        END
        /* Oracle-trigger implementation: end */
-------------------------------------------------------------------
/*the trigger has NO references both to :OLD and :NEW or has references
only to :OLD*/
        FETCH NEXT FROM ForEachDeletedRowTriggerCursor INTO
[@column_old_value$0,] [@column_old_value$A, @column_old_value$B ... ]
      END
      CLOSE ForEachDeletedRowTriggerCursor
      DEALLOCATE ForEachDeletedRowTriggerCursor

/* the trigger has references to :NEW */
FETCH NEXT FROM ForEachInsertedRowTriggerCursor INTO
[@column_new_value$0,] @column_new_value$X, @column_new_value$Y
      END
      CLOSE ForEachInsertedRowTriggerCursor
      DEALLOCATE ForEachInsertedRowTriggerCursor

/*  end of trigger implementation */
```

BEFORE Triggers

Because BEFORE triggers do not exist in SQL Server, SSMA emulates them by means of INSTEAD OF triggers. That change requires that the triggering statement be moved into the body of the trigger. Also, all triggers for a specific event should go into one target INSTEAD OF trigger.

Pattern for BEFORE DELETE triggers

```
CREATE
  TRIGGER [ schema. ] INSTEAD_OF_DELETE_ON_<table> ON <table>
    INSTEAD OF DELETE
    AS
      /*  beginning of trigger implementation */
      SET NOCOUNT ON

      /* column variables declaration */
      DECLARE
        @column_old_value$0 uniqueidentifier

      /* trigger has references to :OLD or :NEW*/
        @column_new_value$X <COLUMN_X_TYPE>,
        @column_new_value$Y <COLUMN_Y_TYPE>,
...
        @column_old_value$A <COLUMN_A_TYPE>,
        @column_old_value$B <COLUMN_B_TYPE>
...
-------------------------------------------------------------------
/* insert all table-level trigger implementations here */
<BEFORE_DELETE table-level trigger_1 body>
<BEFORE_DELETE table-level trigger_2 body>
...
```

```
--------------------------------------------------------------------

      /* iterate for each for from inserted/updated table(s) */
      DECLARE ForEachDeletedRowTriggerCursor CURSOR LOCAL FORWARD_ONLY
READ_ONLY FOR
        SELECT ROWID
/*if the trigger has references to :OLD*/
<COLUMN_A_NAME>,<COLUMN_B_NAME>, ...
 FROM deleted

      OPEN ForEachDeletedRowTriggerCursor
      FETCH NEXT FROM ForEachDeletedRowTriggerCursor INTO
@column_old_value$0
/*if the trigger has references to :OLD*/
, @column_old_value$A
,@column_old_value$B ...

      WHILE @@fetch_status = 0
      BEGIN

/* insert all row-level trigger implementations here*/

/* Oracle-trigger BEFORE_DELETE row-level trigger_1 implementation:
begin */
        BEGIN
          IF (<BEFORE_DELETE row-level trigger_1 WHERE_CLAUSE>)
            BEGIN
              <BEFORE_DELETE row-level trigger_1 body>
            END
        END
/* Oracle-trigger dbo BEFORE_DELETE row-level trigger_1 implementation:
end */

/* Oracle-trigger BEFORE_DELETE row-level trigger_2 implementation:
begin */
        BEGIN
          IF (<BEFORE_DELETE row-level trigger_2 WHERE_CLAUSE>)
            BEGIN
              <BEFORE_DELETE row-level trigger_2 body>
            END
        END
/* Oracle-trigger dbo BEFORE_DELETE row-level trigger_2 implementation:
end */
```

```
...

        /* DML-operation emulation */
        DELETE FROM <table>
          WHERE
            ROWID = @column_old_value$0

      FETCH NEXT FROM ForEachDeletedRowTriggerCursor INTO
@column_old_value$0
/*if the trigger has references to :OLD*/
, @column_old_value$A
,@column_old_value$B ...
      END

      CLOSE ForEachDeletedRowTriggerCursor
      DEALLOCATE ForEachDeletedRowTriggerCursor

      /*  end of trigger implementation */
```

Pattern for BEFORE UPDATE triggers

```
CREATE
  TRIGGER dbo.INSTEAD_OF_UPDATE_ON_<table> ON <table>
    INSTEAD OF UPDATE
    AS
      /*  beginning of trigger implementation */
      SET NOCOUNT ON

      /* column variables declaration */
      /* declare variables to store all table columns */
      DECLARE
        @column_new_value$0 uniqueidentifier,
        @column_new_value$1 <COLUMN_1_TYPE>,
        @column_new_value$2 <COLUMN_1_TYPE>,
...
/*declare variables to store values of :OLD*/
        @column_old_value$A <COLUMN_A_TYPE>,
        @column_old_value$B <COLUMN_B_TYPE>,
-------------------------------------------------------------------
/* insert all table-level trigger implementations here */
<BEFORE_UPDATE table-level trigger_1 body>
<BEFORE_UPDATE table-level trigger_2 body>
```

```
...

------------------------------------------------------------------
      /* iterate for each for from inserted/updated table(s) */
      DECLARE ForEachInsertedRowTriggerCursor CURSOR LOCAL FORWARD_ONLY
READ_ONLY FOR
        SELECT ROWID, <COLUMN_NAME_1>, <COLUMN_NAME_2> ... FROM
inserted

      OPEN ForEachInsertedRowTriggerCursor
      FETCH NEXT FROM ForEachInsertedRowTriggerCursor INTO
@column_new_value$0, @column_new_value$1, @column_new_value$2, ...

      WHILE @@fetch_status = 0
      BEGIN

  /*if the trigger has references to :OLD*/
        /* synchronize inserted row with deleted row */
        SELECT @column_old_value$A = <COLUMN_A_NAME>,
@column_old_value$B = <COLUMN_B_NAME>, ...
          FROM deleted
          WHERE ROWID = @column_new_value$0


/* insert all row-level trigger implementations here */

/* Oracle-trigger BEFORE_UPDATE row-level trigger_1 implementation:
begin */
        BEGIN
        -- (UPDATE OF COLUMN[, COLUMN] ... ])
        IF (UPDATE(<COLUMN>) OR UPDATE((<COLUMN>) ...)
          BEGIN
            IF <<BEFORE_UPDATE row-level trigger_1 WHERE_CLAUSE>>
            BEGIN
              <BEFORE_UPDATE row-level trigger_1 body>
            END
          END
        END
/* Oracle-trigger dbo BEFORE_UPDATE row-level trigger_1 implementation:
end */

/* Oracle-trigger BEFORE_UPDATE row-level trigger_2 implementation:
begin */
        BEGIN
        -- (UPDATE OF COLUMN[, COLUMN] ... ])
        IF (UPDATE(<COLUMN>) OR UPDATE((<COLUMN>) ...)
          BEGIN
            IF <<BEFORE_UPDATE row-level trigger_2 WHERE_CLAUSE>>
            BEGIN
              <BEFORE_UPDATE row-level trigger_2 body>
            END
          END
        END
/* Oracle-trigger dbo BEFORE_UPDATE row-level trigger_2 implementation:
end */

...

        /* DML-operation emulation */
        UPDATE <table>
          SET
            <COLUMN_NAME_1> = @column_new_value$1,
            <COLUMN_NAME_1> = @column_new_value$1,
            ...
          WHERE
            ROWID = @column_new_value$0

      FETCH NEXT FROM ForEachInsertedRowTriggerCursor INTO
@column_new_value$0, @column_new_value$1, @column_new_value$2, ...

      END

      CLOSE ForEachInsertedRowTriggerCursor
      DEALLOCATE ForEachInsertedRowTriggerCursor

      /*  end of trigger implementation */
```

Pattern for BEFORE INSERT triggers

```
CREATE TRIGGER dbo.INSTEAD_OF_INSERT_ON_<table> ON <table>
    INSTEAD OF INSERT
    AS
      /*  beginning of trigger implementation */
      SET NOCOUNT ON

      /* column variables declaration */
      /* declare variables to store all table columns */
      DECLARE
        @column_new_value$1 <COLUMN_1_TYPE>,
        @column_new_value$2 <COLUMN_1_TYPE>,
  ...

/*declare variables to store values of :OLD*/
        @column_old_value$A <COLUMN_A_TYPE>,
        @column_old_value$B <COLUMN_B_TYPE>,
  ...
------------------------------------------------------------------------

/* insert all table-level trigger implementations here */
<BEFORE_INSERT table-level trigger_1 body>
<BEFORE_INSERT table-level trigger_2 body>
...
------------------------------------------------------------------------



      /* iterate for each for from inserted/updated table(s) */
      DECLARE ForEachInsertedRowTriggerCursor CURSOR LOCAL FORWARD_ONLY
READ_ONLY FOR
        SELECT <COLUMN_1_NAME>,<COLUMN_2_NAME> ... FROM inserted

      OPEN ForEachInsertedRowTriggerCursor
      FETCH NEXT FROM ForEachInsertedRowTriggerCursor INTO
@column_new_value$1, @column_new_value$2, ...

      WHILE @@fetch_status = 0
      BEGIN

/* insert all row-level trigger implementations here */
/* Oracle-trigger BEFORE_INSERT row-level trigger_1 implementation:
begin */
        BEGIN
          IF (<BEFORE_UPDATE row-level trigger_1 WHERE_CLAUSE>)
            BEGIN
              <BEFORE_UPDATE row-level trigger_1 body>
            END
        END
/* Oracle-trigger dbo BEFORE_UPDATE row-level trigger_1 implementation:
end */

/* Oracle-trigger BEFORE_INSERT row-level trigger_2 implementation:
begin */
        BEGIN
          IF (<BEFORE_UPDATE row-level trigger_2 WHERE_CLAUSE>)
```

```
            BEGIN
              <BEFORE_UPDATE row-level trigger_2 body>
            END
        END
/* Oracle-trigger dbo BEFORE_UPDATE row-level trigger_2 implementation:
end */

...

        /* DML-operation emulation */
        INSERT INTO <table> (<COLUMN_1_NAME>,<COLUMN_2_NAME> ...)
          VALUES (@column_new_value$1, @column_new_value$2, ...)

        FETCH NEXT FROM ForEachInsertedRowTriggerCursor INTO
@column_new_value$1, @column_new_value$2, ...
      END

      CLOSE ForEachInsertedRowTriggerCursor
      DEALLOCATE ForEachInsertedRowTriggerCursor

      /*  end of trigger implementation */
```

INSTEAD OF Triggers

Oracle INSTEAD OF triggers remain INSTEAD OF triggers in SQL Server. Combine multiple INSTEAD OF triggers that are defined on the same event into one trigger. INSTEAD OF trigger statements are implicitly activated for each row.

Pattern for INSTEAD OF UPDATE triggers and INSTEAD OF DELETE triggers

```
CREATE
  TRIGGER [schema. ]INSTEAD_OF_UPDATE_ON_VIEW_<table> ON <table>
    INSTEAD OF {UPDATE | DELETE}
    AS
      /*  beginning of trigger implementation */
      SET NOCOUNT ON

      /* column variables declaration */
      DECLARE
/*if the trigger has no references to :OLD that define one variable to
store first column. Else define only columns that have references to
:OLD*/
```

```
        @column_old_value$1 <COLUMN_1_TYPE>

        @column_old_value$X <COLUMN_X_TYPE>,
        @column_old_value$Y <COLUMN_Y_TYPE>,
...
        /*define columns to store references to :NEW*/
        @column_new_value$A <COLUMN_A_TYPE>,
        @column_new_value$B <COLUMN_B_TYPE>,
...

      /* iterate for each for from inserted/updated table(s) */

      /* For trigger for UPDATE event that has references to :NEW
define and open cursor from inserted as well*/
      DECLARE ForEachInsertedRowTriggerCursor CURSOR LOCAL FORWARD_ONLY
READ_ONLY FOR
        SELECT <COLUMN_A_NAME>, <COLUMN_B_NAME> ... FROM inserted

      OPEN ForEachInsertedRowTriggerCursor
      FETCH NEXT FROM ForEachInsertedRowTriggerCursor INTO
@column_new_value$A, @column_new_value$B ...

      DECLARE ForEachDeletedRowTriggerCursor CURSOR LOCAL FORWARD_ONLY
READ_ONLY FOR
        SELECT <COLUMN_X_NAME>, <COLUMN_Y_NAME> ... FROM deleted

      OPEN ForEachDeletedRowTriggerCursor
      FETCH NEXT FROM ForEachDeletedRowTriggerCursor INTO
      /* trigger has no references to :OLD*/
      @column_old_value$1
      /* trigger has references to :OLD*/
      @column_old_value$X, @column_old_value$Y ...

      WHILE @@fetch_status = 0
      BEGIN

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

/* Oracle-trigger INSTEAD OF UPDATE/DELETE trigger_1 implementation:
begin */
        BEGIN
          < INSTEAD OF UPDATE/DELETE trigger_1 BODY>
        END
```

```
/* Oracle-trigger INSTEAD OF UPDATE/DELETE trigger_1 implementation:
end */

/* Oracle-trigger INSTEAD OF UPDATE/DELETE trigger_2 implementation:
begin */
        BEGIN
          < INSTEAD OF UPDATE/DELETE trigger_1 BODY>
        END
/* Oracle-trigger INSTEAD OF UPDATE/DELETE trigger_2 implementation:
end */

...
------------------------------------------------------------------------

/*Only for trigger for UPDATE event that has references to :NEW*/
      FETCH NEXT FROM ForEachInsertedRowTriggerCursor INTO
@column_new_value$A, @column_new_value$B ...

      OPEN ForEachDeletedRowTriggerCursor
      FETCH NEXT FROM ForEachDeletedRowTriggerCursor INTO
      /* trigger has no references to :OLD*/
      @column_old_value$1
      /* trigger has references to :OLD*/
      @column_old_value$X, @column_old_value$Y ...

      END

/*Only for trigger for UPDATE event that has references to :NEW*/
      CLOSE ForEachInsertedRowTriggerCursor
      DEALLOCATE ForEachInsertedRowTriggerCursor

      CLOSE ForEachDeletedRowTriggerCursor
      DEALLOCATE ForEachDeletedRowTriggerCursor

      /*  end of trigger implementation */
```

Pattern for INSTEAD OF INSERT triggers

INSTEAD OF triggers are converted in the same way as DELETE and UPDATE triggers, except the iteration for each row is made with the inserted table.

```
CREATE TRIGGER [schema. ]INSTEAD_OF_INSERT_ON_VIEW_<table> ON <table>
    INSTEAD OF INSERT
    AS
      /*  beginning of trigger implementation */
      SET NOCOUNT ON

      /* column variables declaration */
      DECLARE
      /*if the trigger has no references to :NEW that define one
variable to store first column. Else define only columns that have
references to :NEW*/
        @column_new_value$1 <COLUMN_1_TYPE>

        @column_new_value$X <COLUMN_X_TYPE>,
        @column_new_value$Y <COLUMN_Y_TYPE>,
...

      /*define columns to store references to :OLD */
        @column_old_value$A <COLUMN_A_TYPE>,
        @column_old_value$B <COLUMN_B_TYPE>,
...

      /* iterate for each for from inserted/updated table(s) */

      DECLARE ForEachInsertedRowTriggerCursor CURSOR LOCAL FORWARD_ONLY
READ_ONLY FOR
        SELECT <COLUMN_X_NAME>, <COLUMN_Y_NAME> ... FROM inserted

      OPEN ForEachInsertedRowTriggerCursor
      FETCH NEXT FROM ForEachDeletedRowTriggerCursor INTO
      /* trigger has no references to :NEW*/
      @column_new_value$1
      /* trigger has references to :NEW*/
      @column_new_value$X, @column_new_value$Y ...

      WHILE @@fetch_status = 0
      BEGIN

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

        /* Oracle-trigger INSTEAD OF INSERT trigger_1 implementation:
begin */
        BEGIN
          < INSTEAD OF INSERT trigger_1 BODY>
        END
        /* Oracle-trigger INSTEAD OF INSERT trigger_1 implementation:
end */
```

```
        /* Oracle-trigger INSTEAD OF INSERT trigger_2 implementation: 
begin */
        BEGIN
          < INSTEAD OF INSERT trigger_1 BODY>
        END
        /* Oracle-trigger INSTEAD OF INSERT trigger_2 implementation: 
end */
------------------------------------------------------------------------

      OPEN ForEachInsertedRowTriggerCursor
      FETCH NEXT FROM ForEachDeletedRowTriggerCursor INTO
      /* trigger has no references to :NEW*/
      @column_new_value$1
      /* trigger has references to :NEW*/
      @column_new_value$X, @column_new_value$Y ...

      END

      CLOSE ForEachInsertedRowTriggerCursor
      DEALLOCATE ForEachInsertedRowTriggerCursor


      /*  end of trigger implementation */
```

Autonomous Transactions in Triggers

Convert triggers with PRAGMA AUTONOMOUS_TRANSACTION as described earlier, except execute the trigger body in a separate connection. SSMA uses the **xp_ora2ms_exec2_ex** extended procedure, which launches the trigger body's procedure implementation. That procedure is created when you install the SSMA Extension Pack.

Note that extended stored procedures functionality is not supported in Azure SQL DB and **xp_ora2ms_exec2_ex** cannot be used to emulate autonomous transactions in triggers in this version of SQL Server.

Pattern for the trigger body

```
declare @spid int, @login_time datetime
select @spid = ssma_oracle.get_active_spid(),
@login_time = ssma_oracle.get_active_login_time()

EXEC master.dbo.xp_ora2ms_exec2_ex @spid, @ login_time,
<database_name>, <schema_name>,
<trigger_implementation_as_procedure_name>,
0, [parameter1, parameter2, ... ,]
```

The trigger body's procedure implementation follows a pattern that depends on the trigger type. For all types of table-level triggers, this procedure has no parameters.

Because the first PL/SQL statement in an autonomous routine begins a new transaction, the procedure body should begin with the set implicit_transactions on statement.

Pattern for implementation of table-level triggers

```
create procedure <trigger_name>$imlp
as begin
set implicit_transactions on
<TRIGGER_BODY>
end
```

For row-level triggers, SSMA passes NEW and OLD rows to the procedure. In BEFORE UPDATE and BEFORE INSERT row-level triggers, you can write to the :NEW value. So in autonomous transactions you must pass a :NEW value back to a trigger.

In that way, the pattern for row-level trigger-body procedure implementation looks like the following.

Pattern for implementing AFTER, INSTEAD OF, and BEFORE DELETE row-level triggers

```
create procedure <trigger_name>$impl
@rowid,@column_new_value$1,@column_new_value$2, ... ,
@column_old_value$1,@column_old_value$2..
as begin
set implicit_transactions on
<TRIGGER_BODY>
end
```

Pattern for implementing BEFORE UPDATE and BEFORE INSERT row-level triggers

```
create procedure <before_trigger_name>$imlp
@rowid,@column_new_value$1 output ,@column_new_value$2 output, ... ,
@column_old_value$1,@column_old_value$2..
as begin
set implicit_transactions on
```

```
<TRIGGER_BODY>
end
```

The logic of these patterns for all types of row-level triggers remains the same, except SSMA creates references to all columns of :NEW and :OLD values.

- In row-level triggers for the INSERT event, you pass references to the :NEW value and null values instead of the :OLD value.
- In row-level triggers for the DELETE event, you pass references to the :OLD value and null values instead of the :NEW value.
- In row-level triggers for the UPDATE event, you pass references to both the :OLD value and the :NEW value.

Notes on Autonomous Transaction Conversion in Triggers

In Oracle, none of the changes made in the main transaction are visible to an autonomous transaction. To protect the autonomous transaction from reading uncommitted data, we recommend using a row-versioning isolation level. To provide the complete emulation of autonomous transactions in SQL Server and to enable a row-versioning isolation level, set the ALLOW_SNAPSHOT_ISOLATION option to ON for each database referenced in the autonomous block. In addition, start the autonomous block with a SNAPSHOT isolation level. Alternatively, you can start an autonomous block with the READ COMMITTED isolation level when the READ_COMMITTED_SNAPSHOT database option is set to ON.

The Execution Order of Triggers

The Oracle trigger syntax now includes the FOLLOWS clause to guarantee execution order for triggers defined with the same timing point.

```
create or replace trigger Trg_2

before insert on My_Table

for each row

FOLLOWS Trg_1

begin

...

end;
```

In SQL Server trigger it is necessary to join all triggers in the only from Oracle's triggers with the same timing point in the order of its execution.

Compound Triggers

A compound trigger allows code for one or more timing points for a specific object to be combined into a single trigger.

```
CREATE OR REPLACE TRIGGER <trigger-name>

  FOR <trigger-action> /*INSERT, UPDATE, DELETE*/

  ON <table-name>

  COMPOUND TRIGGER

  -- Global declaration.

  g_global_variable VARCHAR2(10);


  BEFORE STATEMENT IS

  BEGIN

    NULL; -- Do something here.

  END BEFORE STATEMENT;


  BEFORE EACH ROW IS

  BEGIN

    NULL; -- Do something here.

  END BEFORE EACH ROW;


  AFTER EACH ROW IS

  BEGIN

    NULL; -- Do something here.

  END AFTER EACH ROW;


  AFTER STATEMENT IS

  BEGIN

    NULL; -- Do something here.

  END AFTER STATEMENT;
```

```
    END <trigger-name>;
```

The individual timing points can share a single global declaration section, whose state is maintained for the lifetime of the statement.

In this case it is necessary split this trigger to the several ones with the same timing point by rules as described above.

As for work with global section it is necessary to work with it as described in section Emulating Oracle Packages.

Emulating Oracle Packages

Oracle supports encapsulating variables, types, stored procedures, and functions into a package. This section describes SSMA for Oracle V6.0 conversion algorithms, which allow packages to be emulated in Microsoft SQL Server 2014.

When you convert Oracle packages, you need to convert:

- Packaged procedures and functions (both public and private).
- Packaged variables.
- Packaged cursors.
- Package initialization routines.

Let's examine each of these in turn.

Converting Procedures and Functions

As one of its functions, an Oracle package allows you to group procedures and functions. In SQL Server 2014, you can group procedures and functions by their names. Suppose that you have the following Oracle package:

```
CREATE OR REPLACE PACKAGE MY_PACKAGE
IS
 space varchar(1) := ' ';
 unitname varchar(128) := 'My Simple Package';
 curd date := sysdate;
 procedure MySimpleProcedure;
 procedure MySimpleProcedure(s in varchar);
 function  MyFunction return varchar2;
END;


CREATE OR REPLACE PACKAGE BODY MY_PACKAGE
IS

procedure MySimpleProcedure
is begin
 dbms_output.put_line(MyFunction);
end;

procedure MySimpleProcedure(s in varchar)
is begin
   dbms_output.put_line(s);
end;

function MyFunction return varchar2
is begin
 return 'Hello, World!';
end;
```

```
END;
```

In SQL Server 2014, you can group procedures and functions by giving them names such as Scott.MY_PACKAGE$MySimpleProcedure and Scott.MY_PACKAGE$MyFunction. The naming pattern is <schema name>.<package name>$<procedure or function name>. For more information about converting functions, see Migrating Oracle User-Defined Functions.

Convert the invoker rights clause AUTHID to an EXECUTE AS clause, and apply it to all packaged procedures and functions. Also convert the CURRENT_USER argument to the CALLER argument, and convert the DEFINER argument to the OWNER argument.

Converting Overloaded Procedures

You can create overloaded procedures in Oracle (procedures with same name but with different parameters and bodies). SQL Server 2014, in contrast, does not support procedure overloading. Therefore, you should distinguish each procedure's instance.

The naming pattern could resemble <schema name>.<package name>$<procedure name>$ovl<# of procedure instance>. For example, Scott$MY_PACKAGE$MySimpleProcedure$OVL1 and Scott$MY_PACKAGE$MySimpleProcedure$OVL2.

Here's some sample converted Transact-SQL code:

```
create function Scott.MY_PACKAGE$MyFunction()
returns varchar(max)
as begin
 return 'Hello, world!'
end
go

create procedure Scott.MY_PACKAGE$MySimpleProcedure$OVL1
as begin
 print dbo.MY_PACKAGE$MyFunction()
end
go

create procedure Scott.MY_PACKAGE$MySimpleProcedure$OVL2(@s
varchar(max))
as begin
 print @s
end
go
```

Converting Packaged Variables

To store packaged variables, establish session-depended storage. SSMA for Oracle V6.0 provides an excellent solution. For the task, SSMA uses special tables that reside in an ssma_oracle schema. For access to these variables SSMA uses a set of transaction-independent GET and SET procedures and functions. Also, these procedures ensure session independence — you should distinguish between variables from different sessions. SSMA distinguishes package variables by SPID (session identifier) and the session's login time.

Note: If a packaged variable is declared with an initial value, you must move the initialization to the package's initialization section.

Converting Simple Variables

Simple variables (**numeric**, **varchar**, **datetime**) are stored separately in the appropriate column in table ssma_oracle.db_storage in the converted database.

In some cases you can replace constant packaged variables with user-defined functions that return the appropriate value. For example, you could convert the packaged variable unitname (from the earlier example) as:

```
create function scott$my_package$unitname()
returns varchar(128)
as begin
 return 'My Simple Package'
end
```

And, you should convert all references to this variable:

```
dbms_output.put_line(my_package.unitname);
```

To:

```
print scott.my_package$unitname()
```

Converting Packaged Cursors

SSMA for Oracle V6.0 converts packaged cursors as GLOBAL cursors with names such as <schema>$<package name>$<cursor name>.

The declaration of cursor is invoked in the package initialization section. Each database method that uses packaged cursors contains the call of the package initialization procedure. The call is invoked before the first usage of the packaged cursor.

(For basic information about cursor conversion, see Migrating Oracle Cursors. You will also find a description of converting FOUND, ISOPEN, and NOTFOUND cursor attributes.)

The ROWCOUNT attribute is converted as a package variable. The variable is initialized to null in the init section; after OPEN, its value is set to zero and is incremented after each FETCH.

Converting Initialization Section

The initialization section itself is converted as the usual packaged procedure. Within each converted procedure or function, a call to the initialization procedure is included.

Note   Initialization should be performed only one time per session, so the initialization procedure must check each package's initialization status.

Calling Initialization from the Within Procedure

Calling the initialization procedure from within a GET procedure has one main problem: the initialization of packaged variables requires that a number of rows to be inserted into a storage table and that insertion should be transaction-independent. This is why SSMA uses an extended stored procedure to perform this task.

Calling Initialization from the Within Function

Before the value is obtained from a package variable, it should be initialized. The initialization routine should be called to do this. You cannot call stored procedures directly from within a function, so SSMA calls the initialization procedure by executing an extended stored procedure.

SSMA's Package Variables Implementation Details

SSMA stores package variables in the converted database in an **ssma_oracle.db_storage** table. The table is filtered by SPID and login time. This filtering enables you to distinguish between variables of different sessions.

SSMA creates the initialization procedure with a name such as Scott.MY_PACKAGE$SSMA_Initialize_Package. The name pattern is <schema>.<packagename>$SSMA_Initialize_Package.

At the beginning of each procedure SSMA places a call to the **ssma_oracle.db_check_init_package** procedure. That procedure checks if the package is not initialized yet, and, if not, it initializes the package.

As a mark of package initialization, SSMA uses package variable with a name such as $<dbname>.<schema>.<package>$init$. If that variable is present in the **db_storage** table, the package is already initialized, and therefore no initialization call is required. Because it is not possible to call a procedure from a user-defined function, the check for initialization is performed by the function **db_fn_check_init_package**. In its turn **db_fn_check_init_package** makes a call to **xp_ora2ms_exec2** to execute the package initialization routine.

Each initialization procedure cleans the storage table and sets default values for each packaged variable:

```
CREATE PROCEDURE dbo.MY_PACKAGE$SSMA_Initialize_Package
AS
   EXECUTE ssma_oracle.db_clean_storage
   EXECUTE ssma_oracle.set_pv_varchar
      'SYS',
      'DBO',
      'MY_PACKAGE',
      'SPACE',
      ' '
   EXECUTE ssma_oracle.set_pv_varchar
      'SYS',
      'DBO',
      'MY_PACKAGE',
      'UNITNAME',
      'My Simple Package'
```

Package Conversion Code Example

For further reference, consider the following package conversion example:

```
CREATE FUNCTION dbo.MY_PACKAGE$MyFunction () RETURNS varchar(max)
AS
   BEGIN
      EXECUTE ssma_oracle.db_fn_check_init_package 'SCOTT', 'DBO',
'MY_PACKAGE'
      RETURN 'Hello, World!'
   END
GO

CREATE PROCEDURE dbo.MY_PACKAGE$MySimpleProcedure$1
AS
   BEGIN
      EXECUTE ssma_oracle.db_check_init_package 'SCOTT', 'DBO',
'MY_PACKAGE'
      PRINT dbo.MY_PACKAGE$MyFunction()
   END
GO


CREATE PROCEDURE dbo.MY_PACKAGE$MySimpleProcedure$2
   @s varchar(max)
AS
   BEGIN
      EXECUTE ssma_oracle.db_check_init_package 'SCOTT', 'DBO',
'MY_PACKAGE'
      PRINT @s
   END
GO

CREATE PROCEDURE dbo.MY_PACKAGE$SSMA_Initialize_Package
```

```
AS
   EXECUTE ssma_oracle.db_clean_storage
   EXECUTE ssma_oracle.set_pv_varchar
      'SCOTT',
      'DBO',
      'MY_PACKAGE',
      'SPACE',
      ' '
   EXECUTE ssma_oracle.set_pv_varchar
      'SCOTT',
      'DBO',
      'MY_PACKAGE',
      'UNITNAME',
      'My Simple Package'
   DECLARE
      @temp datetime
   SET @temp = getdate()
   EXECUTE ssma_oracle.set_pv_datetime
      'SCOTT',
      'DBO',
      'MY_PACKAGE',
      'CURD',
      @temp
GO
```

Converting Packages to Azure SQL DB

Emulation of Oracle packages in SQL Server is based on autonomous transaction and ability to call procedure from function. Both functionalities are gained from extended stored procedure xp_ora2ms_exec. Azure SQL DB doesn't support extended stored procedures functionality.

That's why for package emulation on Azure SQL DB the following issues are appeared:

1. Package initialization block;
2. Package state can't be changed inside functions;
3. Transactional behavior of package variables.

Package Initialization Block

In Oracle we can create a special block that will initialize the package before the first usage by user session. To emulate such behavior on SQL Server, we convert that block as a procedure and place a special code in each package routine that checks initialization and executes this procedure if the initialization hasn't been done yet.

But on Azure SQL DB, the initialization procedure can't be run from the context of a function. Thus, the package can't be initialized when we call package function or package variables getters (ssma_oracle.get_pv_* functions).

As a workaround, the call to ssma_oracle.db_check_init_package procedure is placed before the calling package function or package variables getters inside the code that use them but not inside functions of course.

SSMA replaces the initialization function call to ssma_oracle.db_check_init_package procedure call where possible, but in other cases this should be done manually depending on the code that uses the converted package and its routines.

Oracle

```
CREATE OR REPLACE package local_1
as
   function f1 return int;
end;


CREATE OR REPLACE package body local_1
as
   val int;

   function f2 return int   as
   begin
      val:=val+1;
      return val;
   end;

   function f1 return int   as
   begin
      return f2();
   end;

begin
   val := 4;
end;


-- Oracle Call to package
declare
   x int;
begin
      x := local_1.f1() ;
end;
```

SSMA conversion to Azure SQL DB

```
CREATE PROCEDURE dbo.LOCAL_1$SSMA_Initialize_Package
AS
   EXECUTE
      ssma_oracle.db_clean_storage
   EXECUTE ssma_oracle.set_pv_int 'DBO', 'LOCAL_1', 'VAL', 4
GO
```

```
CREATE FUNCTION dbo.LOCAL_1$f1
(
)
RETURNS int
AS
   BEGIN

      /*
      *   SSMA error messages:
      *   O2SS0516: Init block can’t be used inside function.
      EXECUTE ssma_oracle.db_fn_check_init_package 'DBO', 'LOCAL_1'
      */

      RETURN dbo.LOCAL_1$f2()

   END
GO


/*
*   SSMA error messages:
*   O2SS0518: Wrapper functions are not supported by Azure SQL DB
platform. Use $impl procedures instead.

CREATE FUNCTION dbo.LOCAL_1$f2
(
)
RETURNS int
AS
   BEGIN
      DECLARE
         @active_spid INT,
         @login_time DATETIME

      SET @active_spid = ssma_oracle.GET_ACTIVE_SPID()

      SET @login_time = ssma_oracle.GET_ACTIVE_LOGIN_TIME()

      DECLARE @return_value_argument int

      /*
      *   SSMA warning messages:
      *   O2SS0452: "xp_ora2ms_exec2_ex" when called from within UDF
cannot bind to outer transaction. It can lead to dead locks and losing
transaction atomicity. Consider calling $impl procedure directly.
      */

      EXECUTE master.dbo.xp_ora2ms_exec2_ex
         @active_spid,
         @login_time,
         N'ATEST',
         N'ATEST',
         N'LOCAL_1$F2$IMPL',
         N'true',
         @return_value_argument  OUTPUT

      RETURN @return_value_argument

   END
*/
```

```
CREATE PROCEDURE dbo.LOCAL_1$f2$IMPL
   @return_value_argument int  OUTPUT
AS
   BEGIN
      DECLARE
         @temp int
      SET @temp = ssma_oracle.get_pv_int('DBO', 'LOCAL_1', 'VAL') + 1
      EXECUTE ssma_oracle.set_pv_int 'DBO', 'LOCAL_1', 'VAL', @temp
      SET @return_value_argument = ssma_oracle.get_pv_int('DBO',
'LOCAL_1', 'VAL')
      RETURN
   END
GO

-- Azure SQL DB call to the package
BEGIN
   DECLARE
      @x int
   SET @x = dbo.LOCAL_1$f1()
END
GO
```

The following is the workaround that can be applied manually for Azure SQL DB as the package initialization block:

```
CREATE PROCEDURE dbo.LOCAL_1$SSMA_Initialize_Package
AS
   EXECUTE
      ssma_oracle.db_clean_storage
   EXECUTE ssma_oracle.set_pv_int 'DBO', 'LOCAL_1', 'VAL', 4
GO

-- Azure SQL DB call to the package
BEGIN
   DECLARE
      @x int
   EXECUTE ssma_oracle.db_check_init_package 'DBO', 'LOCAL_1'

   SET @x = dbo.LOCAL_1$f1()
END
GO
```

Package State Cannot Be Changed Inside Functions

In Oracle, package variables values can be changed inside user defined functions.

For SQL Server, such functions are emulated using xp_ora2ms_exec2_ex extended stored procedure and implementation procedures. Azure SQL DB does not support extended stored procedures. That is why the above emulation is not applicable for this database.

SSMA marks the converted function that contains variables changed inside it with the following error message: “Wrapper functions are not supported by Azure SQL DB platform. Use $impl procedures instead.”

Regarding to the previous Oracle example and its conversion to Azure SQL DB, the packaged function f2 that changes packaged variable (and function f1 that calls f2) can be rewritten manually into its implementation procedure and the calling code can be changed in the following way:

Azure SQL DB:

```
CREATE PROCEDURE dbo.LOCAL_1$SSMA_Initialize_Package
AS
BEGIN
   EXECUTE
      ssma_oracle.db_clean_storage
   EXECUTE ssma_oracle.set_pv_int 'DBO', 'LOCAL_1', 'VAL', 4
END
GO


CREATE PROCEDURE dbo.LOCAL_1$f1
   @return_value int OUTPUT
AS
BEGIN

   EXECUTE dbo.LOCAL_1$f2 @return_value OUTPUT
   RETURN
END
GO


CREATE PROCEDURE dbo.LOCAL_1$f2
   @return_value int  OUTPUT
AS
BEGIN
   EXECUTE ssma_oracle.db_check_init_package 'DBO', 'LOCAL_1'

   DECLARE @temp int
   SET @temp = ssma_oracle.get_pv_int('DBO', 'LOCAL_1', 'VAL') + 1

   EXECUTE ssma_oracle.set_pv_int 'DBO', 'LOCAL_1', 'VAL', @temp

   SET @return_value = ssma_oracle.get_pv_int('DBO', 'LOCAL_1', 'VAL')
   RETURN
END

-- Azure SQL DB call to the package
BEGIN
   DECLARE
      @x int
   EXECUTE ssma_oracle.db_check_init_package 'DBO', 'LOCAL_1'

   EXECUTE dbo.LOCAL_1$f1 @x OUTPUT
END
GO
```

Transactional behavior of package variables

In Oracle, package variables are not transactional and are stored independently for each user session. But in Azure SQL DB, any data can't be stored in transactionally independent way.

The best workaround is to store package variable value in some table marking it with session identifier. But this value will be lost after rollback or even switched to some previous value if we use rollback to save point.

SSMA adds a warning about that inside initialization block for packages that has variables:

Oracle

```
CREATE OR REPLACE package       local_2
as
       v_1 int;
       v_2 int;
end;
```

Azure SQL DB

```
/*
*   SSMA warning messages:
*   O2SS0519: Package variables value will be changed if rollback
occurred.
*/

CREATE PROCEDURE dbo.LOCAL_2$SSMA_Initialize_Package
AS
   EXECUTE
      ssma_oracle.db_clean_storage
GO
```

Conversion of Oracle Materialized Views

Previous versions of SSMA converted Oracle materialized views to tables but since this version, materialized views are converted to indexed views.

SSMA now displays materialized views in Oracle Metadata Explorer and shows their quantity.

While converting a materialized view, SSMA creates necessary unique clustered index on the view in SQL Server and adds WITH SCHEMABINDING option to the CREATE VIEW statement. The Ixdexes and Triggers nodes are added as subnodes to Views in SQL Server Metadata Explorer.

Indexed view are created using the following statements:

```
CREATE VIEW <materialized_view_name>
WITH SCHEMABINDING
AS
    SELECT ... ;
GO

CREATE UNIQUE CLUSTERED <index_name>
    ON <materialized_view_name> (<field1>, <field2> ...);
GO
```

The view has to have unique clustered index. Index fields are set of primary keys (or other unique fields/field sets) of participating tables at least. The view must reference only base tables that are in the same database as the view. The view cannot reference other views.

SSMA parses SELECT statement of the materialized view DDL definition and determines a degree of compatibility with SQL Server requirements for indexed views.

The following table shows SQL elements that are not supported in the indexed view definition, but SSMA can handle their conversion:

SELECT statement contains	SSMA conversion	Example

COUNT(*)	Replaces to SUM(1)	`select COUNT(*) from customers`
		
		`SSMA: select SUM(1) from customers`

DISTINCT	Replaces to GROUP BY	`select DISTINCT name from people`
		
		`SSMA: select name from`

		people GROUP BY name
AVG	Replases to SUM() / SUM(1)	select AVG(age) from people SSMA: select SUM(age)/SUM(1) from people
SUM(<nullable field>)	Makes <field> definition as NOT NULL (e.g. with DEFAULT 0)	create table people (id int not null, name nvarchar(100)) SSMA: create table people (id int not null, name nvarchar(100) NOT NULL)
GROUP BY	Add column COUNT_BIG(*) to SELECT list	select a, sum(b) sb from abc group by a SSMA: select a, sum(b) sb, count_big(*) [cnt$big] from abc group by a
ORDER BY	Removes ORDER BY clause	select name from people ORDER BY name SSMA: select name from people

The following SQL elements are not supported in the indexed view definition and SSMA marks their conversion with error messages:

- User-defined functions;
- Non-deterministic fields, functions (such as SYSDATE), expressions in SELECT, WHERE or GROUP BY clauses;
- Usage of FLOAT column in SELECT, WHERE or GROUP BY clauses. Indexed view can contain FLOAT column in SELECT list only if this column is not included in the clustered index key;
- Custom data types (including nested tables);
- COUNT(DISTINCT <field>);
- FETCH statement;
- OUTER joins (LEFT, RIGHT, or FULL);
- Subquery or other view;
- OVER clause and ranking functions RANK, LEAD, LAG;
- MIN, MAX functions;
- UNION, MINUS, INTERSECT operators;
- HAVING clause.

Below is the example of SSMA conversion of materialized views to SQL Server:

Oracle

```
CREATE MATERIALIZED VIEW PRODUCTS_MV (PROD_ID, PRODUCT_NAME)
  AS SELECT p.prod_id, p.prod_name
     FROM products p;
```

SQL Server

```
CREATE VIEW dbo.PRODUCTS_MV
WITH SCHEMABINDING
AS
   SELECT p.PROD_ID, p.PROD_NAME
   FROM dbo.PRODUCTS  AS p
GO

IF EXISTS (
       SELECT * FROM sys.objects  so JOIN sys.indexes si
       ON so.object_id = si.object_id
       JOIN sys.schemas sc
       ON so.schema_id = sc.schema_id
       WHERE so.name = N'PRODUCTS_MV'  AND sc.name = N'dbo'  AND
si.name = N'UIX_PROD_dbo_PRODUCTS_MV_p_PROD_ID' AND so.type in (N'U'))
   DROP INDEX [dbo].[PRODUCTS_MV].[UIX_PROD_dbo_PRODUCTS_MV_p_PROD_ID]
GO
CREATE UNIQUE CLUSTERED INDEX [UIX_ATEST_dbo_PRODUCTS_MV_p_PROD_ID] ON
[dbo].[PRODUCTS_MV]
(
   [PROD_ID] ASC
)
WITH (SORT_IN_TEMPDB = OFF, DROP_EXISTING = OFF, IGNORE_DUP_KEY = OFF,
ONLINE = OFF) ON [PRIMARY]
GO
```

Sequences Conversion

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

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

Solution

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

Oracle

```
CREATE SEQUENCE customer_no;
INSERT INTO customers VALUES
      (customer_no.NEXTVAL, 'comment', ...);
```

SQL Server

```
CREATE SEQUENCE dbo.customer_no

INSERT INTO dbo.customers VALUES

      (NEXT VALUE FOR dbo.customer_no, 'comment', ...)
```

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

Azure SQL DB doesn't support sequence objects and the way to convert them to this version of SQL Server is described in the section Conversion of Sequences to Azure SQL DB.

SQL Server Scenario 1: Converting an ORACLE Table with Automatically Generated Primary Key

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

ORACLE

```
CREATE SEQUENCE seq1;

...

INSERT INTO t1 (id, name)

  VALUES (seq1.NEXTVAL, 'name');

INSERT INTO t2 (id, name)

  VALUES (seq1.CURRVAL, 'name');

...
```

SQL Server

```
CREATE SEQUENCE seq1

...

declare @newid int;

select @newid = NEXT VALUE FOR seq1;

INSERT INTO t1 (id, name)

  VALUES (@newid, 'name');

INSERT INTO t2 (id, name)

  VALUES (@newid, 'name');

...
```

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

SQL Server Scenario 2: Converting an Auxiliary Table Representing an ORACLE Sequence

In the second scenario, an ORACLE sequence is used in a way that is incompatible with SQL Server sequence. For example, NEXTVAL and CURRVAL of sequence can be used in different procedures or application modules.

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

```
create table MY_SEQUENCE (
        id int IDENTITY(1 /* seed */, 1 /* increment*/ )
)
go
```

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

```
declare @tran bit,
   @nextval int
set @tran = 0
if @@trancount > 0
   begin
      save transaction seq
      set @tran = 1
   end
else begin transaction
insert into MY_SEQUENCE default values
set @nextval = SCOPE_IDENTITY()
if @tran=1
   rollback transaction seq
else rollback
```

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

Conversion of Sequences to Azure SQL DB

To generate next sequence value SSMA inserts a row in emulation table, this will trigger the identity field to create new value and then it can be referenced using

SCOPE_IDENTITY function. The procedure db_sp_get_next_sequence_value is used to do all this and it returns new value using output parameter.

But in Oracle, .NEXTVAL is a function and can be used wherever function usage is allowed while procedure call isn’t possible. It is emulated using xp_ora2ms_exec extended stored procedure to call the implementation procedure from function, but for Azure SQL DB another way is needed to handle it.

In some places such usage can be easily done by executing procedure, storing the result in a temporary variable and then using the variable instead of the function call.

Depending of the usage of .NEXTVAL function, SSMA can either convert the function call to the procedure ssma_oracle.db_sp_get_next_sequence_value call or mark it with error proposing to rewrite the code with this procedure manually.

Oracle

```
declare
      val int;
begin
   if (Work.customers_seq.nextval != val) then
      val := 0;
   end if;

   insert into Work.Customers
      values (Work.customers_seq.nextval, 'Customer1');
end;
```

Azure SQL DB

```
BEGIN
   DECLARE
      @val int

   DECLARE
      @nextval numeric(38, 0)

   EXECUTE ssma_oracle.db_sp_get_next_sequence_value N'WORK',
N'CUSTOMERS_SEQ', @nextval  OUTPUT

   IF (@nextval != @val)
      SET @val = 0

   DECLARE
      @nextval$2 numeric(38, 0)

   EXECUTE ssma_oracle.db_sp_get_next_sequence_value N'WORK',
N'CUSTOMERS_SEQ', @nextval$2  OUTPUT

   INSERT WORK.CUSTOMERS(COL1, COL2)
      VALUES (@nextval$2, 'Customer1')

END
GO
```

Migrating Hierarchical Queries

This section describes problems and solutions when migrating Oracle hierarchical queries. Oracle provides the following syntax elements to build hierarchical queries:

1. The START WITH condition. Specifies the hierarchy's root rows.
2. The CONNECT BY condition. Specifies the relationship between the hierarchy's parent rows and child rows.
3. The PRIOR operator. Refers to the parent row.
4. The CONNECT_BY_ROOT operator. Retrieves the column value from the root row.
5. The NO_CYCLE parameter. Instructs the Oracle Database to return rows from a query, even if a cycle exists in the data.
6. The LEVEL, CONNECT_BY_ISCYCLE, and CONNECT_BY_ISLEAF pseudocolumns.
7. The SYS_CONNECT_BY_PATH function. Retrieves the path from the root to node.
8. The ORDER SIBLINGS BY clause. Applies ordering to the siblings of the hierarchy.

Oracle processes hierarchical queries in this order:

1. Evaluates a join first, if one is present, whether the join is specified in the FROM clause or with WHERE clause predicates.
2. Evaluates the CONNECT BY condition.
3. Evaluates any remaining WHERE clause predicates.

Oracle then uses the information from these evaluations to form the hierarchy as follows:

4. Oracle selects the hierarchy’s root row(s) (those rows that satisfy the START WITH condition).
5. Oracle selects each root row's child rows. Each child row must satisfy the CONNECT BY condition with respect to one of the root rows.
6. Oracle selects successive generations of child rows. Oracle first selects the children of the rows returned in Step 2, and then the children of those children, and so on. Oracle always selects children by evaluating the CONNECT BY condition with respect to a current parent row.
7. If the query contains a WHERE clause without a join, Oracle eliminates all rows from the hierarchy that do not satisfy the WHERE clause's conditions. Oracle evaluates that condition for each row individually, rather than removing all the children of a row that does not satisfy the condition.
8. Oracle returns the rows in the order shown in Figure 3. In the figure, children appear below their parents.

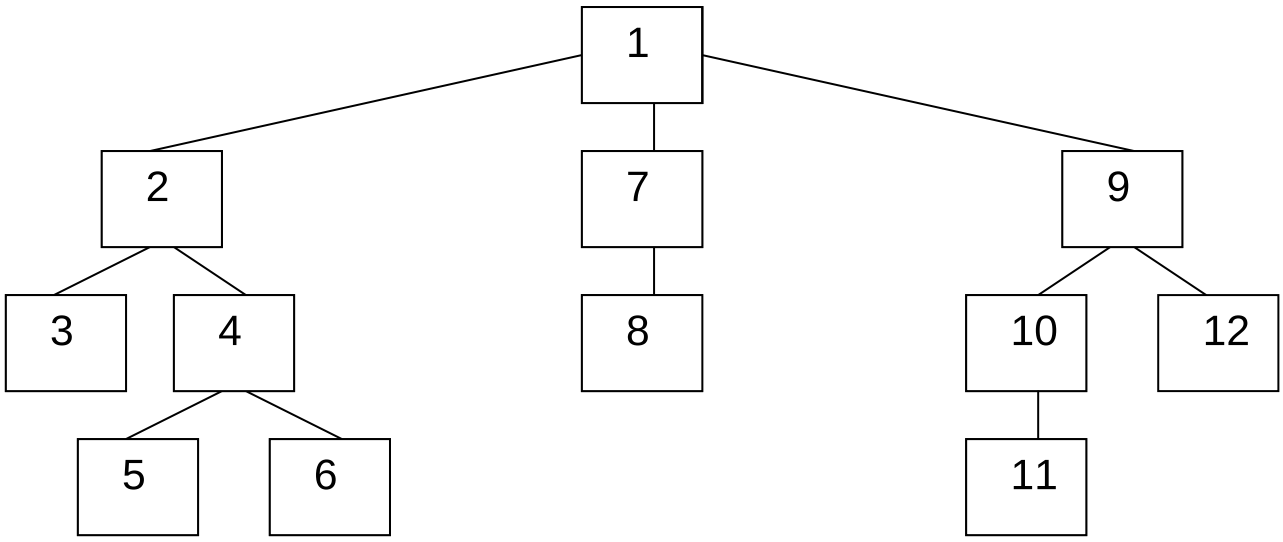


Figure 3: An example of the Oracle tree traversal order

In SQL Server 2014, you can use a recursive common table expression (CTE) to retrieve hierarchical data. For more information about the recursive CTE, see Recursive Queries Using Common Table Expression (http://msdn.microsoft.com/en-us/library/ms186243.aspx) in SQL Server Books Online.

To migrate an Oracle hierarchical query, follow these common rules:

- Use the START WITH condition in the anchor member subquery of the CTE. If there is no START WITH condition, the result of the anchor member subquery should consist of all root rows. Because the START WITH condition is processed before the WHERE condition, ensure that the anchor member subquery returns all necessary rows. This is sometimes needed to move some WHERE conditions from the CTE to the base query.

- Use the CONNECT BY condition in the recursive member subquery. The result of the recursive member subquery should consist of all child rows joined with the CTE itself on the CONNECT BY condition. Use the CTE itself as the inner join member in the recursive subquery. Replace the PRIOR operator with the CTE recursive reference.

- The base query consists of the selection from the CTE, and the WHERE clause to provide all necessary restrictions.

- Emulate the LEVEL pseudocolumn with a simple expression as described in SQL Server Books Online for SQL Server 2014.

- Emulate the **sys_connect_by_path** function with an expression that concatenates column values from recursive CTE references.

This approach makes hierarchical data retrieval possible. But the way to traverse trees is different in Oracle. To emulate the way Oracle orders return data, you can create additional expressions to use in the ORDER BY clause. The expression should evaluate some path from the root to the specific row by using a unique row number at each tree level. You can use the ROW_NUMBER function for this purpose. You can also add expressions based on the column's values to provide ORDER SIBLINGS BY functionality.

You can use GROUP BY and HAVING clauses only in the base query.

SQL Server 2014 cannot detect the cycles in a hierarchical query. You can control the recursion level with the MAXRECURSION query hint.

Note that SSMA does not support the following features:

- The CONNECT_BY_ROOT operator
- The NOCYCLE parameter
- The CONNECT_BY_ISCYCLE and CONNECT_BY_ISLEAF pseudocolumns
- The SYS_CONNECT_BY_PATH function
- The ORDER SIBLINGS BY clause

Example:

The following example code demonstrates how to migrate a simple hierarchical query:

Oracle

```
SELECT "NAME", "PARENT", LEVEL
  FROM COMPANY
 START WITH ("NAME" = 'Company Ltd')
 CONNECT BY ("PARENT" = PRIOR "NAME");
```

SQL Server

```
WITH
   h$cte AS
   (
      SELECT COMPANY.NAME, COMPANY.PARENT, 1 AS LEVEL,
CAST(row_number() OVER(
         ORDER BY @@spid) AS varchar(max)) AS path
      FROM dbo.COMPANY
      WHERE ((COMPANY.NAME = 'Company Ltd'))
       UNION ALL
      SELECT COMPANY.NAME, COMPANY.PARENT, h$cte.LEVEL + 1 AS LEVEL,
path + ',' + CAST(row_number() OVER(
         ORDER BY @@spid) AS varchar(max)) AS path
      FROM dbo.COMPANY, h$cte
      WHERE ((COMPANY.PARENT = h$cte.NAME))
   )

   SELECT h$cte.NAME, h$cte.PARENT, h$cte.LEVEL
   FROM h$cte
   ORDER BY h$cte.path
```

Note The ROW_NUMBER() function evaluates the path column to provide Oracle nodes ordering.

Emulating Oracle Exceptions

This section describes problems and solutions for migrating Oracle exception mechanisms. The Oracle exception model differs from Microsoft SQL Server 2014 both in exception raising and exception handling. It is preferable to use the SQL Server exceptions model during Oracle PL/SQL code migration. At the same time, SSMA provides common emulation methods to cover almost all Oracle exception-model features.

Exception Raising

The Oracle exception raising model comprises the following features:

- The SELECT INTO statement causes an exception if not exactly one row is returned.
- The RAISE statement can raise any exception, including system errors.
- User-defined exceptions can be named and raised by name.
- The RAISE_APPLICATION_ERROR procedure can generate exceptions with a custom number and message.

If the SELECT statement can return zero, one, or many rows, it makes sense to check the number of rows by using the @@ROWCOUNT function. Its value can be used to emulate any logic that was implemented in Oracle by using the TOO_MANY_ROWS or NO_DATA_FOUND exceptions. Normally, the SELECT INTO statement should return only one row, so in most cases you don’t need to emulate this type of exception raising.

For example:

Oracle

```
BEGIN
      SELECT <expression>  INTO <variable> FROM <table>;
EXCEPTION
      WHEN NO_DATA_FOUND THEN
            <Statements>
END
```

SQL Server 2014

```
SELECT <variable> = <expression> FROM <table>

IF @@ROWCOUNT = 0
  BEGIN
      <Statements>
      RETURN
  END
```

Also, PL/SQL programs can sometimes use user-defined exceptions to provide business logic. These exceptions are declared in the PL/SQL block's declaration section. In Transact-SQL, you can replace that behavior by using flags or custom error numbers.

For example:

Oracle

```
declare
      myexception exception;
BEGIN
…
      IF <condition>  THEN
            RAISE myexception;
      END IF;
…
EXCEPTION
      WHEN myexception THEN
            <Statements>
END
```

SQL Server 2014

```
BEGIN TRY
…
      IF <condition>
            RAISERROR (‘myexception’, 16, 1)
…
END TRY
BEGIN CATCH

      IF ERROR_MESSAGE() = ‘myexception’
        BEGIN
            <Statements>
        END
      ELSE THROW

END CATCH
```

If the user-defined exception is associated with some error number by using pragma EXCEPTION_INIT, you can handle the system error in the CATCH block as described later.

To emulate the **raise_application_error** procedure and the system predefined exception raising, you can use the RAISERROR statement with a custom error number and message. Also, change the application logic in that case to support SQL Server 2014 error numbers.

Note that SQL Server 2014 treats exceptions with a severity of less than 11 as information messages. To interrupt execution and pass control to a CATCH block, the exception severity must be at least 11. (In most cases you should use a severity level of 16.)

Exception Handling

Oracle provides the following exception-handling features:

- The EXCEPTION block
- The WHEN … THEN block
- The SQLCODE and SQLERRM system functions
- Exception reraising

Transact-SQL implements error handling with a TRY...CATCH construct. To provide exception handling, place all “trying” statements into a BEGIN TRY … END TRY block, while placing the exception handler itself into a BEGIN CATCH … END CATCH block. TRY … CATCH blocks also can be nested.

To recognize the exception (WHEN … THEN functionality), you can use the following system functions:

- ERROR_NUMBER
- ERROR_LINE
- ERROR_PROCEDURE
- ERROR_SEVERITY
- ERROR_STATE
- ERROR_MESSAGE

You can use the ERROR_NUMBER and ERROR_MESSAGE functions instead of the SQLCODE and SQLERRM Oracle functions. Note that error messages and numbers are different in Oracle and SQL Server, so they should be translated during migration.

For example:

Oracle

```
BEGIN
…
      INSERT INTO <table> VALUES …
…
EXCEPTION
…
      WHEN DUP_VAL_ON_INDEX THEN
            <Statements>
…
END
```

SQL Server 2014

```
BEGIN TRY
…
      INSERT INTO <table> VALUES …
…
END TRY
BEGIN CATCH
…
      IF ERROR_NUMBER() = 2627
            <Statements>
      ELSE THROW
…
END CATCH
```

SSMA Exceptions Migration to SQL Server 2014

Next, let's examine how SSMA provides a common approach to full emulation of Oracle exception functionality.

Oracle exceptions are encoded into a character string according to the following rules:

- Predefined exceptions (exceptions declared in some system package and not assigned to any error number) are encoded this way:

  ```
  oracle:{<OWNER_NAME>|<PACKAGE_NAME>|<EXCEPTION_NAME>}
  ```

  Where:

    - PACKAGE_NAME is the package name where the exception is declared in upper case.
    - OWNER_NAME is the owner name of the package, in uppercase.
    - EXCEPTION_NAME is the exception name itself, in uppercase.

- User-defined exceptions names declared in modules such as stored procedures acquire the “local:” prefix:

  ```
  local:oracle:{<OWNER_NAME>|<MODULE_NAME>}:<EXCEPTION_NAME>:N
  ```

  Where:

    - OWNER_NAME is the owner name of the module where the exception is declared.
    - MODULE_NAME is the name of the stored procedure where the exception is declared.
    - N is an integer value that provides scope name uniqueness.

- User-defined exception names declared in anonymous PL/SQL blocks (test statements) have an additional PL\SQL keyword:

  ```
        local:PL\SQL:<EXCEPTION_NAME>:N
  ```

  Where N is the integer value that provides scope name uniqueness.

- To support Oracle error numbers, system errors are stored in the following format:

  ```
   'ORAXXXXXX'
  ```

During migration SSMA performs the following steps:

1. All statements between BEGIN and EXCEPTION are enclosed with BEGIN TRY … END TRY.
2. An exception handler is placed into BEGIN CATCH … END CATCH.
3. Error numbers are translated to Oracle format by using the **ssma_oracle.db_error_get_oracle_exception_id()** function. That function returns an exception identifier as a character string, as described earlier. Each WHEN…THEN statement is migrated to an IF statement that compares the exception identifier to constant exception names that are translated according to the same rules.
4. The exception handler for OTHERS, if any, is migrated to an ELSE statement.
5. If there is no OTHERS exception handler, the exception is reraised in an ELSE statement by means of **ssma_oracle.ssma_rethrowerror** procedure.
6. To emulate predefined Oracle exceptions NO_DATA_FOUND and TOO_MANY_ROWS there is then only case to be taken into account:
    - Exceptions can be raised as a part of the program execution after all SELECT statements, except statements which contain an aggregate function without grouping (because such query always returns at least one row)

```
DECLARE @rc int = @@ROWCOUNT
```

```
IF      @rc = 0 RAISERROR (59999, 16, 1, N'ORA+00100')

ELSE IF @rc > 1 RAISERROR (59999, 16, 1, N'ORA-01422')
```

7. The number 59999 is used for all Oracle system, user-defined, or predefined exceptions.
8. The RAISE statement is migrated to the RAISERROR statement with a 59999 error number and the exception identifier as a message. The exception identified is formed as described earlier.
9. To emulate the **raise_application_error** procedure, there is the additional error number 59998. The procedure call is replaced by a RAISERROR call with error number 59998 and the following string as a message:

```
‘ORA<error_number>:<message>’
```

For example:

```
RAISERROR (59998, 16, 1,’ORA-20000:test’)
```

10. All exceptions are raised with severity level 16 to provide handling by a CATCH block.
11. **ssma_oracle.db_error_sqlcode** user-defined function emulates the SQLCODE function. It returns an Oracle error number.
12. Either **ssma_oracle.db_error_sqlerrm_0** or **ssma_oracle.db_error_sqlerrm_1** emulates the SQLERRM function, depending on the parameters.
13. SSMA does not support using the SQLCODE and SQLERRM functions outside of an EXCEPTION block (by the way in ORACLE in this case SQLCODE=0 and SQLERRM=’ORA-0000: normal, successful completion’ respectively).

SSMA Exceptions Migration to Azure SQL DB

Now, let's examine how SSMA provides migration of Oracle exceptions to Azure SQL DB.

Emulation of Oracle exceptions in SQL Server is based on user-defined error messages stored by using sp_addmessage system stored procedure. Azure SQL DB doesn't support such behavior. That’s why RAISERROR statement can’t be used to generate exception with specified both error code and error message. But instead Azure SQL DB supports THROW statement which first appeared in SQL Server 2012.

Thus, in SQL Server 2014 and Azure SQL DB, a new mechanism exists to generate exceptions – THROW statement.

```
THROW [ { error_number | @local_variable },

        { message | @local_variable },

        { state | @local_variable } ]
```

```
       [ ; ]
```

In SQL Server exception emulation, two user defined error messages with placeholder are used when SSMA generates exceptions using RAISERROR statement.

Code	Message
59998	'%s'
59999	'SSMA Oracle exception emulation for [%s]'

For THROW, the same error messages are generated inline.

SQL Server 2014

```
RAISERROR(59998, 16, 1, @db_raise_application_error_message)

RAISERROR(59999, 16, 1, @ex_some$exception)
```

Azure SQL DB

```
; THROW 59998, @db_raise_application_error_message, 1;


DECLARE @tmp nvarchar(4000) = N'SSMA Oracle exception emulation for ['
+
                                  @ex_some$exception + ']';
; THROW 59999, @tmp, 1;
```

For SQL Server, user defined exceptions are emulated using varchar variable that store ORA-***** code. The same variable is used in CATCH block to differentiate specific exceptions. For SQL Server RAISERROR statement it isn't a problem that this variable contains '%' sign, but it is a problem for THROW. For THROW statement, this sign is removed from exception message and its contents is concatenated to the exception message only inside a CATCH block.

Also RAISERROR doesn't provide a way to recreate exception that was already caught. SSMA converts it to SQL Server 2014 with special block of statements and ssma_oracle.ssma_rethrowerror procedure. But in Azure SQL DB, THROW statement ability is used to rethrow exception.

The exception should be rethrowed if there is an exception handling block without WHEN OTHERS clause or when exception is regenerated explicitly by RAISE statement in Oracle source code.

Oracle

```
DECLARE
   x INT;
   ex_low_salary EXCEPTION;
   PRAGMA EXCEPTION_INIT(ex_low_salary, -20005);
BEGIN
   RAISE ex_low_salary;
```

```
EXCEPTION
   WHEN ex_low_salary THEN
      SELECT 2 into x FROM dual;
END;
```

Azure SQL DB

```
DECLARE @x int, @ex_low_salary$exception nvarchar(1000)
BEGIN TRY

   SET @ex_low_salary$exception = N'ORA-20005'
   declare @tmp nvarchar(4000) = N'SSMA Oracle exception emulation for 
[' +
                                  @ex_low_salary$exception + ']';

   ;THROW 59999, @tmp, 1

END TRY
BEGIN CATCH
   DECLARE @errornumber int
   SET @errornumber = ERROR_NUMBER()

   DECLARE @errormessage nvarchar(4000)
   SET @errormessage = ERROR_MESSAGE()

   DECLARE @exceptionidentifier nvarchar(4000)
   SELECT @exceptionidentifier = 
ssma_oracle.db_error_get_oracle_exception_id(@errormessage, 
@errornumber)

   IF (@exceptionidentifier LIKE @ex_low_salary$exception + '%')
      SELECT @x = 2
   ELSE
      THROW;  -- regenerate exception block

END CATCH
```

Migrating Oracle Cursors

This section describes problems and solutions for Oracle cursor migration. Keep in mind that a packaged cursor needs special handling during conversion. For more information, see Emulating Oracle Packages.

Oracle always requires that cursors be used with SELECT statements, regardless of the number of rows requested from the database. In Microsoft SQL Server 2014, a SELECT statement that is not enclosed within a cursor returns rows to the client as a default result set. This is an efficient way to return data to a client application.

SQL Server 2014 provides two interfaces for cursor functions:

- When cursors are used in Transact-SQL batches or stored procedures, SQL statements can declare, open, and fetch from cursors—as well as positioned updates and deletes.
- When cursors from a DB-Library, ODBC, or OLE DB program are used, the SQL Server client libraries transparently call built-in server functions to handle cursors more efficiently.

Syntax

The following table shows cursor statement syntax in both platforms.

Operation	Oracle	Microsoft SQL Server

Declaring a cursor	`CURSOR cursor_name [(cursor_parameter(s))] IS select_statement;`	`ISO Syntax`
		`DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ] CURSOR`
		`FOR select_statement`
		`[ FOR { READ ONLY | UPDATE [ OF column_name [ ,...n ] ] } ]`
		`Transact-SQL Extended Syntax`
		`DECLARE cursor_name CURSOR [LOCAL | GLOBAL] [FORWARD_ONLY | SCROLL] [STATIC | KEYSET | DYNAMIC | FAST_FORWARD]`

Operation	Oracle	Microsoft SQL Server
		`[READ_ONLY | SCROLL_LOCKS | OPTIMISTIC] [TYPE_WARNING] FOR select_statement [FOR UPDATE [OF column_name [,…n]]]`
Ref cursor type definition	`TYPE type_name IS REF CURSOR [RETURN { {db_table_name | cursor_name | cursor_variable_name} % ROWTYPE | record_name % TYPE | record_type_name | ref_cursor_type_name}];`	See below.
Opening a cursor	`OPEN cursor_name [(cursor_parameter(s))];`	`OPEN cursor_name`
Cursor attributes	`{ cursor_name | cursor_variable_name | :host_cursor_variable_name} % {FOUND | ISOPEN | NOTFOUND | ROWCOUNT}`	See below.
SQL cursors	`SQL % {FOUND | ISOPEN | NOTFOUND | ROWCOUNT | BULK_ROWCOUNT(index) | BULK_EXCEPTIONS(index).{ERROR_INDEX | ERROR_CODE}}`	See below.
Fetching from cursor	`FETCH cursor_name INTO variable(s)`	`FETCH [[NEXT | PRIOR | FIRST | LAST | ABSOLUTE {n | @nvar} | RELATIVE {n | @nvar}] FROM] cursor_name [INTO @variable(s)]`
Update fetched row	`UPDATE table_name SET statement(s)… WHERE CURRENT OF cursor_name;`	`UPDATE table_name SET statement(s)… WHERE CURRENT OF cursor_name`
Delete fetched row	`DELETE FROM table_name WHERE CURRENT OF cursor_name;`	`DELETE FROM table_name WHERE CURRENT OF`

Operation	Oracle	Microsoft SQL Server
		`cursor_name`
Closing cursor	`CLOSE cursor_name;`	`CLOSE cursor_name`
Remove cursor data structures	N/A	`DEALLOCATE cursor_name`
OPEN … FOR cursors	`OPEN {cursor_variable_name | :host_cursor_variable_name}` `FOR dynamic_string [using_clause]`	See below.

Declaring a Cursor

Although the Transact-SQL DECLARE CURSOR statement does not support cursor arguments, it does support local variables. The values of these local variables are used in the cursor when it is opened. Microsoft SQL Server 2014 offers numerous additional capabilities in its DECLARE CURSOR statement.

The INSENSITIVE option defines a cursor that makes a temporary copy of the data to be used by that cursor. The temporary table answers all of the requests to the cursor. Consequently, modifications made to base tables are not reflected in the data returned by fetches made to that cursor. Data accessed by this cursor type cannot be modified.

Applications can request a cursor type, and then execute a Transact-SQL statement that is not supported by server cursors of the type requested. SQL Server returns an error that indicates that the cursor type has changed, or, given a set of factors, implicitly converts a cursor.

The following table shows the factors that trigger SQL Server to implicitly convert a cursor from one type to another.

Step	Conversion triggered by	Forward-only	Keyset-driven	Dynamic	Go to step
1	Query FROM clause references no tables	Becomes static	Becomes static	Becomes static	Done
2	Query contains: select list aggregates GROUP BY UNION DISTINCT HAVING	Becomes static	Becomes static	Becomes static	Done
3	Query generates an internal work table, for example the columns of an ORDER BY are not covered by an index	Becomes keyset		Becomes keyset	5
4	Query references remote tables in linked servers	Becomes keyset		Becomes keyset	5
5	Query references at least one table without a unique index. Transact-SQL cursors only.		Becomes static		Done

The SCROLL option allows backward, absolute, and relative fetches, and also forward fetches. A scroll cursor uses a keyset cursor model in which committed deletes and updates made to the underlying tables by any user are reflected in subsequent fetches. This is true only if the cursor is not declared with the INSENSITIVE option.

If the READ ONLY option is chosen, updates are prevented from occurring against any row within the cursor. That option overrides the default capability of a cursor to be updated.

The UPDATE [OF column_list] statement defines updatable columns within the cursor. If [OF column_list] is supplied, only the columns listed allow modifications. If a list is not supplied, all columns can be updated, unless the cursor is defined as READ ONLY.

Note that the name scope for a SQL Server cursor is the connection itself. That differs from the name scope of a local variable. A second cursor with the same name as an existing cursor on the same user connection cannot be declared until the first cursor is deallocated.

Following are descriptions of the SSMA algorithm of cursor conversion for several specific cases.

- If the cursor is declared in the local subprogram, SSMA converts it to:

  ```
  DECLARE cursor_name CURSOR LOCAL FOR select_statement
  ```

  SSMA puts this cursor declaration directly before the OPEN statement that opens the cursor and removes the RETURN clause.

  Instead of the cursor declaration, SSMA generates a variable declaration.

- If the cursor is declared as a public packaged cursor, SSMA converts it into a global cursor:

  ```
  DECLARE cursor_name CURSOR FOR select_statement
  ```

For more information, see Emulating Oracle Packages.

- SSMA declares a local variable for each parameter with the following naming pattern:

  ```
  @CURSOR_PARAM_<cursor_name>_<parameter_name>
  ```

  The data type is converted according to the effective SSMA type mapping for local variables.

- SSMA removes a REF cursor definition and converts it to a variable declaration as follows:

```
cursor_variable_declaration ::=
```

```
   cursor_variable_name type_name;
```

Convert to:

```
@cursor_variable_name CURSOR;
```

Opening a Cursor

Unlike PL/SQL, Transact-SQL does not support passing arguments to a cursor when it is opened. When a Transact-SQL cursor is opened, the result set membership and ordering are fixed. Updates and deletes that have been committed against the cursor's base tables by other users are reflected in fetches made against all cursors defined without the INSENSITIVE option. In the case of an INSENSITIVE cursor, a temporary table is generated.

SSMA tests to see whether the cursor was declared with formal cursor parameters. For each formal cursor parameter, generate a SET statement before the cursor declaration to assign the actual cursor parameter to the appropriate local variable:

```
SET @CURSOR_PARAM_<cursor_name>_<parameter_name> =
actual_cursor_parameter
```

If there is no actual parameter for the formal parameter, use a DEFAULT expression as declared in the cursor parameter declaration:

```
SET @CURSOR_PARAM_<cursor_name>_<parameter_name> = expression
```

Fetching Data

Oracle cursors can move in a forward direction only—there is no backward or relative scrolling capability. SQL Server 2014 cursors can scroll forward and backward with the fetch options shown in the following table. You can use these fetch options only if the cursor is declared with the SCROLL option.

Scroll option	Description
NEXT	Returns the result set's first row if this is the first fetch against the cursor; otherwise, moves the cursor one row in the result set. NEXT is the primary method for moving through a result set. NEXT is the default cursor fetch.
PRIOR	Returns the previous row in the result set.
FIRST	Moves the cursor to the first row in the result set and returns the first row.
LAST	Moves the cursor to the last row in the result set and returns the last row.

Scroll option	Description
ABSOLUTE *n*	Returns the *n*th row in the result set. If *n* is a negative value, the returned row is the *n*th row counting backward from the last row of the result set.
RELATIVE *n*	Returns the *n*th row after the currently fetched row. If *n* is a negative value, the returned row is the *n*th row counting backward from the cursor's relative position.

The Transact-SQL FETCH statement does not require the INTO clause. If return variables are not specified, the row is automatically returned to the client as a single-row result set. However, if your procedure must get the rows to the client, a noncursor SELECT statement is much more efficient.

Issues

SSMA recognizes the following FETCH formats:

- FETCH INTO <record>: SSMA splits the record into its components and fetches each variable separately.
- FETCH … BULK COLLECT INTO

The @@FETCH_STATUS function is updated following each FETCH. This function resembles the PL/SQL CURSOR_NAME%FOUND and CURSOR_NAME%NOTFOUND variables. The @@FETCH_STATUS function is set to the value of 0 following a successful fetch. If the fetch tries to read beyond the end of the cursor, a value of -1 is returned. If the requested row was deleted from the table after the cursor was opened, the @@FETCH_STATUS function returns -2. The value of -2 usually occurs only in a cursor that was declared with the SCROLL option. That variable must be checked following each fetch to ensure the validity of the data.

How SSMA converts cursor attributes

SSMA converts cursor attributes as follows:

- FOUND attribute: Converts to @@FETCH_STATUS = 0
- NOTFOUND attribute: Converts to @@FETCH_STATUS <> 0
- ISOPEN attribute: Converts as follows:
    - For global cursors:

```
(CURSOR_STATUS('global', N'<cursor_name>') > -1)
```

    - For local cursors:

      ```
      (CURSOR_STATUS(‘local’, N’<cursor_name>’) > -1)
      ```

    - For a cursor variable:

      ```
      (CURSOR_STATUS(‘variable’, N’@<cursor_variable_name>’) > -1)
      ```

- ROWCOUNT attribute: To convert ROWCOUNT, SSMA does the following:

  1. It generates a declaration of an INT variable with the name @v_<cursor_name | cursor_variable_name >_rowcount at the beginning of the block where cursor was declared (see Declaring a Cursor).
  2. Before the OPEN statement for the cursor or cursor variable, it puts a variable initialization code:

     ```
     SET @v_<cursor_name | cursor_variable_name >_rowcount = 0
     ```

  3. Immediately after the cursor FETCH statement, it puts:

     ```
     IF @@FETCH_STATUS = 0
     SET @v_<cursor_name | cursor_variable_name >_rowcount =
     @v_<cursor_name | cursor_variable_name >_rowcount + 1
     ```

  4. SSMA converts cursor_name%ROWCOUNT to:
     `@v_<cursor_name | cursor_variable_name >_rowcount`

How SSMA converts SQL cursor attributes

- FOUND: Converts to (@@ROWCOUNT > 0)
- NOTFOUND: Converts to (@@ROWCOUNT = 0)
- ISOPEN: Converts to any condition that is always false, for example (1=2)
- ROWCOUNT: Converts to @@ROWCOUNT. For example:

Oracle

```
IF SQL%FOUND THEN …;
```

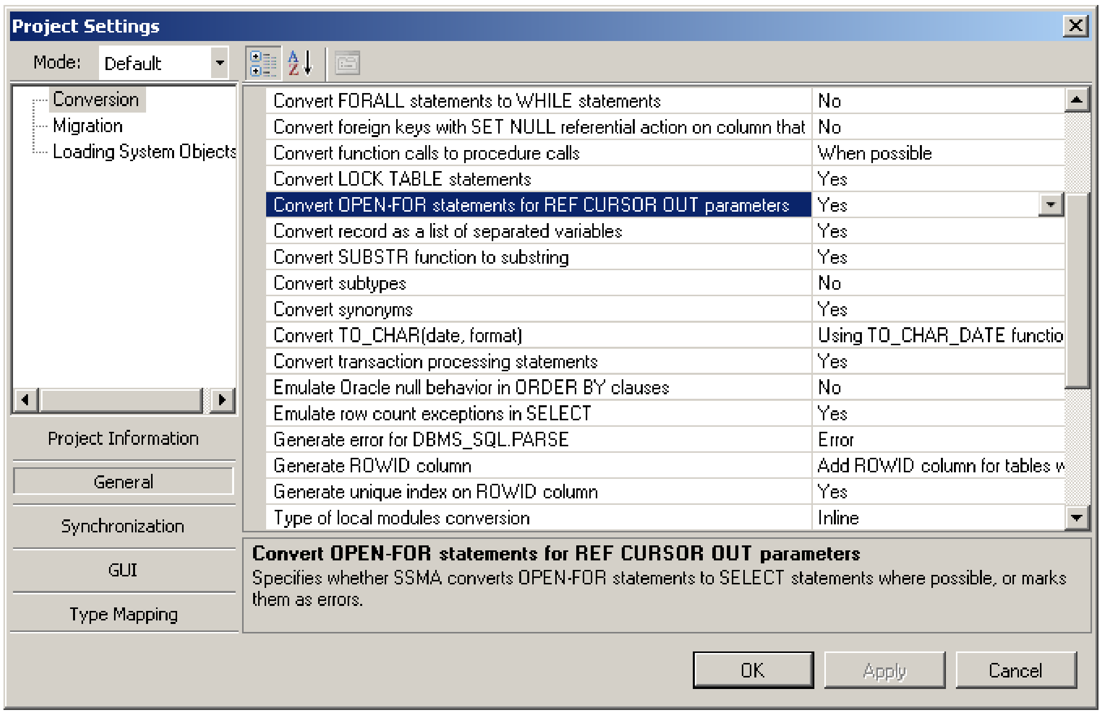
SQL Server 2014

```
IF @@ROWCOUNT > 0 …
```

SQL Server does not support Oracle’s cursor FOR loop syntax, but SSMA can convert these loops. See the examples in the previous section.

How SSMA converts OPEN … FOR cursors

The SSMA conversion option Convert OPEN-FOR statement for REF CURSOR OUT parameters (see Figure 4) is used because there is an ambiguity when a REF CURSOR output parameter is opened in the procedure. The REF CURSOR might be fetched in the caller procedure (SSMA does not support this usage) or used directly by the application (SSMA can handle this if the option is set to Yes).



Convert FORALL statements to WHILE statements	No
Convert foreign keys with SET NULL referential action on column that	No
Convert function calls to procedure calls	When possible
Convert LOCK TABLE statements	Yes
Convert OPEN-FOR statements for REF CURSOR OUT parameters	Yes
Convert record as a list of separated variables	Yes
Convert SUBSTR function to substring	Yes
Convert subtypes	No
Convert synonyms	Yes
Convert TO_CHAR(date, format)	Using TO_CHAR_DATE functio
Convert transaction processing statements	Yes
Emulate Oracle null behavior in ORDER BY clauses	No
Emulate row count exceptions in SELECT	Yes
Generate error for DBMS_SQL.PARSE	Error
Generate ROWID column	Add ROWID column for tables w
Generate unique index on ROWID column	Yes
Type of local modules conversion	Inline

Convert OPEN-FOR statements for REF CURSOR OUT parameters
Specifies whether SSMA converts OPEN-FOR statements to SELECT statements where possible, or marks them as errors.

Figure 4: Setting the Convert OPEN-FOR statement for REF CURSOR OUT parameters SSMA conversion option

Generally, an OPEN-FOR statement is converted in the following way:

- If the OPEN-FOR statement is used for a local cursor variable, SSMA converts it to:

  ```
  SET @cursor_variable_name = CURSOR FOR select_statement
  ```

- If the OPEN-FOR statement is used for an output procedure parameter and the option is set to ON, it’s converted to:

```
select_statement
```

This returns a result set to the client application.

- If the OPEN-FOR statement is used for an output procedure parameter and the option is set to OFF, SSMA generates the following error:
  "Conversion of OPEN-FOR statement is disabled."

The OPEN-FOR-USING statement, when it is used for a local cursor variable, is converted somewhat differently, as in the following steps:

1. SSMA generates the following code:

   ```
   DECLARE
      @auxiliary_cursor_definition_sql$N NVARCHAR(max),
      @auxiliary_exec_param$N NVARCHAR(max)

   IF (cursor_status('variable', N'<cursor_variable_name>') >  -2)
         DEALLOCATE <cursor_variable_name>
   SET @auxiliary_exec_param$N = '[@auxiliary_paramN <datatype>
   [OUTPUT],] … @auxiliary_tmp_cursor$N cursor OUTPUT'
   ```

2. Then SSMA generates the following error message: 'OPEN ... FOR statement will be converted, but the dynamic string must be converted manually.'

3. It adds the following line into the Attempted target code section:

   ```
   SET @auxiliary_cursor_definition_sql$N = ('SET
   @auxiliary_tmp_cursor =  CURSOR    LOCAL FOR ' +
   <dynamic_string>+ '; OPEN @auxiliary_tmp_cursor')
   ```

   SSMA uses integer value N as part of declared variable names to provide scope name uniqueness.

   The parameter @auxiliary_paramN is declared in @auxiliary_exec_param$N for every bind_argument of the using_clause. SSMA determines the data type of the argument to declare the parameters. It also specifies OUTPUT in case of a bind_argument specified with an OUT or an IN_OUT option.

4. SSMA generates the following code:

   ```
   EXEC sp_executesql @auxiliary_cursor_definition_sql$N,
   @auxiliary_exec_param$N, [bind_argument [OUTPUT], ]…
   cursor_variable_name OUTPUT
   ```

   Where bind_argument is the bind_argument from the using_clause. Specify

OUTPUT for the bind arguments that were declared with OUTPUT specified in @auxiliary_exec_param$N.

When used for an ouput procedure parameter, the OPEN-FOR-USING statement and the **Convert OPEN-FOR statement for REF CURSOR OUT parameters** option is set to ON.

1. SSMA generates the following code:

   ```
   DECLARE
             @auxiliary_cursor_definition_sql$N NVARCHAR(max),
             @auxiliary_exec_param$N NVARCHAR(max)
   SET @auxiliary_exec_param$N = '[@auxiliary_paramN <datatype> [OUTPUT]]'
   ```

2. Then it generates the following error message: “OPEN ... FOR statement will be converted, but the dynamic string must be converted manually.”
3. SSMA puts the following line into the Attempted target code section:

   ```
   SET @auxiliary_cursor_definition_sql$N = ( <dynamic_string>)
   ```

   SSMA uses the integer value N as part of the declared variable names to provide scope name uniqueness.

4. The @auxiliary_paramN parameter is declared in @auxiliary_exec_param$N for every bind_argument of the using_clause. SSMA determines the data type of the argument to declare the parameters. It specifies OUTPUT if a bind_argument is specified with an OUT or an IN_OUT option.
5. SSMA generates the following code:

   ```
   EXEC sp_executesql @auxiliary_cursor_definition_sql$N,
   @auxiliary_exec_param$N [, bind_argument ]…
   ```

   bind_argument is the bind_argument from the using_clause.

CURRENT OF Clause

The CURRENT OF clause syntax and function for updates and deletes is the same in both PL/SQL and Transact-SQL. A positioned UPDATE or DELETE operation is performed against the current row within the specified cursor.

Closing a Cursor

The Transact-SQL CLOSE CURSOR statement closes the cursor but leaves the data structures accessible for reopening. The PL/SQL CLOSE CURSOR statement closes and releases all data structures.

Transact-SQL requires the DEALLOCATE CURSOR statement to remove the cursor data structures. The DEALLOCATE CURSOR statement differs from CLOSE CURSOR in that a closed cursor can be reopened. The DEALLOCATE CURSOR statement releases all data structures associated with the cursor and removes the definition of the cursor.

During conversion, SSMA adds a DEALLOCATE CURSOR statement. The source statement:

```
CLOSE { cursor_name | cursor_variable_name |
:host_cursor_variable_name}
```

becomes two statements in SQL Server:

```
CLOSE { cursor_name | @cursor_variable_name }
DEALLOCATE { cursor_name | @cursor_variable_name }
```

Examples of SSMA for Oracle V6.0 Conversion

FOR Loop Cursor Conversion

Oracle

```
CREATE OR REPLACE PROCEDURE db_proc_for_loop (mgr_param NUMBER)
AS
BEGIN
       DECLARE
                  CURSOR emp_cursor IS
                           SELECT empno, ename
                           FROM emp WHERE mgr = mgr_param;
       BEGIN
              FOR emp_rec IN emp_cursor
              LOOP
                    UPDATE emp SET sal = sal * 1.1;
              END LOOP;
       END;
END db_proc_for_loop;
```

SQL Server

```
CREATE PROCEDURE dbo.DB_PROC_FOR_LOOP
   @mgr_param float(53)
AS
   BEGIN
```

```
      BEGIN
         DECLARE
            @v_emp_cursor_rowcount int

         DECLARE
            @emp_rec$empno float(53),
            @emp_rec$ename varchar(max)

         DECLARE
             emp_cursor CURSOR LOCAL FORWARD_ONLY FOR
               SELECT EMP.EMPNO, EMP.ENAME
               FROM dbo.EMP
               WHERE EMP.MGR = @mgr_param

         OPEN emp_cursor

         WHILE 1 = 1
            BEGIN
               FETCH emp_cursor
                   INTO @emp_rec$empno, @emp_rec$ename

               IF @@FETCH_STATUS =  -1
                  BREAK

               UPDATE dbo.EMP
                  SET
                     SAL = EMP.SAL * 1.1

            END

         CLOSE emp_cursor
         DEALLOCATE emp_cursor
      END
   END
```

Cursor with Parameters

Oracle

```
CREATE OR REPLACE PROCEDURE  db_proc_cursor_parameters
AS
  CURSOR rank_cur (id_ NUMBER, sn CHAR)
    IS SELECT rank, rank_name
```

```
         FROM rank_table
        WHERE r_id = id_ AND r_sn = sn;
BEGIN
  OPEN rank_cur (1, 'c');
  OPEN rank_cur (2, 'd');
END;
```

SQL Server

```
CREATE PROCEDURE dbo.DB_PROC_CURSOR_PARAMETERS
AS
   BEGIN
      DECLARE
         @CURSOR_PARAM_rank_cur_id_$2 float(53)
      SET @CURSOR_PARAM_rank_cur_id_$2 = 1
      DECLARE
         @CURSOR_PARAM_rank_cur_sn$2 varchar(max)
      SET @CURSOR_PARAM_rank_cur_sn$2 = 'c'
      DECLARE
          rank_cur CURSOR LOCAL FOR
            SELECT RANK_TABLE.RANK, RANK_TABLE.RANK_NAME
            FROM dbo.RANK_TABLE
            WHERE RANK_TABLE.R_ID = @CURSOR_PARAM_rank_cur_id_$2 AND
RANK_TABLE.R_SN = @CURSOR_PARAM_rank_cur_sn$2

      OPEN rank_cur
      DECLARE
         @CURSOR_PARAM_rank_cur_id_ float(53)
      SET @CURSOR_PARAM_rank_cur_id_ = 2
      DECLARE
         @CURSOR_PARAM_rank_cur_sn varchar(max)
      SET @CURSOR_PARAM_rank_cur_sn = 'd'
      DECLARE
          rank_cur CURSOR LOCAL FOR
            SELECT RANK_TABLE.RANK, RANK_TABLE.RANK_NAME
            FROM dbo.RANK_TABLE
            WHERE RANK_TABLE.R_ID = @CURSOR_PARAM_rank_cur_id_ AND
RANK_TABLE.R_SN = @CURSOR_PARAM_rank_cur_sn

      OPEN rank_cur

   END
```

Cursor Attributes Conversion

Oracle

```
CREATE OR REPLACE PROCEDURE db_proc_cursor_attributes
AS
  ID number;
  CURSOR Cur IS SELECT ID FROM rank_table;
BEGIN
  IF NOT Cur%ISOPEN THEN
    OPEN Cur;
  END IF;
  LOOP
     FETCH Cur INTO ID;
     EXIT WHEN Cur%NOTFOUND;
     dbms_output.put_line(to_char(ID + Cur%ROWCOUNT));
  END LOOP;
  CLOSE Cur;
END;
```

SQL Server

```
CREATE PROCEDURE dbo.DB_PROC_CURSOR_ATTRIBUTES
AS
   BEGIN
      DECLARE
         @ID float(53),
         @v_Cur_rowcount int

      IF NOT CURSOR_STATUS('local', N'Cur') >  -1
         BEGIN

            DECLARE
                Cur CURSOR LOCAL FOR
                  SELECT RANK_TABLE.ID
                  FROM dbo.RANK_TABLE
            SET @v_Cur_rowcount = 0
            OPEN Cur

         END

      WHILE 1 = 1
         BEGIN
```

```
            FETCH Cur
                INTO @ID
            IF @@FETCH_STATUS = 0
               SET @v_Cur_rowcount = @v_Cur_rowcount + 1
            IF @@FETCH_STATUS <> 0
               BREAK

            PRINT CAST(@ID + CAST(@v_Cur_rowcount AS float(53)) AS
varchar(max))

         END

      CLOSE Cur
      DEALLOCATE Cur
   END
```

CONTINUE Statement of a LOOP

The CONTINUE statement exits the current iteration of a loop, either conditionally or unconditionally, and transfers control to the next iteration of either the current loop or an enclosing labeled loop.

If a CONTINUE statement exits a cursor FOR loop prematurely (for example, to exit an inner loop and transfer control to the next iteration of an outer loop), the cursor closes (in this context, CONTINUE works like GOTO).

Example

Oracle

```
begin
        <<OuterLoop>>
        for outer in 1..10 loop
           dbms_output.put_line('-> outer='||outer);
           for inner in 1..10 loop
                continue OuterLoop when inner > 5;
                dbms_output.put_line('..-> inner='||inner);
           end loop;
        end loop;
end;
```

SQL Server

```
declare @outer int = 1
while @outer <= 10
begin
   print('-> outer=' + cast(@outer as varchar(100)))
   InnerLoop:
   declare @inner int = 1
   while @inner <= 10
   begin
      if @inner > 5 GOTO OuterLoop
      print('..-> inner=' + cast(@inner as varchar(100)))
      set @inner = @inner + 1
   end
OuterLoop:
   set @outer = @outer + 1
end
```

Simulating Oracle Transactions in SQL Server 2014

During migration from Oracle to Microsoft SQL Server 2014, you must account for the differences in their default transaction management behavior. SSMA for Oracle V6.0 can convert Oracle's transaction-related statements, but you will find additional issues to consider, as described in this section.

If the SSMA **Convert transaction processing statements** option is turned on, SSMA tries to convert the Oracle statements for transaction management (COMMIT, ROLLBACK, and SAVEPOINT), but it does not add any statement for opening a transaction. So, you must decide which transaction management model to use in your application. Because SQL Server 2014 now allows optimistic escalation mode, choose between a pessimistic and an optimistic concurrency model.

Choosing a Transaction Management Model

In Oracle, a transaction automatically starts when an insert, update, or delete operation is performed. An application must issue a COMMIT command to save changes to the database. If a COMMIT is not performed, all changes are rolled back or undone automatically.

By default, SQL Server 2014 automatically performs a COMMIT statement after every insert, update, or delete operation. Because the data is automatically saved, you cannot roll back any changes.

You can start transactions in SQL Server 2014 as autocommit, implicit, or explicit transactions. Autocommit is the default behavior; you can use implicit or explicit transaction modes to change the default behavior.

Autocommit Transactions

Autocommit transactions are the default mode for SQL Server 2014. Each individual Transact-SQL statement is committed when it completes. You do not have to specify any statements to control transactions.

Implicit Transactions

As in Oracle, an implicit transaction starts whenever an INSERT, UPDATE, DELETE, or other data manipulating function is performed. To allow implicit transactions, use the SET IMPLICIT_TRANSACTIONS ON statement.

If this option is ON and there are no outstanding transactions, every SQL statement automatically starts a transaction. If there is an open transaction, no new transaction will start. The user must explicitly commit the open transaction with the COMMIT TRANSACTION statement for the changes to take effect and for all locks to be released.

Explicit Transactions

An explicit transaction is a grouping of SQL statements surrounded by BEGIN TRAN and COMMIT or ROLLBACK commands.

Therefore, for the complete emulation of the Oracle transaction behavior, use a SET IMPLICIT_TRANSACTIONS ON statement.

Choosing a Concurrency Model

Consider changing your application's isolation level. In a multiple-user environment, there are two models for updating data in a database:

- Pessimistic concurrency involves locking the data at the database when you read it. You exclusively lock the database record and don't allow anyone to touch it until you are done modifying and saving it back to the database. You have 100 percent assurance that nobody will modify the record while you have it checked out. Another person must wait until you have made your changes. Pessimistic concurrency complies with ANSI-standard isolation levels as defined in the SQL-99 standard. Microsoft SQL Server 2014 has four pessimistic isolation levels:
    - READ COMMITTED
    - READ UNCOMMITTED
    - REPEATABLE READ
    - SERIALIZABLE
- Optimistic concurrency means that you read the database record but don't lock it. Anyone can read and modify the record at any time, so the record might be modified by someone else before you modify and save it. If data is modified before you save it, a collision occurs. Optimistic concurrency is based on retaining a view of the data as it is at the start of a transaction. This model is embodied in Oracle. The transaction isolation level that implements an optimistic form of database concurrency is called a row versioning-based isolation level.

Because SQL Server 2014 has completely controllable isolation-level models, you can choose the most appropriate isolation level. To control a row-versioning isolation level, use the SET TRANSACTION ISOLATION LEVEL command. SNAPSHOT is the isolation level that is similar to Oracle and does optimistic escalations.

Make Transaction Behavior Look Like Oracle

For complete transaction management emulation in SQL Server 2014, and using a row-versioning isolation level, set the ALLOW_SNAPSHOT_ISOLATION option to ON for each database that is referenced in the Transact-SQL object (view, procedure, function, or trigger). In addition, either each Transact-SQL object must be started with a SNAPSHOT isolation level; otherwise, this level must be set on each client connection.

Alternatively, the autonomous block must be started with the READ COMMITTED isolation level with the READ_COMMITTED_SNAPSHOT database option set to ON.

Both the READ_COMMITTED_SNAPSHOT and ALLOW_SNAPSHOT_ISOLATION database options are set to ON in Azure SQL DB by default and cannot be changed.

Simulating Oracle Autonomous Transactions

This section describes how SSMA for Oracle V6.0 handles autonomous transactions (PRAGMA AUTONOMOUS_TRANSACTION). These autonomous transactions do not have direct equivalents in Microsoft SQL Server 2014.

When you define a PL/SQL block (anonymous block, procedure, function, packaged procedure, packaged function, database trigger) as an *autonomous transaction*, you isolate the DML in that block from the caller's transaction context. The block becomes an independent transaction started by another transaction, referred to as the *main transaction*.

To mark a PL/SQL block as an autonomous transaction, you simply include the following statement in your declaration section:

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

SQL Server 2014 does not support autonomous transactions. The only way to isolate a Transact-SQL block from a transaction context is to open a new connection.

To convert a procedure, function, or trigger with an AUTONOMOUS_TRANSACTION flag, you split it into two objects. The first object is a stored procedure containing the body of the converted object. It looks like it was converted without a PRAGMA AUTONOMOUS_TRANSACTION flag and is implemented as a stored procedure. The second object is a wrapper that opens a new connection where it invokes the first object. It is implemented via an original object type (procedure, function, or trigger).

Use the **xp_ora2ms_exec2** extended procedure and its extended version **xp_ora2ms_exec2_ex**, bundled with the SSMA 6.0 Extension Pack, to open new transactions. The procedure's purpose is to invoke any stored procedure in a new connection and help invoke a stored procedure within a function body. The **xp_ora2ms_exec2** procedure has the following syntax:

```
xp_ora2ms_exec2
      <active_spid> int,
      <login_time> datetime,
      <ms_db_name> varchar,
      <ms_schema_name> varchar,
      <ms_procedure_name> varchar,
      <bind_to_transaction_flag> varchar,
      [optional_parameters_for_procedure]
```

Where:

- <active_spid> [input parameter] is the session ID of the current user process.
- <login_time> [input parameter] is the login time of the current user process.

- <ms_db_name> [input parameter] is the database name owner of the stored procedure.
- <ms_schema_name> [input parameter] is the schema name owner of the stored procedure.
- <ms_procedure_name> [input parameter] is the name of the stored procedure.
- optional_parameters_for_procedure [input/output parameter] are the procedure parameters.

In general, you can retrive the active_spid parameter from the @@spid system function. You can query the login_time parameter with the statement:

- declare @login_time as datetime
- select @login_time=start_time from sys.dm_exec_requests where session_id=@@spid

We recommend that you use SSMA methods to retrieve the active_spid and login_time values before passing them to the **xp_ora2ms_exec2** procedure. Use the following recommended general template to invoke **xp_ora2ms_exec2**:

```
DECLARE @spid int, @login_time datetime
SELECT @spid = ssma_oracle.get_active_spid(),
@login_time = ssma_oracle.get_active_login_time()
EXEC master.dbo.xp_ora2ms_exec2_ex @spid, @login_time, <database_name>,
<schema_name>, <procedure_name>, [parameter1, parameter2, ... ]
```

Note that Azure SQL DB doen’t support extended stored procedures functionality and the solution proposed in this section is not applicable to this version of SQL Server.

Simulating Autonomous Procedures and Packaged Procedures

As mentioned earlier, SSMA ignores the PRAGMA AUTONOMOUS_TRANSACTION flag when it converts procedures. We recommend naming that procedure differently from the original, because it will not be invoked directly. You can implement the procedure wrapper body according to the following pattern:

```
CREATE PROCEDURE [schema.] <procedure_name>
<parameters list>
AS BEGIN
DECLARE @spid int, @login_time datetime
SELECT @spid = ssma_oracle.get_active_spid(),
@login_time = ssma_oracle.get_active_login_time()

EXEC master.dbo.xp_ora2ms_exec2 @ spid, @ login _spid, <database_name>,
<schema_name>, <procedure_name>$IMPL, [parameter1, parameter2, ... ]
```

```
END
```

- The <procedure_name>$IMPL parameter is the name of the procedure containing the converted source code.
- Note that the parameters list that is passed to the **xp_ora2ms_exec2** procedure should keep the IN/OUT options in the parameters for <procedure_name>$IMPL.
- Because the first PL-SQL statement in an autonomous routine begins a transaction, the procedure body should be begun with the set implicit_transactions on statement. The procedure body should be converted as the following pattern:

```
CREATE PROCEDURE [schema.] <procedure_name>$IMPL
<parameters list>
AS BEGIN
      set implicit_transactions on
<procedure_body>
END
```

Simulating Autonomous Functions and Packaged Functions

The method to simulate autonomous functions resembles that for procedures. Make the wrapper method for a function, and then implement the function body via a stored procedure. Add the additional parameter to the procedure's parameter list. Give the parameter a type corresponding to a function return value and an output direction.

Implement the function wrapper body according to the following pattern:

```
CREATE FUNCTION [schema.] <function_name>
(<parameters list>)
RETURNS <return_type>
AS BEGIN
DECLARE @spid int, @login_time datetime

SELECT @spid = ssma_oracle.get_active_spid(),
@login_time = ssma_oracle.get_active_login_time()

DECLARE @return_value_variable <function_return_type>

EXEC master.dbo.xp_ora2ms_exec2 @@spid,@login_time, <database_name>,
<schema_name>, <function_name>$IMLP,
[parameter1, parameter2, ... ,] @return_value_variable OUTPUT
```

```
RETURN @return_value_variable
END
```

The function body will be transformed into the following procedure:

```
CREATE PROCEDURE [schema.] <function_name>$IMPL
 <parameters list> ,
 @return_value_argument <function_return_type> OUTPUT
 AS BEGIN
      set implicit_transactions on
<function implementation>
SET @return_value_argument = <return_expression>
 END
```

The <return_expression> is an expression that a function uses in the RETURN operator.

Simulation of Autonomous Triggers

For conversion of autonomous triggers, see Autonomous Transactions in Triggers.

Code Example

The following code provides CREATE PROCEDURE examples that can be leveraged for Oracle and SQL Server 2014 respectively.

Oracle

```
CREATE OR REPLACE PROCEDURE update_salary (emp_id IN NUMBER)
IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
UPDATE employees SET site_id = site_id * 2 where employee_id=emp_id;
COMMIT;
EXCEPTION WHEN OTHERS THEN ROLLBACK;
END;
```

SQL Server 2014

```
CREATE PROCEDURE dbo.UPDATE_SALARY @emp_id float(53)
AS BEGIN
DECLARE @active_spid INT, @login_time DATETIME
SET @active_spid = ssma_oracle.GET_ACTIVE_SPID()
SET @login_time = ssma_oracle.GET_ACTIVE_LOGIN_TIME()
EXECUTE master.dbo.xp_ora2ms_exec2
@active_spid, @login_time,
'SYSTEM', 'DBO', 'UPDATE_SALARY$IMPL', @emp_id
END

CREATE PROCEDURE dbo.UPDATE_SALARY$IMPL @emp_id float(53)
AS BEGIN
SET IMPLICIT_TRANSACTIONS ON
BEGIN TRY
UPDATE dbo.EMPLOYEES SET SITE_ID = EMPLOYEES.SITE_ID * 2
WHERE EMPLOYEES.EMPLOYEE_ID = @emp_id

WHILE @@TRANCOUNT > 0 COMMIT WORK
END TRY
BEGIN CATCH
IF @@TRANCOUNT > 0
ROLLBACK WORK
END CATCH
END
```

Note that Azure SQL DB doen’t support extended stored procedures functionality and thus the above example is not applicable to this version of SQL Server.

Migrating Oracle Records and Collections

Unlike Oracle, Microsoft SQL Server 2014 supports neither records nor collections. When you migrate from Oracle to SQL Server 2014, therefore, you must apply substantial transformations to the PL/SQL code.

The approach used by SSMA for Oracle V6.0 is to convert both records and collections as a user-defined type implemented as SQL CLR type.

Note: SSMA for Oracle V6.0 does not convert collections. Therefore, this section describes manual migration activity.

Implementing Collections

To emulate collections, you have four options:

- Option 1. Rewrite your PL/SQL code to avoid collections.
- Option 2. When collections are used within the scope of a subroutine, you can use SQL Server table variables.
- Option 3. When you pass a collection as a parameter into a procedure or a function, a local temporary table can be the solution.
- Option 4. This option is a modification of Option 3. Instead of using temporary tables (which cannot be accessed from within function), you use permanent tables.
- Option 5. You can use the **xml** data type to represent the internal structure of a collection. For more information, see Implementing Records and Collections via XML.
- Option 6. Use SQL Server CLR user-defined type to create an analog of PL/SQL collection. As said before, this approach was chosen for implementation in SSMA for Oracle V6.0. For more information, see Emulating Records and Collections via CLR UDT.

Option 1. Rewrite your code to avoid records and collections. In many cases, collections or records are not justified. Generally, you can perform the same tasks by using set-oriented operators, meanwhile gaining performance benefits and code clearness.

In the PL/SQL code (from here and following we use the SCOTT demo scheme):

```
declare
  type emptable is table of integer;
  emps emptable;
  i integer;
begin
```

```
  select empno bulk collect into emps
  from Emp where deptno = 20;
  for i in emps.first..emps.last loop
   update scott.emp set sal=sal*1.2 where EmpNo=emps(i);
  end loop;
end;
```

The corresponding Transact-SQL code looks like:

```
update emp set sal=sal*1.2 where deptno = 20
```

Usually, nobody would write such awkward code in Oracle, but you may find something similar in, for example, proprietary systems. It might be a good opportunity to refactor the source code to use SQL where possible.

Option 2. In some situations you have no choice but to use collections (or something similar such as arrays).

Suppose you want to retrieve a list of employer IDs, and for each ID from the list execute a stored procedure to raise each salary.

If the PL/SQL source code looks like:

```
declare
  type emptable is table of integer;
  emps emptable;
  i integer;
begin
  select empno bulk collect into emps
  from Emp
  where deptno = 20;
  for i in emps.first..emps.last loop
   scott.raisesalary(Emp => emps(i),Amount => 10);
  end loop;
end;
```

The corresponding Transact-SQL code may look like:

```
declare @empno int
declare cur cursor local static forward_only for
select empno from emp where deptno = 20
open cur
fetch next from cur into @empno
while @@fetch_status = 0 begin
 exec raisesalary @emp=@empno,@amount=10
fetch next from cur into @empno
```

```
end
deallocate cur
```

Sometimes you need not only to run through a list and make an action for each record (as seen earlier), but you also want to randomly access elements in the list.

In this situation it is useful to use table variables. The general idea is to replace a collection (integer-indexed array) with a table (indexed by its primary key).

For the following PL/SQL code:

```
declare
  type emptable is table of integer;
  emps emptable;
  i integer;
  s1 numeric;
  s2 numeric;
begin
  select empno bulk collect into emps
  from Emp;
  for i in emps.first+1..emps.last-1 loop
   select sal into s1 from scott.emp where empno = emps(i-1);
   select sal into s2 from scott.emp where empno = emps(i+1);
   update emp set sal=(s1+s2)/2 where EmpNo=emps(i);
  end loop;
end;
```

The corresponding Transact-SQL code may look like:

```
declare @tab table(_idx_ int not null primary key, empno int)
insert into @tab(_idx_,empno) select row_number() over(order by
empno),empno from emp
declare @first int,@last int,@i int,@s1 money,@s2 money
select top 1 @first=_idx_ from @tab order by _idx_ asc
select top 1 @last =_idx_ from @tab order by _idx_ desc
set @i = @first+1
while @i < @last-1 begin
 select @s1 = sal from emp where empno = (select empno from @tab where
_idx_=@i-1)
 select @s2 = sal from emp where empno = (select empno from @tab where
_idx_=@i+1)
 update emp set sal = (@s1+@s2)/2 where empno = (select empno from @tab
where _idx_=@i)
 set @i = @i +1
end
```

In this example, the table variable *@tab*, indexed with an _idx_ field, represents our collection.

Pay attention to the row_number() function in the select statement. If you do not plan to insert explicit values in the collection, you can avoid using row_number:

```
declare @tab table(_idx_ int identity(1,1) not null primary key, empno
int)
insert into @tab(empno) select empno from emp
```

Now the @tab variable is sequentially indexed starting from 1.

If you are using a collection of %ROWTYPE, you can declare a table variable with an appropriate list of fields and use it as shown earlier.

By using table variables, you can emulate the functionality of almost any local collection, as shown in the following table.

Task	Collection	Emulation with table variable	Remarks
Declaration	`type emptable is table of integer;` `emps emptable;`	`declare @emp table(_idx_ int not null primary key, empno int)` or `declare @emps table(_idx_ int identity(1,1) not null primary key, empno int)`	First declaration for “manual” indexing and second for “automatic” (by identity) indexing.
Set value into collection	`emp(i) := 12;`	`update @emp set empno = 12 where _idx_=@i` `if @@rowcount = 0` `insert into @emps(_idx_,empno)` `values(@i,12)`	You are trying to update the record with _idx_=@i. If it doesn’t exist (@@rowcount=0) , simply insert the needed data. Note: If you use an identity field as _idx_, you cannot insert an explicit value into the _idx_ field.
Get value from collection	`Empno = emp(i);`	`select @empno = empno from @emps where _idx_ = @i`	
FIRST method	`I_first := emp.FIRST;`	`select @i_first = min(_idx_) from @emps`	Comment on set @i_last=null

Task	Collection	Emulation with table variable	Remarks

		`or`
		`set @i_last=null`
		`select top 1 @i_first = _idx_ from @emps order by _idx_ asc`
			If the select statement does not return any row, @i_first will not change its value, keeping the previously stored value. So, first initialize this variable as null.

LAST method	`I_last := emp.LAST;`	`select @i_last = max(_idx_) from @emps`
		`or`
		`set @i_last=null`
		`select top 1 @i_last = _idx_ from @emps order by _idx_ desc`	

NEXT method	`I_next := emp.NEXT(j);`	`select @i_last = min(_idx_) from @emps where _idx_ > @i`	
PRIOR method	`I_prior := emp.PRIOR(j);`	`select @i_last = max(_idx_) from @emps where _idx_ < @i`	

DELETE method	`emps.delete(i);`
	`emps.delete;`
		`DELETE FROM @emps WHERE _idx_ = @i`
		`DELETE FROM @emps`	

TRIM method	`emps.trim;`
	`emps.trim(n);`
		`declare @_idx_ int`
		`select top(@n) @_idx_= _idx_ from @emps order by _idx_ desc`
		`delete @emps where _idx_ >= @_idx_`
			emps.trim is equivalent to emps.trim(1).

EXISTS method	`t.exists(i)`	`exists(select * from @emps where _idx_ = @i)`	
COUNT method	`i = t.COUNT;`	`select @t_count = COUNT(*) FROM @emps`	

Bulk collect into	`select empno bulk collect into emps`	`INSERT INTO @emps (_idx_, empno)`
		` SELECT row_number()`
			The row_number() function depends

Task	Collection	Emulation with table variable	Remarks

	from emp	over(order by empno) as _idx_, empno
		from emp
		or
		INSERT INTO @emps (empno)
		SELECT empno from emp
			on @emps table declaration. For declaration with identity _idx_ column do not use row_number().

EXTEND method	t.extend;
	t.extend(n);
	t.extend(n, i);
		SELECT @t_next_value = ISNULL(MAX(_idx_),0)+1 FROM @emps
		
		  INSERT INTO @emps (_idx_, empno)
		   VALUE(@t_next_value, NULL)
		---------------------------- 	
		
		  SELECT @t_cur_value = ISNULL(MAX(_idx_),0) FROM @emps
		
		  WHILE @n <> 0
		  BEGIN
		    @t_cur_value = @t_cur_value + 1
		    INSERT INTO @emps (_idx_, empno)
		     VALUE(@t_cur_value, NULL)
		    SET @n = @n-1
		  END
		---------------------------- 	
		
		  SELECT @t_cur_value = ISNULL(MAX(_idx_),0) FROM @emps
		  SELECT @v = empno FROM @emps where _idx_ = @i
		
		  WHILE @n <> 0
		  BEGIN
		    @t_cur_value =	

Task	Collection	Emulation with table variable	Remarks

		`@t_cur_value + 1`
		`    INSERT INTO @emps (_idx_, empno)`
		`     VALUE(@t_cur_value, @v)`
		`    SET @n = @n-1`
		`  END`	

FORALL … INSERT INTO	`FORALL i IN 1..20 INSERT INTO emp(empno) VALUES (t(i))`	`INSERT INTO emp (empno)`
		`   SELECT empno FROM @emps WHERE _idx_ between 1 and 20`	

FORALL … UPDATE	`FORALL i IN 6..10 UPDATE emp SET sal = sal * 1.10 WHERE empno = t(i);`	`UPDATE emp SET sal = sal * 1.10`
		`FROM (SELECT * FROM @emps WHERE _idx_ between 6 and 10) as t_a`
		` INNER JOIN emp`
		`    ON (emp.empno = t_a.empno)`	

FORALL … DELETE	`FORALL i IN 6..10`
	`DELETE FROM emp`
	`WHERE empno = t(i);`
		`DELETE FROM emp WHERE empno IN (SELECT empno FROM @t WHERE _idx_ between 6 and 10)`	

Option 3. Another collection scenario is when you pass a collection as a parameter into a procedure or a function.

The solution is similar to the solution that uses table variables. The main difference is that instead of a table variable you use a local temporary table (#tab, for example). The table will be visible in the procedure that created this table and in all subsequent procedures.

PL/SQL code

Stored procedure:

```
create procedure emp_raise(emps in emptable)
i int;
is begin
 for i in emps.first..emps.last loop
  raisesalary(Emp => emps(i),Amount => 10);
 end loop;
end;
```

Procedure call:

```
declare
type emptable is table of integer;
emps emptable;
begin
 select empno
 bulk collect into emps
 from scott.emp;
 emp_raise(emps);
end;
```

Transact-SQL code

Stored procedure:

```
create procedure emp_raise
as begin
 declare @empno int
 declare cur cursor local static forward_only for
 select empno from #emp
 open cur
 fetch next from cur into @empno
 while @@fetch_status = 0 begin
```

```
  exec raisesalary @emp=@empno,@amount=10
 fetch next from cur into @empno
 end
 deallocate cur
end
```

Procedure call:

```
create table #emp(_idx_ int not null identity,empno int)
insert into #emp(empno) select empno from emp
exec emp_raise
drop table #emp
```

Instead of using a collection, you pass needed data to a stored procedure via a temporary table. Of course you miss useful things such as parameter substitution. (The name of the temporary table you create outside of the stored procedure must be the same name as the temporary table in the stored procedure.) That is, you do not cover situations in which different actual collections are passed to the procedure. But, unfortunately, you cannot access a temporary table from within SQL Server functions.

Option 4. This option is a slight modification of Option 3. Instead of using temporary tables (which cannot be accessed from within function), you use permanent tables.

Unlike temporary tables, you can access permanent tables and views from within functions. But be aware that you cannot use DML statements in functions, so this collection emulation is read-only. If you want to modify a collection from within a user-defined function, you must use another kind of emulation; you cannot modify permanent tables from within user-defined functions. (For more information, see Sample Functions for XML Record Emulation.)

The only difference between Option 4 and Option 3 is that the table should be cleaned before use.

PL/SQL code

```
declare
  type emptable is table of integer;
  emps emptable;
  i integer;
  s1 numeric;
  s2 numeric;
begin
  select empno bulk collect into emps
  from Emp;
  for i in emps.first+1..emps.last-1 loop
   select sal into s1 from scott.emp where empno = emps(i-1);
   select sal into s2 from scott.emp where empno = emps(i+1);
   update emp set sal=(s1+s2)/2 where EmpNo=emps(i);
  end loop;
end;
```

Transact-SQL code

Create a table for collection emulation:

```
create table emps_t(SPID smallint not null default @@SPID,_idx_ int not
null,empno int null)
go
create clustered index cl on emps_t(SPID,_idx_)
go
create view emps
as select _idx_,empno from emps_t where spid = @@spid
go
```

The converted code:

```
delete emps

insert into emps(_idx_,empno) select row_number() over(order by empno),empno
from emp
declare @first int,@last int,@i int,@s1 money,@s2 money
select top 1 @first=_idx_ from emps order by _idx_ asc
select top 1 @last =_idx_ from emps order by _idx_ desc
set @i = @first+1
while @i < @last-1 begin
 select @s1 = sal from emp where empno = (select empno from emps where
_idx_=@i-1)
 select @s2 = sal from emp where empno = (select empno from emps where
_idx_=@i+1)
 update emp set sal = (@s1+@s2)/2 where empno = (select empno from emps where
_idx_=@i)
 set @i = @i +1
end
```

Be aware that, unlike table variables, permanent tables are transaction-dependent, which may lead to unwanted lock contention. Pay attention when using this option; you cannot avoid using a row_number() function.

Implementing Records

Usually you use records to simplify your PL/SQL code.

For example, instead of writing:

```
declare
 empno number(4);
 ename varchar(10);
 job   varchar(9);
 mgr   number(4);
 hiredate date;
 sal number(7,2);
 comm number(7,2);
 deptno number(2);
begin
 select * into empno,ename,job,mgr,hiredate,sal,comm,deptno from scott.emp
where empno = 7369;
 dbms_output.put_line(ename);
end;
```

You could write simple and clear code:

```
declare
   emps scott.emp%rowtype;
begin
 select * into emps from scott.emp where empno = 7369;
 dbms_output.put_line(emps.ename);
end;
```

Unfortunately, SQL Server doesn't support records. The default SSMA for Oracle V6.0 approach is to split the record into a group of the constituting variables.

To do that, declare a separate variable for each column as in the following code:

```
declare @empno int,@ename varchar(10),@job varchar(9),@mgr int,@hiredate
datetime,@sal numeric(7,2),@comm numeric(7,2),@deptno int


select @empno=empno, @ename=ename, @job=job, @mgr=mgr, @hiredate=hiredate,
@sal=sal, @comm=comm, @deptno=deptno
from emp where empno = 7369


print @ename
```

The situation is the same situation passing records into procedures or functions; you should pass variables one by one into a procedure.

PL/SQL code

```
declare
   emps scott.emp%rowtype;
begin
 select * into emps from scott.emp where empno = 7369;
 raise_emp_salary(emps);
end;
```

Transact-SQL code

```
declare @empno int,@ename varchar(10),@job varchar(9),@mgr int,@hiredate
datetime,@sal numeric(7,2),@comm numeric(7,2),@deptno int
select @empno=empno, @ename=ename, @job=job, @mgr=mgr, @hiredate=hiredate,
@sal=sal, @comm=comm, @deptno=deptno
from emp where empno = 7369

exec raise_emp_salary @empno,@ename,@job,@mgr,@hiredate,@sal,@comm,@deptno
```

Implementing Records and Collections via XML

The most universal but most complex way to emulate collections or records is emulation via XML. With XML implementation, you can store records and collections in a database (for example, in an XML field in a table), and pass records and collections into stored procedures and user-defined functions. However, take into account that manipulation with XML (especially modifying) is relatively slow.

Implementing Records

For complex cases you can emulate records via XML. For example, you could emulate scott.emp%rowtype with the following XML structure:

```
<row>
  <f_name>DEPTNO</f_name>
  <_val>20</_val>
</row>
<row>
  <f_name>SAL</f_name>
  <_val>800</_val>
</row>
<row>
  <f_name>HIREDATE</f_name>
  <_val>Dec 17 1980 12:00:00:000AM</_val>
</row>
<row>
  <f_name>MGR</f_name>
  <_val>7902</_val>
</row>
<row>
  <f_name>JOB</f_name>
  <_val>CLERK</_val>
</row>
<row>
  <f_name>ENAME</f_name>
  <_val>SMITH</_val>
</row>
<row>
  <f_name>EMPNO</f_name>
  <_val>7369</_val>
</row>
```

To work with such a structure you need additional supplemental procedures and functions to simplify access to the data. (Examples of the modules provided by SSMA are at the end of this section.)

Now you can rewrite your sample:

```
DECLARE
       CURSOR emp_cursor IS
       SELECT empno, ename FROM scott.emp;
       emps emp_cursor%rowtype;
BEGIN
  open emp_cursor;
  loop
    fetch emp_cursor into emps;
    exit when emp_cursor%notfound;
    raise_emp_salary(emp_rec);
  end loop;
  close emp_cursor;
END;
```

As the following Transact-SQL code:

```
DECLARE @emps xml,@emps$empno int,@emps$ename varchar(max)
DECLARE emp_cursor CURSOR LOCAL FOR
SELECT EMP.EMPNO, EMP.ENAME
FROM dbo.EMP
OPEN emp_cursor
FETCH next from emp_cursor INTO @emps$empno, @emps$ename
WHILE @@fetch_status = 0 begin
         SET @emps = ssma_oracle.SetRecord_varchar(@emps, N'ENAME',
@emps$ename)
         SET @emps = ssma_oracle.SetRecord_float(@emps, N'EMPNO',
@emps$empno)
         EXECUTE raise_emp_salary @emps
FETCH next from emp_cursor INTO @emps$empno, @emps$ename
END
CLOSE emp_cursor
DEALLOCATE emp_cursor
```

The code here is slightly different from SSMA-generated code. It shows only basic techniques for working with XML records. (You fetch data from a cursor into separate variables, and then you construct from it and an XML record.)

To extract data back from XML you could use an appropriate function such as:

```
set @ename = ssma_oracle.GetRecord_varchar(@emps, N'ENAME')
```

Implementing Collections

PL/SQL code

```
DECLARE
  TYPE Colors IS TABLE OF VARCHAR2(16);
  rainbow Colors;
BEGIN
  rainbow := Colors('Red', 'Yellow');
END;
```

Transact-SQL code, collection

```
DECLARE @rainbow XML
SET @rainbow = '<coll_row _idx_="1">
```

```
                  <row> <_val>Red</_val> </row>
                </coll_row>
                <coll_row _idx_="2">
                  <row> <_val>Yellow</_val> </row>
                </coll_row>'
```

Transact-SQL code, collection of records

```
DECLARE @x XML
SET @x =
'<coll_row _idx_="1">
<row>
   <f_name>record_field_1</f_name>
   <_val>value_1</_val>
 </row>
</coll_row>
<coll_row _idx_="2">
 <row>
   <f_name>record_field_2</f_name>
   <_val>value_2</_val>
 </row>
</coll_row>
'
```

After these declarations you can modify a collection, record, or collection of records by using XQuery. You may find it useful to write wrapper functions to work with XML, such as GET and SET functions.

Sample Functions for XML Record Emulation

Transact-SQL GET wrapper function for the varchar data type

```
CREATE FUNCTION GetRecord_Varchar
 (@x XML, @column_name varchar(128)) RETURNS varchar(MAX)
BEGIN
  DECLARE @v_x_value varchar(MAX)
  SELECT TOP 1 @v_x_value = T.c.value('(_val)[1]', 'varchar(MAX)')
    FROM @x.nodes('/row') T(c) WHERE T.c.value('(f_name)[1]', 'varchar(128)')
= @column_name
  return(@v_x_value)
```

```
END
```

Transact-SQL SET wrapper function for the varchar data type

```
CREATE FUNCTION SetRecord_Varchar (
    @x XML, @column_name varchar(128), @v varchar(max))
    RETURNS XML
    AS
  BEGIN
    IF @x IS NULL SET @x = ''
    IF @x.exist('(/row/f_name[.=sql:variable("@column_name")])[1]') = 1
    BEGIN
      if @v is not null
      BEGIN
        SET @x.modify( 'delete
                         (/row[f_name=sql:variable("@column_name")])[1]
                       ')
        SET @x.modify( 'insert (<row>
<f_name>{sql:variable("@column_name")}</f_name>
                                      <_val>{sql:variable("@v")}</_val>
</row>)
                          into (/)[1] ' )
      END
      else
        SET @x.modify( 'delete
                         (/row[f_name=sql:variable("@column_name")]
                          /_val[1])[1]
                       ')
    END
    ELSE
    if @v is not null
      SET @x.modify( 'insert (<row>
<f_name>{sql:variable("@column_name")}</f_name>
                                       <_val>{sql:variable("@v")}</_val>
</row>)
                        into (/)[1] ' )
    RETURN(@x)

  END;
```

A sample call

```
DECLARE
  @x xml

SET @x = dbo.SetRecord_varchar(@x, N'RECORD_FIELD_1', 'value_1')
SET @x = dbo.SetRecord_varchar(@x, N'RECORD_FIELD_2', 'value_2')
PRINT dbo.GetRecord_varchar(@x, N'RECORD_FIELD_2')
```

For more information, see XQuery Functions against the xml Data Type (http://msdn.microsoft.com/en-us/library/ms189254.aspx) in SQL Server Books Online.

Emulating Records and Collections via CLR UDT

The emulation method chosen in SSMA for Oracle V6.0 uses SQL CLR user-defined types (UDT). This method is more efficient than the emulation by XML, and generally it does not lead to code bloat, which can happen with solutions based on table variables or on temporary tables. Nevertheless, this solution is not based on SQL Server native mechanisms, and in some cases, you can find the emulation by tables quicker and more convenient. Note also that this solution includes creation of assemblies in the target database, which could create problems during deployment and during maintenance of the system after the migration.

Declaring Record or Collection Types

SSMA creates three CLR-based UDTs:

- CollectionIndexInt
- CollectionIndexString
- Record

The CollectionIndexInt type is intended for simulating collections indexed by integer, such as VARRAYs, nested tables and integer key based associative arrays. The CollectionIndexString type is used for associative arrays based indexed by character keys. Oracle record functionality is emulated by the Record type.

All declarations of record or collection types are converted to this Transact-SQL declaration:

```
declare @Collection$TYPE varchar(max) = '<type definition>'
```

Here <type definition> is a descriptive text uniquely identifying the source PL/SQL type. For example:

Oracle

```
TYPE animal IS RECORD (id integer, name varchar2(40), canFly integer);
TYPE animals is TABLE OF animal INDEX BY PLS_INTEGER;
```

SQL Server

```
DECLARE
@Record$TYPE varchar(max) = 'RECORD ( ID INT , NAME STRING , CANFLY INT )',
@CollectionIndexInt$TYPE varchar(max) = 'TABLE INDEX BY INT OF (' +
@Record$TYPE + ')'
```

Declaring Record or Collection Variables

Each of the types CollectionIndexInt, CollectionIndexString, and Record has a static property [Null] returning an empty instance. Method SetType is called to receive an empty object of a specific type. For example, the conversion of a TABLE OF declaration will look like this.

Oracle

```
declare
TYPE <type_name> TABLE OF <element_type> INDEX BY [PLS_INTEGER |
BINARY_INTEGER];
<var_name> <type_name>;
```

SQL Server

```
DECLARE
@CollectionIndexInt$TYPE varchar(max) = 'TABLE INDEX BY INT OF
<element_type>'
DECLARE
@<var_name> dbo.CollectionIndexInt = = dbo.CollectionIndexInt
::[Null].SetType(@CollectionIndexInt$TYPE)
```

Converting Constructor Calls

Constructor notation can be used only for nested tables and VARRAYs, so all the explicit constructor calls are converted using the CollectionIndexInt type. Empty constructor calls are converted via SetType call invoked on null instance of CollectionIndexInt. The [Null] property returns the null instance. If the constructor contains a list of elements, special method calls are applied sequentially to add the value to the collection.

Oracle

```
DECLARE
   TYPE nested_type IS TABLE OF VARCHAR2(20);
   TYPE varray_type IS VARRAY(5) OF INTEGER;
   v1 nested_type;
   v2 varray_type;
BEGIN
   v1 := nested_type('Arbitrary','number','of','strings');
   v2 := varray_type(10, 20, 40, 80, 160);
END;
```

SQL Server

```
DECLARE
      @CollectionIndexInt$TYPE varchar(max) = ' TABLE OF STRING',
      @CollectionIndexInt$TYPE$2 varchar(max) = ' VARRAY OF INT',
      @v1 dbo.CollectionIndexInt,
      @v2 dbo.CollectionIndexInt

   SET @v1 = dbo.CollectionIndexInt
::[Null].SetType(@CollectionIndexInt$TYPE).AddString('Arbitrary').AddString('
number').AddString('of').AddString('strings')

   SET @v2 = dbo.CollectionIndexInt
::[Null].SetType(@CollectionIndexInt$TYPE$2).AddInt(10).AddInt(20).AddInt(40)
.AddInt(80).AddInt(160)
```

Referencing and Assigning Record and Collection Elements

Each of the UDTs has a set of methods working with elements of various data types. For example, the SetDouble method assigns a **float**(53) value to record or collection, and GetDouble can read this value. The complete list of methods is here:

```
GetCollectionIndexInt(@key <KeyType>) returns CollectionIndexInt;
SetCollectionIndexInt(@key <KeyType>, @value CollectionIndexInt) returns
<UDT_type>;
GetCollectionIndexString(@key <KeyType>) returns CollectionIndexString;
SetCollectionIndexString(@key <KeyType>, @value CollectionIndexString)
returns <UDT_type>;
Record GetRecord(@key <KeyType>) returns Record;
```

```
SetRecord(@key <KeyType>, @value Record) returns <UDT_type>;
GetString(@key <KeyType>) returns nvarchar(max);
SetString(@key <KeyType>, @value nvarchar(max)) returns nvarchar(max);
GetDouble(@key <KeyType>) returns float(53);
SetDouble(@key <KeyType>, @value float(53)) returns <UDT_type>;
GetDatetime(@key <KeyType>) returns datetime;
SetDatetime(@key <KeyType>, @value datetime) returns <UDT_type>;
GetVarbinary(@key <KeyType>) returns varbinary(max);
SetVarbinary(@key <KeyType>, @value varbinary(max)) returns <UDT_type>;
SqlDecimal GetDecimal(@key <KeyType>);
SetDecimal(@key <KeyType>, @value numeric) returns <UDT_type>;
GetXml(@key <KeyType>) returns xml;
SetXml(@key <KeyType>, @value xml) returns <UDT_type>;
GetInt(@key <KeyType>) returns bigint;
SetInt(@key <KeyType>, @value bigint) returns <UDT_type>;
```

These methods are used when referencing or assigning a value to an element of a collection/record.

Oracle

```
a_collection(i) := 'VALUE';
```

SQL Server

```
SET @a_collection = @a_collection.SetString(@i, 'VALUE');
```

When converting assignment statements for multidimensional collections or collections with record elements, SSMA adds the following methods to refer to a parent element inside the set method:

```
GetOrCreateCollectionIndexInt(@key <KeyType>) returns CollectionIndexInt;

GetOrCreateCollectionIndexString(@key <KeyType>) returns
CollectionIndexString;

GetOrCreateRecord(@key <KeyType>) returns Record;
```

For example, a collection of record elements is created this way:

Oracle

```
declare
TYPE rec_details IS RECORD (id int,name varchar2(20));
type ntb1 is table of rec_details index by binary_integer;
c ntb1;
begin
c(1).id := 1;
end;
```

SQL Server

```
DECLARE
   @CollectionIndexInt$TYPE varchar(max) = ' TABLE INDEX BY INT OF ( RECORD (
ID INT , NAME STRING ) )',
   @c dbo.CollectionIndexInt = dbo.CollectionIndexInt
::[Null].SetType(@CollectionIndexInt$TYPE)
SET @c = @c.SetRecord(1, @c.GetOrCreateRecord(1).SetInt(N'ID', 1))
```

Collection Built-in Methods

SSMA uses the following UDT methods to emulate built-in methods of PL/SQL collections.

Oracle collection methods	CollectionIndexInt and CollectionIndexString equivalent
COUNT	Count returns int
DELETE	RemoveAll() returns <UDT_type>
DELETE(n)	Remove(@index int) returns <UDT_type>
DELETE(m,n)	RemoveRange(@indexFrom int, @indexTo int) returns <UDT_type>
EXISTS	ContainsElement(@index int) returns bit
EXTEND	Extend() returns <UDT_type>
EXTEND(n)	Extend() returns <UDT_type>
EXTEND(n,i)	ExtendDefault(@count int, @def int) returns <UDT_type>
FIRST	First() returns int
LAST	Last() returns int
LIMIT	N/A
PRIOR	Prior(@current int) returns int
NEXT	Next(@current int) returns int
TRIM	Trim() returns <UDT_type>
TRIM(n)	TrimN(@count int) returns <UDT_type>

BULK COLLECT operation

SSMA converts BULK COLLECT INTO statements into SQL Server SELECT … FOR XML PATH statement, whose result is wrapped into one of the following functions:

```
ssma_oracle.fn_bulk_collect2CollectionSimple
ssma_oracle.fn_bulk_collect2CollectionComplex
```

The choice depends on the type of the target object. These functions return XML values that can be parsed by CollectionIndexInt, CollectionIndexString and Record types. A special AssignData function assigns XML-based collection to the UDT.

SSMA recognizes three kinds of BULK COLLECT INTO statements:

1. The collection contains elements with scalar types, and the SELECT list contains one column:

   Oracle

   ```
   SELECT column_name_1
     BULK COLLECT INTO <collection_name_1>  FROM <data_source>
   ```

   SQL Server

   ```
   SET @<collection_name_1>  =
   @<collection_name_1>.AssignData(ssma_oracle.fn_bulk_
   collect2CollectionSimple((select column_name_1 from <data_source> for
   xml path)))
   ```

2. The collection contains elements with record types, and the SELECT list contains one column:

   Oracle

   ```
   SELECT column_name_1[, column_name_2...]
     BULK COLLECT INTO <collection_name_1>  FROM <data_source>
   ```

   SQL Server

   ```
   SET @<collection_name_1>  =
   @<collection_name_1>.AssignData(ssma_oracle.fn_bulk_
   collect2CollectionComplex((select column_name_1 as
   [collection_name_1_element_field_name_1], column_name_2 as
   ```

```
[collection_name_1_element_field_name_2] from <data_source> for xml
path)))
```

3. The collection contains elements with scalar type, and the SELECT list contains multiple columns:

   Oracle

```
SELECT column_name_1[, column_name_2 ...]
  BULK COLLECT INTO <collection_name_1>[, <collection_name_2> ...]
  FROM <data_source>
```

   SQL Server:

```
;with bulkC as (select column_name_1
[collection_name_1_element_field_name_1], column_name_2
[collection_name_1_element_field_name_2] from <data_source>)
select @<collection_name_1> =
@<collection_name_1>.AssignData(ssma_oracle.fn_bulk_
collect2CollectionSimple((select
[collection_name_1_element_field_name_1] from bulkC for xml path))),
@<collection_name_2> =
@<collection_name_2>.AssignData(ssma_oracle.fn_bulk_
collect2CollectionSimple ((select
[collection_name_1_element_field_name_2] from bulkC for xml path)))
```

SELECT INTO Record

When the result of Oracle query is saved in a PL/SQL record variable, you have two options, depending on the SSMA setting for **Convert record as a list of separated variables**. If the value of this setting is **Yes** (the default), SSMA does not create an instance of Record type. Instead, it splits the record into the constituting fields by creating a separate Transact-SQL variable per each record field. If the setting is **No**, the record is instantiated and each field is assigned a value using **Set** methods.

SSMA Records and Collections Migration to Azure SQL DB

Azure SQL DB doesn’t support CLR. So SSMA doesn’t use CLR emulations for collections and records in this case. Instead, it uses XML emulation for Azure SQL DB.

There are a set of predefined functions created in ssma_oracle schema of the converted database by SSMA. Their names look like `SetCollection_<datatype>` and

`GetCollection_<datatype>` and they emulate working with collections and records using XML.

Below is an example of SSMA conversion of collections and records to Azure SQL DB:

Oracle

```
declare
   TYPE intTab IS TABLE OF int index by int;
   v_tab intTab ;

   TYPE intTabStr IS TABLE OF int index by varchar2(100);
   v_tab2 intTabStr ;

   TYPE timeRec IS Record (hh int, mm int, ss int);
   v_rec timeRec ;

   v int;
BEGIN
   v_tab(17) := 1;
   v_tab(25) := 2;
   v := v_tab(17);

   v_tab2('17') := 1;
   v_tab2('25') := 2;
   v := v_tab2('17');

   v_rec.hh := 18;
   v_rec.mm := 10;
   v_rec.ss := 20;
   v := v_rec.mm;
END;
```

Azure SQL DB

```
BEGIN

   DECLARE
      @v_tab xml,
      @v_tab2 xml

   DECLARE
      @v_rec$hh int,
      @v_rec$mm int,
      @v_rec$ss int,
      @v int

   SET @v_tab = ssma_oracle.SetCollection_int(@v_tab, 17, 1)

   SET @v_tab = ssma_oracle.SetCollection_int(@v_tab, 25, 2)

   SET @v = ssma_oracle.GetCollection_int(@v_tab, 17)

   SET @v_tab2 = ssma_oracle.SetCollection_int_varchar(@v_tab2, '17', 1)
   SET @v_tab2 = ssma_oracle.SetCollection_int_varchar(@v_tab2, '25', 2)

   SET @v = ssma_oracle.GetCollection_int_varchar(@v_tab2, '17')

   SET @v_rec$hh = 18

   SET @v_rec$mm = 10

   SET @v_rec$ss = 20

   SET @v = @v_rec$mm

END
GO
```

Migrating Tables to Memory-Optimized Tables

SQL Server 2014 introduced In-Memory OLTP database concept which improves OLTP database performance. The In-Memory OLTP feature includes memory-optimized tables, table types and native compilation of stored procedures for efficient access to these tables.

Memory-optimized tables is an advanced technology of table storage that provides high speed of data access due to holding data in memory. Memory-optimized tables is are based on special OLTP engine (together with In-Memory Precompiled procedures).

The increased speed of memory-optimized tables processing allows to reproduce processing of Oracle tables build on hash cluster index.

AS memory-optimized tables reside in memory, rows in the table are read from and written to memory. A second copy of the table data is maintained on disk, but only for durability purposes. Each row in the table potentially has multiple versions. This row versioning is used to allow concurrent reads and writes on the same row.

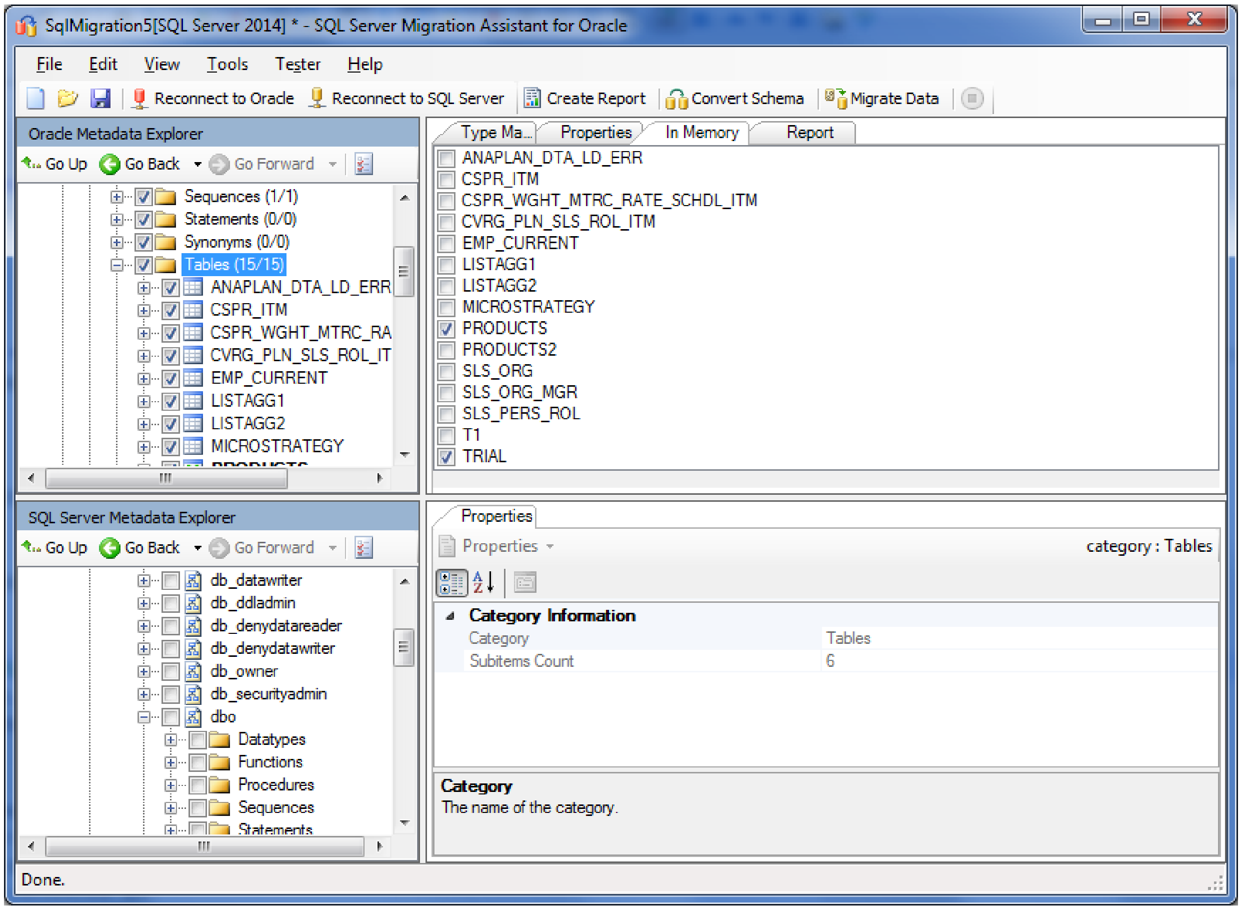
SSMA allows migrating Oracle tables to memory-optimized tables in SQL Server. For Tables node in Oracle Metadata Explorer there is **In Memory** tab on the right pane of SSMA window. It allows checking the tables you want to migrate to memory-optimized ones (see **Figure 5**).

Another way to check a table for conversion to memory-optimized tables is clicking on the table name under Tables node in Oracle Metadata Explorer and check **Convert to memory optimized table** check box on **In Memory** tab on the right pane of SSMA window.

DDL syntax for creating memory-optimized table is as follows:

```
CREATE TABLE database_name.schema_name.table_name
(
    column_name data_type
        [COLLATE collation_name] [NOT] NULL
        [DEFAULT constant_expression]
        [IDENTITY]
        [PRIMARY KEY NONCLUSTERED [HASH WITH (BUCKET_COUNT = bucket_count)]]
        [INDEX index_name
                    [NONCLUSTERED [HASH WITH (BUCKET_COUNT = bucket_count)]]]
    [,…]
    [PRIMARY KEY
    {
        NONCLUSTERED HASH (column [,…]) WITH (BUCKET_COUNT = bucket_count) |
        NONCLUSTERED      (column [ASC|DESC] [,…] ) }
    }]
    [INDEX index_name
    {
        NONCLUSTERED HASH (column [,…]) WITH (BUCKET_COUNT = bucket_count) |
        NONCLUSTERED      (column [ASC|DESC] [,…] ) }
    }] [,…]
```

```
)
WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
```



☐ ANAPLAN_DTA_LD_ERR
☐ CSPR_ITM
☐ CSPR_WGHT_MTRC_RATE_SCHDL_ITM
☐ CVRG_PLN_SLS_ROL_ITM
☐ EMP_CURRENT
☐ LISTAGG1
☐ LISTAGG2
☐ MICROSTRATEGY
☑ PRODUCTS
☐ PRODUCTS2
☐ SLS_ORG
☐ SLS_ORG_MGR
☐ SLS_PERS_ROL
☐ T1
☑ TRIAL

Category Information	
Category	Tables
Subitems Count	6

Category
The name of the category.

Figure 5: The In Memory tab for Tables node

Below is an example of Oracle table converted to memory-optimized one by SSMA:

Oracle

```
CREATE TABLE PROD.PRODUCTS
(
      PROD_ID NUMBER(38, 0) NOT NULL,
      PROD_NAME VARCHAR2(50)
);
ALTER TABLE PROD.PRODUCTS ADD CONSTRAINT PK_PROD
      PRIMARY KEY (PROD_ID);
```

SQL Server

```
CREATE TABLE [dbo].[PRODUCTS]
(
   [PROD_ID] numeric(38, 0)  NOT NULL,

   /*
   *   SSMA warning messages:
   *   O2SS0499: Column type VARCHAR(50): [VARCHAR] is changed to
NVARCHAR(50): [NVARCHAR] because of Memory-optimized table columns of types
CHAR or VARCHAR support 1252 codepage only
   */

   [PROD_NAME] nvarchar(50)  NULL,

   PRIMARY KEY NONCLUSTERED
   (
      [PROD_ID] ASC
   )

) WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA)
GO
```

Restrinctions for conversion to memory-optimized tables

There are some restrictions for creating memory-optimized tables that shouls be taken into account.

When converting an Oracle table that uses sequence to SQL Server table using IDENTITY property, it must be created only with SEED equal to 1 and INCREMENT equal to 1. If this condition is not met, SSMA generates a warning message likr this: “Cannot create identity with seed 1000 and increment 5 for Memory optimized table. Allowed only Identity(1,1)”. There are two ways to solve this issue.

First one is to convert the table with IDENTITY (1, 1) and add a corresponding seed to the identity column value and multiplying this value into the corresponding increment. For example, if Oracle sequence has seed value equal to 10 and increment value equal to 2:

SQL Server

```
CREATE TABLE imt(
id INT NOT NULL IDENTITY(1,1) PRIMARY KEY NONCLUSTERED,
name VARCHAR(50) NOT NULL)
WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA)

SELECT 10 + (id - 1) * 2
FROM imt;
```

The second way is to use SQL Server SEQUENCE objects instead of IDENTITY property when inserting new records:

SQL Server

```
CREATE TABLE imt(
id INT NOT NULL PRIMARY KEY NONCLUSTERED,
name VARCHAR(50) NOT NULL)
WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
GO

CREATE SEQUENCE imt_seq AS INT START WITH 10 INCREMENT BY 2
GO

INSERT INTO imt(id, name)
SELECT NEXT VALUE FOR imt_seq, 'New Name';
```

The next restriction is that uniqueidentifier column default is not supported for memory-optimized tables. Besides, column defaults support only constant expressions. SSMA issues warning about that and removes the column default. A workarownd for this can be defining the column that uses uniqueidentifier default as varchar column that can contain at least 36 characters (this is the length of uniqueidentifier value in SQL Server). Insert the value to this column explicitly every time when inserts to the table are performed:

Oracle

```
CREATE TABLE IMT
(
      ID RAW(32) DEFAULT sys_guid(),
      NAME VARCHAR2(50)
);
```

SQL Server

```
CREATE TABLE [dbo].[IMT]
(
   [ID] varchar(36)  NULL,
   [NAME] nvarchar(50)  NULL,
   [PKCol] int IDENTITY(1, 1)  NOT NULL,
   PRIMARY KEY NONCLUSTERED
   (
      [PKCol] ASC
   )

) WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA)
GO
```

Some data types are not supported by memory-optimized tables.

When Oracle table contains a column of TIMESTAMP WITH TIME ZONE data type, SSMA converts it to DATETIMEOFFSET data type in case of migration to ordinary table. But when migrating to memory-optimized table, SSMA issues a warning about changing a data type and changes DATETIMEOFFSET to DATETIME because of Memory-optimized table columns of types DATETIMEOFFSET are not allowed.

The same is with LOB Oracle datatypes. Memory-optimized table columns do not support long types. SSMA issues warning about that and changes VARCHAR(max), NVARCHAR(max) and VARBINARY(max) to VARBINARY(8000).

SSMA also changes CHAR and VARCHAR columns to NCHAR and NVARCHAR correspondingly as columns in memory-optimized tables with data types CHAR and VARCHAR must use code page 1252 (Latin*) only.

Calculated column are also not supported by in memory-optimized tables. You can rewrite your code so that it inserts the calculated values later every time inserts into the table are performed.

Memory-optimized tables also require that a PRIMARY KEY constraint exists on the table. I no unique columns exist on the table, SSMA creates an extra unique column. PRIMARY KEY is created as NONCLUSTERED only.

FOREIGN KEY, UNIQUE constraints, UNIQUE indexes and triggers are also not supported by memory-optimized tables. SSMA removes them from the table definition and generates a warning about this. Also SSMA removes ASC and DESC modifiers from an index definition in order to meet memory-optimized tables requirements.

Conversion to memory-optimized tables is not supported on Azure SQL DB.

Conclusion

This migration guide covers the differences between Oracle and SQL Server 2014 database platforms, and it includes the steps necessary to convert an Oracle database to SQL Server. It explains the algorithms that SSMA for Oracle uses to perform this conversion so that you can better understand the processes that are executed when you run the SSMA **Convert Schema** and **Migrate Data** commands. For those cases when SSMA does not handle a particular migration issue, approaches to manual conversion are included.

About DB Best Technologies

DB Best Technologies is a leading provider of database and application migration services and custom software development. We have been focused on heterogeneous database environments (SQL Server, Oracle, Sybase, DB2, MySQL) since starting at 2002 in Silicon Valley. Today, with over 140 employees in the United States and Europe, we develop database tools and provide services to customers worldwide.

DB Best developed migration tools to automate conversion between SQL dialects. In 2005 Microsoft acquired this technology, which later became a family of SQL Server Migration Assistant (SSMA) products. We continue to develop new versions of SSMA, and support Microsoft customers who are migrating to SQL Server.

We also provide migration services covering all major steps of a typical migration project: complexity assessment, schema conversion, data migration, application conversion, testing, integration, deployment, performance tuning, training, and support.

For more details, visit us at http://www.dbbest.com, e-mail us at info@dbbest.com, or call 1-855-855-3600.

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.