# logo-sql08.gif

Guide to Migrating from Oracle to SQL Server 2008

SQL Server Technical Article

**Writers:** Vladimir Kisil (DB Best Technologies), Valery Fomenko (DB Best Technologies), Yuri Rusakov (DB Best Technologies)

**Technical Reviewer:** Dmitry Balin (DB Best Technologies)

**Published:** August 2009

**Applies to:** SQL Server 2008 and SQL Server 2008 R2

**Summary:** This white paper explores challenges that arise when you migrate from an Oracle 7.3 database or later to SQL Server 2008. 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) 2008 for Oracle is explained in depth, with a special focus on converting database objects and PL/SQL code.

Created by: DB Best Technologies LLC

P.O. Box 7461, Bellevue, WA 98008

Tel.: (408) 202-4567

E-mail: [info@dbbest.com](mailto:info@dbbest.com)

Web: [www.dbbest.com](http://www.dbbest.com)

Copyright

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2009 Microsoft Corporation. All rights reserved.

Microsoft and SQL Server are registered trademarks of Microsoft Corporation in the United States and other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Contents

[Introduction 6](#_Toc237860997)

[Overview of Oracle-to-SQL Server 2008 Migration 7](#_Toc237860998)

[Main Migration Steps 7](#_Toc237860999)

[Conversion of Database Objects 7](#_Toc237861000)

[Differences in SQL Languages 9](#_Toc237861001)

[PL/SQL Conversion 9](#_Toc237861002)

[Data Migration Architecture of SSMA for Oracle 11](#_Toc237861003)

[Implementation in SSMA 11](#_Toc237861004)

[Solution Layers 11](#_Toc237861005)

[Client Application 11](#_Toc237861006)

[Stored Procedures Interface 12](#_Toc237861007)

[Database Layer 12](#_Toc237861008)

[Migration Executable 12](#_Toc237861009)

[Message Handling 13](#_Toc237861010)

[Validation of the Results 13](#_Toc237861011)

[Migrating Oracle Data Types 14](#_Toc237861012)

[Numeric Data Types 15](#_Toc237861013)

[Character Data Types 15](#_Toc237861014)

[Date and Time 17](#_Toc237861015)

[Boolean Type 17](#_Toc237861016)

[Large Object Types 17](#_Toc237861017)

[XML Type 18](#_Toc237861018)

[ROWID Types 18](#_Toc237861019)

[Migrating Oracle Spatial Data 19](#_Toc237861020)

[Emulating Oracle System Objects 21](#_Toc237861021)

[Converting Oracle System Views 21](#_Toc237861022)

[Converting Oracle System Functions 27](#_Toc237861023)

[Converting Oracle System Packages 37](#_Toc237861024)

[Converting Nested PL/SQL Subprograms 57](#_Toc237861025)

[Inline Substitution 57](#_Toc237861026)

[Emulation by Using Transact-SQL Subprograms 62](#_Toc237861027)

[Migrating Oracle User-Defined Functions 66](#_Toc237861028)

[Conversion Algorithm 66](#_Toc237861029)

[Converting Function Calls When a Function Has Default Values for Parameters and with Various Parameter Notations 73](#_Toc237861030)

[Migrating Oracle Triggers 75](#_Toc237861031)

[Conversion Patterns 77](#_Toc237861032)

[Emulating Oracle Packages 97](#_Toc237861033)

[Converting Procedures and Functions 97](#_Toc237861034)

[Converting Overloaded Procedures 98](#_Toc237861035)

[Converting Packaged Variables 98](#_Toc237861036)

[Converting Packaged Cursors 99](#_Toc237861037)

[Converting Initialization Section 100](#_Toc237861038)

[Package Conversion Code Example 102](#_Toc237861039)

[Emulating Oracle Sequences 104](#_Toc237861040)

[How SSMA for Oracle V4.0 Creates and Drops Sequences 104](#_Toc237861041)

[NEXTVAL and CURRVAL Simulation in SSMA for Oracle V4.0 106](#_Toc237861042)

[Examples of Conversion 107](#_Toc237861043)

[Migrating Hierarchical Queries 111](#_Toc237861044)

[Emulating Oracle Exceptions 115](#_Toc237861045)

[Exception Raising 115](#_Toc237861046)

[Exception Handling 117](#_Toc237861047)

[SSMA Exceptions Migration 118](#_Toc237861048)

[Migrating Oracle Cursors 121](#_Toc237861049)

[Syntax 121](#_Toc237861050)

[Declaring a Cursor 122](#_Toc237861051)

[Opening a Cursor 124](#_Toc237861052)

[Fetching Data 125](#_Toc237861053)

[CURRENT OF Clause 130](#_Toc237861054)

[Closing a Cursor 130](#_Toc237861055)

[Examples of SSMA for Oracle V4.0 Conversion 132](#_Toc237861056)

[Simulating Oracle Transactions in SQL Server 2008 136](#_Toc237861057)

[Choosing a Transaction Management Model 136](#_Toc237861058)

[Autocommit Transactions 136](#_Toc237861059)

[Implicit Transactions 136](#_Toc237861060)

[Explicit Transactions 136](#_Toc237861061)

[Choosing a Concurrency Model 137](#_Toc237861062)

[Make Transaction Behavior Look Like Oracle 137](#_Toc237861063)

[Simulating Oracle Autonomous Transactions 138](#_Toc237861064)

[Simulating Autonomous Procedures and Packaged Procedures 139](#_Toc237861065)

[Simulating Autonomous Functions and Packaged Functions 140](#_Toc237861066)

[Simulation of Autonomous Triggers 141](#_Toc237861067)

[Code Example 142](#_Toc237861068)

[Migrating Oracle Records and Collections 143](#_Toc237861069)

[Implementing Collections 143](#_Toc237861070)

[Implementing Records 153](#_Toc237861071)

[Implementing Records and Collections via XML 155](#_Toc237861072)

[Sample Functions for XML Record Emulation 158](#_Toc237861073)

[Emulating Records and Collections via CLR UDT 159](#_Toc237861074)

[Conclusion 167](#_Toc237861075)

[About DB Best Technologies 167](#_Toc237861076)

# Introduction

Migrating from an Oracle database to Microsoft® SQL Server® 2008 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 V4.0 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 2008. It describes the implementation differences of database objects, SQL dialects, and procedural code between the two platforms.

# Overview of Oracle-to-SQL Server 2008 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.

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](#_Data_Migration_Architecture).

## 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](#_Migrating_Oracle_Data).
* An Oracle view is converted to an SQL Server view. The only exception is the materialized view, which becomes an ordinary table. SSMA creates emulations for commonly used Oracle system views. For more information about system view conversion, see [Emulating Oracle System Objects](#_Emulating_Oracle_System).
* 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. The current version of SSMA does not support nested subprograms, but you can find methods to manually convert them in [Converting Nested PL/SQL Subprograms](#_Converting_Nested_PL/SQL).
* 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](#_Migrating_Oracle_User-Defined). SSMA emulates most of the Oracle standard functions. See the complete list in [Emulating Oracle System Objects](#_Emulating_Oracle_System).
* 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](#_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](#_Emulating_Oracle_Packages). In addition, SSMA can emulate some commonly used Oracle system packages. See their description in [Emulating Oracle System Objects](#_Emulating_Oracle_System).
* SQL Server has no exact equivalent to Oracle sequences. SSMA can use one of two sequence conversion methods. The first method is to convert a sequence to an SQL Server identity column. That is the optimal solution, but as Oracle sequence objects are not linked to tables, using sequences may not be compatible with identity column functionality. In that situation, SSMA uses a second method, which is to emulate sequences by additional tables. This is not as effective as the first method, but it ensures better compatibility with Oracle. See details in [Emulating Oracle Sequences](#_Emulating_Oracle_Sequences).
* Oracle private synonyms are converted to SQL Server synonyms stored in the target database. SSMA converts public synonyms to synonyms defined in the **sysdb** database.

## 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. 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](#_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](#_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](#_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 2008](#_Simulating_Oracle_Transactions) and [Simulating Oracle Autonomous Transactions](#_Simulating_Oracle_Autonomous).

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](#_Migrating_Oracle_Records).

# Data Migration Architecture of SSMA for Oracle

This section describes SSMA for Oracle V4.0 components and their interaction during data migration. The components execute on different computers and use Microsoft SQL Server 2008 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 V4.0 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

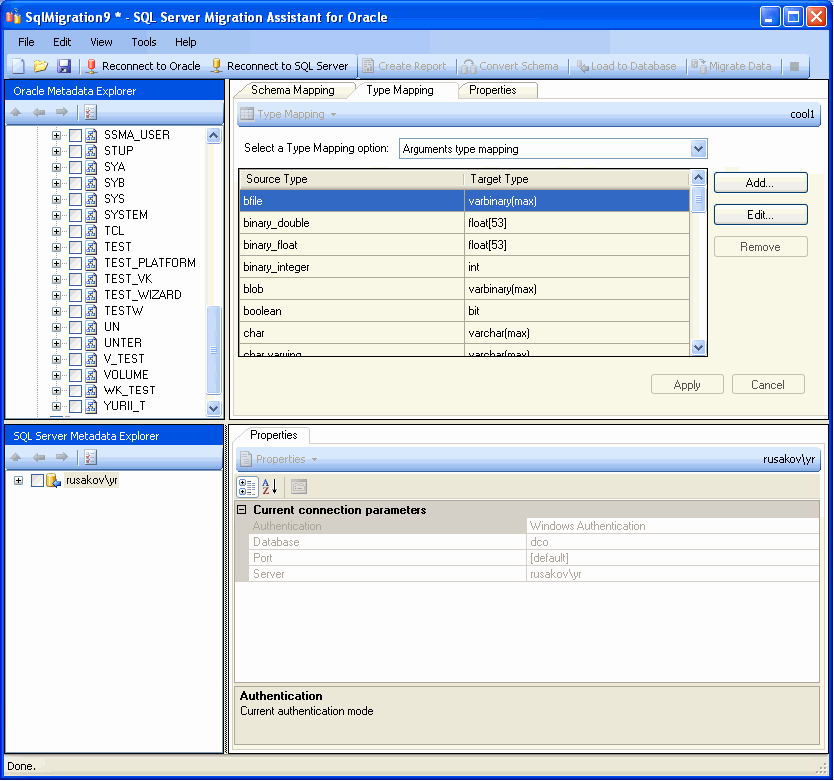
Before the migration starts, the client application calculates the number of rows in each table that will be migrated. With this data, you can evaluate the correct progress position.

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 2008. They differ in scale, precision, length, and functionality. This section explains the data type mapping implemented in SSMA for Oracle V4.0, and it includes remarks about conversion issues.

SSMA supports the ANSI and DB2 types implemented in Oracle, as well as the 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).



**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 2008 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 2008 can store time zone information in dates. This is supported by the [datetimeoffset](http://msdn.microsoft.com/en-us/library/bb630289.aspx) 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 new types introduced in SQL Server 2008: **varchar(max)**, **nvarchar(max)**, and **varbinary(max)**.

|  |  |
| --- | --- |
| **Oracle** | **SQL Server 2008** |
| 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 2005 and SQL Server 2008 operations over new types are simple compared to the approaches in both Oracle and SQL Server 2000. 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 2008 FILESTREAM attribute with the **varbinary(max)** data type.

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 2008 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 2008 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 2008, including Express.

SSMA for Oracle V4.0 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 2008 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.

# Emulating Oracle System Objects

This section describes how SSMA for Oracle V4.0 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 V4.0 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 2008. 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 V4.0

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 V4.0 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

### 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 2008: 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')

### DBA\_FREE\_SPACE System View

SSMA does not automatically convert DBA\_FREE\_SPACE. You can emulate it in SQL Server 2008 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

### DBA\_SEGMENTS System View

SSMA does not automatically convert the DBA\_SEGMENTS view. You can emulate it in SQL Server 2008 with the following columns: owner, segment\_name, segment\_type, bytes.

We propose the following emulation:

CREATE VIEW ssma\_ora.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

## 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 sysdb database when you install the SSMA Extension Pack. 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 sysdb.ssma\_oracle schema, as required for SQL Server functions that are part of the Extension Pack installation.

| **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) |  |
| 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) |  |
| COS(p1) | Y | M | COS(p1) |  |
| COSH(p1) | Y | F | ssma\_oracle.COSH(p1) no spaces are allowed in ssma\_ora user name. |  |
| COUNT(p1) | Y | M | COUNT(p1) |  |
| 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\_HOUR, TIMEZONE\_MINUTE, TIMEZONE\_REGION, TIMEZONE\_ABBR) a message is generated saying that it is impossible to convert. |
| FLOOR(p1) | Y | M | FLOOR(p1) |  |
| FROM\_TZ(p1, p2) | Y | M | **TODATETIMEOFFSET(p1, p2)** |  |
| GREATEST(p1,p2 | P | F | ssma\_oracle. | Function type is based on the p1 data type. If the Oracle source is |
| [,p3…pn]) | GREATEST\_DATETIME(p1, p2) | GREATEST(p1,p2,p3), SSMA transforms  it as |
|  | GREATEST\_FLOAT(p1, p2) | GREATEST(p1, GREATEST(p2,p3)) and so on. |
|  | GREATEST\_INT(p1, p2) |  |
|  | GREATEST\_NVARCHAR(p1, p2) |  |
|  | GREATEST\_REAL(p1, p2) |  |
|  | GREATEST\_VARCHAR(p1, p2) |  |
| INITCAP(p1) | Y | F | ssma\_oracle. | Function type is based on the p1 data type. Currently supports the following argument types: CHAR, NCHAR, VARCHAR2, NVARCHAR2. For other types, a message is generated. |
| INITCAP \_VARCHAR(p1) |
| INITCAP \_NVARCHAR(p1) |
| INSTR(p1,p2[,p3,p4]) | P | F | ssma\_oracle. | INSTRB,  INSTRC,  INSTR2, INSTR4  currently not converted. |
| 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) |
| LAST\_DAY(p1) | Y | F | ssma\_oracle.LAST\_DAY(p1) |  |
| LEAST(p1, p2 [, p3 … pn]) | P | F | ssma\_oracle. | Function type is based on the p1 data type. If Oracle source is |
| LEAST\_DATETIME (p1, p2) | LEAST (p1,p2,p3), SSMA transforms it as |
| LEAST\_FLOAT (p1, p2) | LEAST (p1,  LEAST (p2,p3)) and so on. |
| LEAST\_INT (p1, p2) |  |
| LEAST\_NVARCHAR (p1, p2) |  |
| LEAST\_REAL (p1, p2) |  |
| LEAST\_VARCHAR (p1, p2) |  |
| LENGTH(p1) | P | F | ssma\_oracle. | LENGTHB, LENGTHC, LENGTH2, LENGTH4 currently not converted. |
| LENGTH\_CHAR(p1) | Function type determined based on the p1 data type. |
| LENGTH\_NCHAR(p1) |  |
| 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. | 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\_VARCHAR(p1, p2, p3) |
| LPAD\_NVARCHAR(p1, p2, p3) |
| LPAD(p1, p2, p3) | Y | F | ssma\_oracle. | Function type is based on the p1 data type. Currently supports the following argument types: CHAR, NCHAR, VARCHAR2, NVARCHAR2. |
| LPAD\_VARCHAR(p1, p2, p3) |
| LPAD\_NVARCHAR(p1,p2,p3) |
| LTRIM(p1) | Y | M | LTRIM(p1) |  |
| LTRIM(p1, p2) | Y | F | ssma\_oracle. | Function type is based on the p1 data type. Currently supports the following argument types: CHAR, NCHAR, VARCHAR2, NVARCHAR2. |
| LTRIM2\_VARCHAR(p1, p2) |
| LTRIM2\_NVARCHAR(p1, p2) |
| 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. | 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. |
| NLS\_INITCAP\_NVARCHAR(p1) |
| NTILE() | Y | M | NTILE() |  |
| NULLIF(p1, p2) | Y | M | NULLIF(p1, p2) |  |
| NVL(p1, p2) | Y | M | ISNULL(p1, p2) |  |
| 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. |
| 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 numeric ] | Y | M | ROUND (p1, p2) |  |
| ROW\_NUMBER() | Y | M | ROW\_NUMBER() |  |
| RPAD(p1, p2) | Y | F | ssma\_oracle. | 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\_VARCHAR(p1, p2, p3) |
| RPAD\_NVARCHAR(p1, p2, p3) |
| RPAD(p1, p2, p3) | Y | F | ssma\_oracle. | 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 |
| RPAD\_VARCHAR(p1, p2, p3) |
| RPAD\_NVARCHAR(p1,p2,p3) |
| RTRIM(p1) | Y | M | RTRIM(p1) |  |
| RTRIM(p1,p2) | Y | F | ssma\_oracle. | The function type is based on the p1 data type. Currently supported following argument types are: CHAR, NCHAR, VARCHAR2, NVARCHAR2. |
| RTRIM2\_VARCHAR(p1,p2) |
| RTRIM2\_NVARCHAR(p1,p2) |
| 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. | The function type is based on the p1 data type. |
| 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) |
| 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 work correctly. |
| TO\_CHAR(p1, p2) | P | F | ssma\_oracle. | 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.’ |
| TO\_CHAR\_DATE (p1, p2) | Character value of p1 is not supported. |
| TO\_CHAR\_NUMERIC (p1, p2) |  |
| 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. | 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 message is generated. |
| TRANSLATE\_VARCHAR(p1, p2, p3) |
| TRANSLATE\_NVARCHAR(p1, p2, p3) |
| TRUNC(p1[, p2]) | Y | F | ssma\_oracle. | Currently supported only for p1 of NUMERIC and DATE types. |
| TRUNC(p1[, p2]) |
| TRUNC\_DATE(p1) |
| TRUNC\_DATE2(p1, p2) |
| 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) |  |

## 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 dynamic SQL is processed manually.
* The statement is not SELECT.

| **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 sysdb.ssma\_oracle.dbms\_sql\_open\_cursor @result = @cur OUTPUT

EXECUTE sysdb.ssma\_oracle.dbms\_sql\_parse @cur, 'SELECT t1.col1 FROM dbo.t1'

DECLARE @temp nvarchar(4000)

SET @temp = db\_name()

EXECUTE sysdb.ssma\_oracle.dbms\_sql\_execute @cur, @temp, @ssma$rows\_processed = @ret OUTPUT

EXECUTE sysdb.ssma\_oracle.dbms\_sql\_close\_cursor @cur

end

### 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 sysdb.ssma\_oracle.UTL\_FILE\_FOPEN$IMPL 'USER\_DIR', '1.txt', 'w', 1280, @outfile OUTPUT

EXEC sysdb.ssma\_oracle.UTL\_FILE\_PUT\_LINE @outfile, 'Hello, world!'

EXEC sysdb.ssma\_oracle.UTL\_FILE\_PUT @outfile, 'Hello, world NEW! '

EXEC sysdb.ssma\_oracle.UTL\_FILE\_FFLUSH @outfile

IF (sysdb.ssma\_oracle.UTL\_FILE\_IS\_OPEN(@outfile) != /\* FALSE \*/ 0)

EXEC sysdb.ssma\_oracle.UTL\_FILE\_FCLOSE @outfile

EXEC sysdb.ssma\_oracle.UTL\_FILE\_FOPEN$IMPL 'USER\_DIR', '1.txt', 'r', 1024, @outfile OUTPUT

EXEC sysdb.ssma\_oracle.UTL\_FILE\_GET\_LINE @outfile, @V1 OUTPUT, 32767

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

IF (sysdb.ssma\_oracle.UTL\_FILE\_IS\_OPEN(@outfile) != /\* FALSE \*/ 0)

EXEC sysdb.ssma\_oracle.UTL\_FILE\_FCLOSE @outfile

END

### 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 | [sysdb].ssma\_oracle.unique\_session\_id() | Return value is different |

### DBMS\_PIPE Package

SSMA for Oracle V4.0 does not convert the DBMS\_PIPEsystem 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:

Use sysdb

Go

Create Table sysdb.ssma.Pipes(

ID Bigint Not null Identity(1, 1),

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

DataValue Varchar(8000)

);

go

Grant Select, Insert, Delete On sysdb.ssma.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 sysdb.ssma.Pipes where PipeName = '<Pipe\_Name>'

If @s is not null

Begin

Select @chr = DataValue From sysdb.ssma.Pipes where ID = @s

Delete From sysdb.ssma.Pipes where ID = @s

End

END

*Oracle*

dbms\_pipe.pack\_message(info);

status := dbms\_pipe.send\_message('<Pipe\_Name>');

*SQL Server*

INSERT INTO sysdb.ssma.Pipes (PipeName, DataValue) Values ('<Pipe\_Name>', @info)

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 extension pack procedures and functions.

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 discussed are:

Submit a job to the job queue:

DBMS\_JOB.SUBMIT (

<job\_id> OUT binary\_integer,

<what> IN varchar2,

<next\_date> IN date DEFAULT defaultsysdate,

<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 **sysdb** 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 V4.0.

*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 sysdb 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-----------------------------

use sysdb

go

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

use sysdb

go

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

# 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 2008 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

AS

BEGIN

DECLARE

@on\_year int = 2000,

@dept\_sales int = 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

EXECUTE sysdb.ssma\_oracle.db\_error\_exact\_one\_row\_check @@ROWCOUNT

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 = 2000,

@lv\_out\_sales int,

@dept\_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

EXECUTE sysdb.ssma\_oracle.db\_error\_exact\_one\_row\_check @@ROWCOUNT

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

EXECUTE sysdb.ssma\_oracle.db\_error\_exact\_one\_row\_check @@ROWCOUNT

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

EXECUTE sysdb.ssma\_oracle.db\_error\_exact\_one\_row\_check @@ROWCOUNT

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 V4.0 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 (=).
* TheAUTHID 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 = sysdb.ssma\_ora.get\_active\_spid(),@login\_time = sysdb.ssma\_ora.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](#_Simulating_Oracle_Autonomous).

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\_1>

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;

# Migrating Oracle Triggers

This section describes the differences between Oracle and Microsoft SQL Server 2008 triggers, and how SSMA for Oracle V4.0 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.



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, INSERTorUPDATE), 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.

**Pattern for the trigger body**

declare @spid int, @login\_time datetime

select @spid = ssma\_ora.get\_active\_spid(),

@login\_time = ssma\_ora.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 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.

# Emulating Oracle Packages

Oracle supports encapsulating variables, types, stored procedures, and functions into a package. This section describes SSMA for Oracle V4.0 conversion algorithms, which allow packages to be emulated in Microsoft SQL Server 2008.

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 2008, 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 2008, 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](#_Migrating_Oracle_User-Defined).

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 2008, 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 V4.0 provides an excellent solution. For the task, SSMA uses special tables that reside in a sysdb database. 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 **sysdb** 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 V4.0 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](#_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 because 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 sysdb 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 **sysdb.ssma\_oracle.db\_check\_init\_package** procedure. That procedure checks if the package is not yet initialized, 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 sysdb.ssma\_oracle.db\_clean\_storage

EXECUTE sysdb.ssma\_oracle.set\_pv\_varchar

'SYS',

'DBO',

'MY\_PACKAGE',

'SPACE',

' '

EXECUTE sysdb.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 sysdb.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 sysdb.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 sysdb.ssma\_oracle.db\_check\_init\_package 'SCOTT', 'DBO', 'MY\_PACKAGE'

PRINT @s

END

GO

CREATE PROCEDURE dbo.MY\_PACKAGE$SSMA\_Initialize\_Package

AS

EXECUTE sysdb.ssma\_oracle.db\_clean\_storage

EXECUTE sysdb.ssma\_oracle.set\_pv\_varchar

'SCOTT',

'DBO',

'MY\_PACKAGE',

'SPACE',

' '

EXECUTE sysdb.ssma\_oracle.set\_pv\_varchar

'SCOTT',

'DBO',

'MY\_PACKAGE',

'UNITNAME',

'My Simple Package'

DECLARE

@temp datetime

SET @temp = getdate()

EXECUTE sysdb.ssma\_oracle.set\_pv\_datetime

'SCOTT',

'DBO',

'MY\_PACKAGE',

'CURD',

@temp

GO

# Emulating Oracle Sequences

When migrating from Oracle to Microsoft SQL Server 2008, you must remember that SQL Server 2008 does not natively support sequences as Oracle does. But with SSMA for Oracle V4.0, it is easy to simulate Oracle sequences by using an SSMA function.

The essential tasks that the sequences simulating engine should provide are:

* Generate the next value of a sequence by using the NEXTVAL method.
* Retrieve current value of the sequence by using the CURRVAL method. This value is bound to the current session scope.
* Keep the sequence value if the transaction is rolled back.

The SSMA for Oracle V4.0 solution is based on SQL Server identity columns. A table with an identity column is created for every sequence. In the IDENTITY property, the same properties are used as in the ORACLE sequence, except for MAXVALUE, MINVALUE, and CYCLE. The identity value is transaction-independent.

## How SSMA for Oracle V4.0 Creates and Drops Sequences

The following procedures are intended for sequence DML operations, which are creation and dropping.

sysdb.ssma\_oracle.db\_create\_sequence

@dbname,

@schema,

@name,

@seed,

@increment

Arguments:

* @dbname: The name of the database that contains the sequence.
* @schema: The name of the schema that contains the sequence.
* @name: The sequence name.
* @seed: The seed value.
* @increment: The increment value.

The procedure creates a permanent table with the name that identifies the sequence. The table has one identity column of **numeric**(38) data type named as ID. Also, the **db\_create\_sequence** procedure creates a procedure that inserts the default value into the given table. The procedure is created in the same database in which the sequence table is located. Execute permission on the procedure is granted to public when the sequence is created, giving users indirect access to the sequence tables.

The following example creates a sequence with the name orders\_seq in the target database:

exec sysdb.ssma\_oracle.db\_create\_sequence @dbname = 'customers', @name = 'orders\_seq', @increment = 2

The following function drops the sequence:

sysdb.ssma\_oracle.db\_drop\_sequence

@dbname,

@schema,

@name

Arguments:

* @dbname: The database name that contains the sequence.
* @schema: The schema name that contains the sequence.
* @name: The sequence name.

The following example drops a sequence named orders\_seq in the target database:

exec ssma.db\_drop\_sequence @dbname = 'customers', @name = 'orders\_seq'

## NEXTVAL and CURRVAL Simulation in SSMA for Oracle V4.0

In SSMA for Oracle V4.0, ORACLE sequence simulation is implemented via both Transact-SQL procedures and functions. The implementation of a sequence via a Transact-SQL procedure does not allow its use in DML commands, but it significantly improves performance.

The NEXTVAL simulation method executes an insert command. The insert command is rolled back immediately to keep the table empty. This approach gains maximum speed.

If there is an external transaction, the transaction point is saved and the transaction is rolled back to it after insert.

The following procedure is the stored procedure version of NEXTVAL:

sysdb.ssma\_oracle.db\_sp\_get\_next\_sequence\_value(

@dbname,

@schema,

@name,

[@curval] output

Arguments:

* @dbname: The name of the database that contains the sequence.
* @schema: The name of the schema that contains the sequence.
* @name: The sequence name.
* @curval: The current value of a sequence.

The ORACLE sequence implementation via a Transact-SQL function allows using it in DML commands. Because Transact-SQL functions cannot use DML commands and invoke stored procedures, an SSMA NEXTVAL function implementation issues an autonomous command via **xp\_ora2ms\_exec2** to invoke the NEXTVAL procedure version. This causes a decrease in performance as compared with the procedure version.

The following function is the user-defined function version of NEXTVAL:

sysdb.ssma\_oracle.db\_get\_next\_sequence\_value(@dbname,@schema,@name)

Arguments:

* @dbname: The name of the database that contains the sequence.
* @schema: The name of the schema that contains the sequence.
* @name: The sequence name.

Return types: numeric(38,0).

The following function returns the current value of a sequence:

sysdb.ssma\_oracle. db\_get\_curval\_sequence\_value(@dbname,@schema,@name)

Arguments:

* @dbname: The database name that contains the sequence.
* @schema: The schema name that contains the sequence.
* @name: The sequence name.

Return types: numeric(38,0).

## Examples of Conversion

### ****Inserting Sequence Values into a Table****

This example increments the employee sequence and uses its value for a new employee inserted into the sample table employees.

*Oracle*

INSERT INTO employees (id, name)

VALUES(employees\_seq.nextval, 'David Miller');

*Transact-SQL*

DECLARE @nextval numeric(38, 0)

EXECUTE sysdb.ssma\_oracle.db\_sp\_get\_next\_sequence\_value 'customers','dbo','employees\_seq', @nextval OUTPUT

INSERT employees (id, name) VALUES(@nextval, 'David Miller')

The following statement more closely follows the original but takes more time to execute:

INSERT employees (id, name) VALUES(sysdb.ssma\_oracle.db\_get\_next\_sequence\_value ('customers', 'dbo', 'employees\_seq'), 'David Miller')

The second example adds a new order with the next order number to the order table. Then it adds suborders with this number to the detail order table.

*Oracle*

INSERT INTO orders(id, customer\_id)

SELECT orders\_seq.nextval, customer\_id from orders\_cache;

INSERT INTO order\_items (order\_id, line\_item\_id, product\_id)

VALUES (orders\_seq.currval, 1, 2412);

INSERT INTO order\_items (order\_id, line\_item\_id, product\_id)

VALUES (orders\_seq.currval, 2, 3456);

*Transact-SQL*

INSERT orders(id, customer\_id)

SELECT sysdb.ssma\_oracle.db\_get\_next\_sequence\_value('customers', 'dbo', 'orders\_seq'), customer\_id from orders\_cache;

INSERT order\_items(order\_id, line\_item\_id, product\_id)

SELECT sysdb.ssma\_oracle.db\_get\_curval\_sequence\_value ('customers ', 'dbo', 'orders\_seq'), 1, 2412);

INSERT order\_items(order\_id, line\_item\_id, product\_id)

SELECT sysdb.ssma\_oracle.db\_get\_curval\_sequence\_value ('customers ', 'dbo', 'orders\_seq'), 2, 3456);

### Optimization Tips

You can try an easier way to convert your Oracle sequences and get more performance, but only if you know exactly how the sequence is used. For example, if there are no methods using CURRVAL without previous NEXTVAL calls, you need not save and store the current sequence value, and you can use a local variable to store it. That gains performance because it’s not necessary to use DML routines to save and get the sequence current value.

For example, if you have an ORACLE sequence:

CREATE SEQUENCE employees\_seq INCREMENT BY 1 START WITH 1

You must create a table with an IDENTITY column:

create table employees\_seq (id numeric(38) identity(1,1))

The statement INSERT INTO..VALUES can be transformed to Transact-SQL in the following way:

*Oracle*

begin

INSERT INTO employees (id, name)

VALUES(employees\_seq.nextval, 'David Miller');

end;

*Transact-SQL*

begin

declare @curval numeric(38)

begin tran

insert employees\_seq default values

set @curval=scope\_identity()

rollback

INSERT INTO employees (id, name)

VALUES(@curval, 'David Miller');

end;

You can wrap the INSERT statement in a stored procedure. Additionally, it should check for an external opened transaction. If one exists, the transaction point should be saved; a new one should not be opened:

create proc employees\_seq\_nextval(@curval numeric(38) out = null)

as

declare @tran bit

set @tran = 0

if @@trancount>0

begin

save tran seq

set @tran = 1

end

else begin tran

insert employees\_seq default values

set @curval=scope\_identity()

if @tran=1

rollback tran seq

else rollback

Then the statement can be transformed to the following:

begin

declare @curval numeric(38)

exec employees\_seq\_nextval @curval out

INSERT INTO employees (id, name) VALUES(@curval, Dylan Miller');

end;

To convert statements where the next value of a sequence is retrieved in DML statements such as INSERT INTO..SELECT, wrap your stored procedure for getting a sequence in a function. You can do so with a master..**xp\_ora2ms\_exec2** extended procedure that helps to invoke stored procedures from a function body.

To invoke the **xp\_ora2ms\_exec2** procedure, you must pass the current process ID and login time as parameters:

create function fn\_employees\_seq\_nextval() RETURNS numeric(38,0)

as begin

declare @curval numeric(38,0)

declare @spid int, @login\_time datetime

select @spid = sysdb.ssma\_oracle.get\_active\_spid(),@login\_time = sysdb.ssma\_oracle.get\_active\_login\_time()

exec master..xp\_ora2ms\_exec2 @spid,@login\_time,'orders','dbo',

'employees\_seq\_nextval',@dbname,@schema,@name,@curval output

return @curval

end

# 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:

1. Oracle selects the hierarchy’s root row(s) (those rows that satisfy the START WITH condition).
2. 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.
3. 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.
4. 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.
5. Oracle returns the rows in the order shown in Figure 3. In the figure, children appear below their parents.

1

2

7

8

3

4

9

5

6

10

11

12

**Figure 3:** An example of the Oracle tree traversal order

In SQL Server 2008, 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://msdn2.microsoft.com/en-us/library/ms186243.aspx) (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 2008.
* 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 2008 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 2008 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 2008*

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

IF @@ROWCOUNT = 0

BEGIN

<Statements>

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 2008*

BEGIN TRY

…

IF <condition>

RAISERROR (‘myexception’, 16, 1)

…

END TRY

BEGIN CATCH

IF ERROR\_MESSAGE() = ‘myexception’

BEGIN

<Statements>

END

ELSE

<rest\_of\_handler code>

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 2008 error numbers.

Note that SQL Server 2008 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 2008*

BEGIN TRY

…

INSERT INTO <table> VALUES …

…

END TRY

BEGIN CATCH

…

IF ERROR\_NUMBER() = 2627

<Statements>

…

END CATCH

Unfortunately, SQL Server 2008 does not support exception reraising. If the exception is not handled, it can be passed to the calling block by using the RAISERROR statement with a custom error number and appropriate message.

## SSMA Exceptions Migration

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 **sysdb.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 as an alternative execution block after all handlers.
5. If there is no OTHERS exception handler, the exception is reraised by the special stored procedure **sysdb.ssma\_oracle.ssma\_rethrowerror** that emulates reraising using a custom error number. It also emulates a RAISE statement with no exception name.
6. To emulate predefined Oracle exceptions NO\_DATA\_FOUND and TOO\_MANY\_ROWS, the special stored procedure EXEC **sysdb.ssma.db\_error\_exact\_one\_row\_check** @@ROWCOUNT is placed after all SELECT statements. The procedure checks the row count and raises an exception with the custom number 59999 and the message ‘ORA+00100’ or ‘ORA-01422,’ depending on its value.
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’)

1. All exceptions are raised with severity level 16 to provide handling by a CATCH block.
2. **sysdb.ssma.db\_error\_sqlcode** user-defined function emulates the SQLCODE function. It returns an Oracle error number.
3. Either **sysdb.ssma.db\_error\_sqlerrm\_0** or **sysdb.ssma.db\_error\_sqlerrm\_1** emulates the SQLERRM function, depending on the parameters.
4. SSMA does not support using the SQLCODE and SQLERRM functions outside of an EXCEPTION block.

# 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](#_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 2008, 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 2008 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; | DECLARE cursor\_name CURSOR [LOCAL | GLOBAL] [FORWARD\_ONLY | SCROLL] [STATIC | KEYSET | DYNAMIC | FAST\_FORWARD] [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 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 2008 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](#_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 2008 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. |
| 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](#_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

1. 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

1. 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 …;

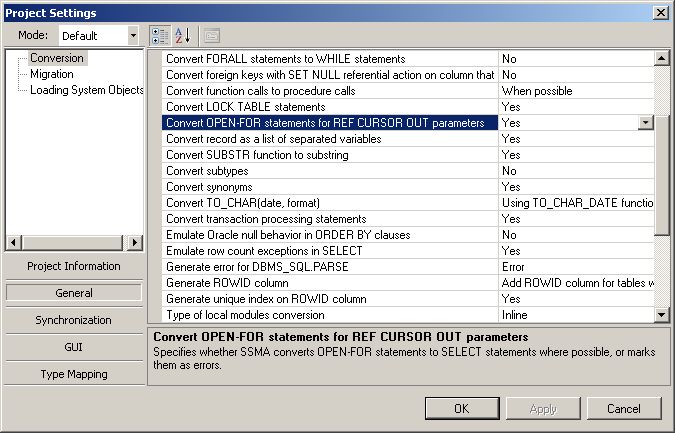
*SQL Server 2008*

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 subprogram 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).



**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'

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

1. 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 subprogram 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]]'

1. Then it generates the following error message: “OPEN ... FOR statement will be converted, but the dynamic string must be converted manually.”
2. 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.

1. 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.
2. 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 V4.0 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

# Simulating Oracle Transactions in SQL Server 2008

During migration from Oracle to Microsoft SQL Server 2008, you must account for the differences in their default transaction management behavior. SSMA for Oracle V4.0 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 2008 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 2008 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 2008 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 2008. 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 2008 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 2008 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 2008, 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.

# Simulating Oracle Autonomous Transactions

This section describes how SSMA for Oracle V4.0 handles autonomous transactions (PRAGMA AUTONOMOUS\_TRANSACTION). These autonomous transactions do not have direct equivalents in Microsoft SQL Server 2008.

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 2008 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 4.0 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 Extension Pack 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 = sysdb.ssma\_oracle.get\_active\_spid(),

@login\_time = sysdb.ssma\_oracle.get\_active\_login\_time()

EXEC master.dbo.xp\_ora2ms\_exec2\_ex @spid, @login\_time, <database\_name>, <schema\_name>, <procedure\_name>, [parameter1, parameter2, ... ]

## 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 = sysdb.ssma\_oracle.get\_active\_spid(),

@login\_time = sysdb.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 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 = sysdb.ssma\_oracle.get\_active\_spid(),

@login\_time = sysdb.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](#_Autonomous_Transactions_in).

## Code Example

The following code provides CREATE PROCEDURE examples that can be leveraged for Oracle and SQL Server 2008 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 2008*

CREATE PROCEDURE dbo.UPDATE\_SALARY @emp\_id float(53)

AS BEGIN

DECLARE @active\_spid INT, @login\_time DATETIME

SET @active\_spid = sysdb.ssma\_oracle.GET\_ACTIVE\_SPID()

SET @login\_time = sysdb.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

IF @@TRANCOUNT > 0

COMMIT WORK

END TRY

BEGIN CATCH

IF @@TRANCOUNT > 0

ROLLBACK WORK

END CATCH

END

# Migrating Oracle Records and Collections

Unlike Oracle, Microsoft SQL Server 2008 supports neither records nor collections. When you migrate from Oracle to SQL Server 2008, therefore, you must apply substantial transformations to the PL/SQL code.

The approach used by SSMA for Oracle V4.0 is to convert both records and collections as a user-defined type implemented as SQL CLR type.

**Note**: SSMA for Oracle V3.0 does not convert collections. Therefore, this section describes manual migration activity.

## Implementing Collections

To emulate collections, you have four options:

* [Option 1](#Option1). Rewrite your PL/SQL code to avoid collections.
* [Option 2](#Option2). When collections are used within the scope of a subroutine, you can use SQL Server table variables.
* [Option 3](#Option3). When you pass a collection as a parameter into a procedure or a function, a local temporary table can be the solution.
* [Option 4](#Option4). 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](#_Implementing_Records_and).
* 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 V4.0. For more information, see [Emulating Records and Collections via CLR UDT](#_Emulating_Records_and).

**Option 1**. Rewrite your code to avoid records and collections. In many cases, collections or records are not justified. Generally, you can perform the sametasks 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 the 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  or  set @i\_last=null  select top 1 @i\_first = \_idx\_ from @emps order by \_idx\_ asc | Comment on set @i\_last=null  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  from emp | INSERT INTO @emps (\_idx\_, empno)  SELECT row\_number() over(order by empno) as \_idx\_, empno  from emp  or  INSERT INTO @emps (empno)  SELECT empno from emp | The row\_number() function depends 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 = @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](#_Sample_Functions_for).)

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 V4.0 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 = sysdb.ssma\_oracle.SetRecord\_varchar(@emps, N'ENAME', @emps$ename)

SET @emps = sysdb.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 = sysdb.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) (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 V4.0 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:

sysdb.ssma\_oracle.fn\_bulk\_collect2CollectionSimple sysdb.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(sysdb.ssma\_oracle.fn\_bulk\_  
collect2CollectionSimple((select column\_name\_1 from <data\_source> for xml path)))

1. 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(sysdb.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)))

1. 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(sysdb.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(sysdb.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.

# Conclusion

This migration guide covers the differences between Oracle and SQL Server 2008 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 75 employees in the United States and Europe, we develop database tools and provide services to customers worldwide.

DB Best developed migration tools to automate conversion between SQL dialects. In 2005 Microsoft acquired this technology, which later became a family of SQL Server Migration Assistant (SSMA) products. We continue to develop new versions of SSMA, and support Microsoft customers who are migrating to SQL Server.

We also provide migration services covering all major steps of a typical migration project: complexity assessment, schema conversion, data migration, application conversion, testing, integration, deployment, performance tuning, training, and support.

For more details, visit us at <http://www.dbbest.com>, e-mail us at [info@dbbest.com](mailto:info@dbbest.com), or call 1-408-202-4567.

**For more information:**

<http://www.microsoft.com/sqlserver/>: SQL Server Web site

<http://technet.microsoft.com/en-us/sqlserver/>: SQL Server TechCenter

<http://msdn.microsoft.com/en-us/sqlserver/>: SQL Server DevCenter

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

* Are you rating it high due to having good examples, excellent screenshots, clear writing, or another reason?
* Are you rating it low due to poor examples, fuzzy screenshots, unclear writing?

This feedback will help us improve the quality of the white papers we release.

[Send feedback](mailto:sqlfback@microsoft.com?subject=White%20Paper%20Feedback:%20Guide%20to%20Migrating%20from%20Oracle%20to%20SQL%20Server%202008).