

# Best Practices for Migrating Non-Unicode Data Types to Unicode

SQL Server Best Practices Article

Writer: Kohei Ueda

Technical Reviewers: Paul Mestemaker, Michael Redman, Wanda He, Juergen Thomas, Burzin Patel, Peter Scharlock, Michael Wang, Fernando Caro, Sanjay Mishra, Osamu Hirayama, Akio Jose

Published: January 2008

Applies To: SQL Server 2005

Summary: The database system in a company that is growing internationally must support multilingual characters in tables with Unicode data types. Existing databases that support only non-Unicode information must be migrated from non-Unicode data type to Unicode data type. This paper summarizes best practices for migrating database systems from non-Unicode to Unicode data type.

### Copyright

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.

 2008 Microsoft Corporation. All rights reserved.

Microsoft and SQL Serverare either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

### Table of Contents

[Introduction 1](#_Toc187263380)

[Data Type Migration 1](#_Toc187263381)

[ALTER TABLE 2](#_Toc187263382)

[Flat File 2](#_Toc187263383)

[SELECT INTO 3](#_Toc187263384)

[INSERT SELECT 3](#_Toc187263385)

[SQL Server Management Studio 3](#_Toc187263386)

[Changing Collation 4](#_Toc187263387)

[Test Objective 4](#_Toc187263388)

[Test Methodology 4](#_Toc187263389)

[Test Results and Observations 7](#_Toc187263390)

[Results 7](#_Toc187263391)

[Duration and Resource Utilization 7](#_Toc187263392)

[Inside Operations of SSMS Migration 13](#_Toc187263393)

[Changing Collation 14](#_Toc187263394)

[Observations 16](#_Toc187263395)

[Comparison of ALTER TABLE, Flat File, and INSERT SELECT Migrations 16](#_Toc187263396)

[Database Availability 17](#_Toc187263397)

[Comparison of INSERT SELECT and SSMS Migration 19](#_Toc187263398)

[Recommendations 19](#_Toc187263399)

[References 20](#_Toc187263400)

[Appendix A 21](#_Toc187263401)

[Appendix B 23](#_Toc187263402)

#### Introduction

In today's global economy, it is increasingly necessary for database systems to support multilingual characters in the database tables. Companies that manage only non-Unicode information are faced with altering their database systems to support Unicode data types. It might also be necessary that their databases have different collations across several servers. For more information about collations and Unicode data types, see [Collation Options and International Support](http://msdn2.microsoft.com/en-us/library/ms143503.aspx) in SQL Server 2005 Books Online.

This paper summarizes best practice recommendations about migrating tables from non-Unicode data type to Unicode data type. It also provides information about changing collations. Many packaged applications or line-of-business applications provide their own tools and procedures for changing data types from non-Unicode to Unicode. Many of these applications also require a different set of executables to work with data stored in Unicode on the database side. The transformation of the data in the database to Unicode is only one part of a migration to Unicode. In most cases, applications that use the data need extensive changes in order to work perfectly with data stored in Unicode.

This paper describes possibilities for transforming the data on the database side and covers some potential problems to watch out for. It does not cover preparing an existing application to work with data stored in Unicode.

#### Data Type Migration

Changing the data type of a column is a straightforward task: use SQL Server Management Studio (SSMS) to alter the table or run a single ALTER TABLE statement. Table 1 shows non-Unicode data types and their corresponding Unicode data types. The data length *n* specifies the number of characters. For example, you would convert a column with the non‑Unicode data type **char**(5000) to the Unicode data type **nvarchar(max)**. For details about the size of Unicode data types, see [nchar and nvarchar (Transact-SQL)](http://msdn2.microsoft.com/en-us/library/ms186939.aspx) in SQL Server Books Online.

Table 1: Data type conversion

|  |  |  |
| --- | --- | --- |
| **Non-Unicode data type** | **Data length** | **Corresponding Unicode data type** |
| char(*n*) | *n* <= 4,000 | nchar(*n*) |
| *n* > 4,000 | nvarchar(max) |
| varchar(*n*) | *n* <= 4,000 | nvarchar(*n*) |
| *n* > 4,000 | nvarchar(max) |
| varchar(MAX) | - | nvarchar(max) |
| text | - | nvarchar(max)[[1]](#footnote-2) |

But when you want to change all character type columns throughout a database from non-Unicode to Unicode, there are a number of things to consider, such as column size and disk size increase. These are described in the [Microsoft SQL Server ISV Program Management Team blog](http://blogs.msdn.com/mssqlisv/archive/2006/07/07/659374.aspx). This white paper covers migration methodologies.

There are a number of options for the migration:

* Run ALTER TABLE Transact‑SQL statements.
* Export and import data with flat files.
* Use a staging table and load data by using an SELECT INTO Transact‑SQL statement.
* Use a staging table and load data by using an INSERT SELECT Transact‑SQL statement.
* Change the table definition by using the SQL Server Management Studio interface.

##### ALTER TABLE

Execute ALTER statements for each character type column in each table in the database, as in the following example code:

alter table t1 alter column c1 nchar(10);

alter table t1 alter column c2 nvarchar(20);…

Where [t1] is defined as:

t1 (c1 char(10),c2 varchar(20),…)

##### Flat File

Use the bulk export and bulk import operations for data migration. For more information on bulk data operations, see [Importing and Exporting Bulk Data](http://msdn2.microsoft.com/en-us/library/ms175937.aspx) in SQL Server Books Online. There are two methods of flat file migration: normal flat file migration and flat file migration using staging tables.

To perform a flat file migration

1. Export data into flat files.
2. Drop the old tables.
3. Re-create the tables with Unicode data type.
4. Import data from the flat files.

To perform a migration using staging tables

1. Export data into flat files.
2. Create staging tables.
3. Import data from the flat files into the new tables.
4. Drop the old tables.
5. Rename the new table.

The second option uses staging tables to maintain the old data. The new tables are created with the exact same schema as the original tables except for the character type columns. For example, to create a staging table for table t1 used in the previous code example:

create table stg\_t1 (c1 nchar(10),c2 nvarchar(20),…)

A bulk export is performed by **bcp out** and a bulk import is performed by the BULK INSERT statement. The options for Unicode data type operations, the WITHDATATYPE=widechar option for BULK INSERT, and the **-w** option for **bcp out**, are required in the requests. [[2]](#footnote-3)

For more information on bulk data operations, see [Using Unicode Character Format to Import or Export Data](http://msdn2.microsoft.com/en-us/library/ms188289.aspx) in SQL Server 2005 Books Online.

The following example code is used for migration with a flat file:

bcp testdb.dbo.t1 out E:\t1.dat –w –Slocalhost –T

drop table t1

create table t1 (c1 nchar (10), c2 nvarchar(20),…)

bulk insert t1 from ’E:\t1.dat’ with (DATAFILETYPE=’widechar’)

For flat file migration with a staging table, additional disk space for the intermediate data is required, while migration without staging tables does not require this. However, as migration with staging tables maintains the old tables during data loading, it provides higher availability of the database than other types of migration. This is discussed later in this white paper.

##### SELECT INTO

This migration option uses a staging table to maintain the data in Unicode data type. This type of migration imports data from the original table directory by using a SELECT CAST(column1 AS nchar(n)) AS column1, … INTO stg\_Table1 FROM Table1 statement as in the following code example:

select cast(c1 as nchar(10)) as c1, cast(c2 as nvarchar(20)) as c2,… into stg\_t1 from t1

As this is the minimally logged operation, it should be the fastest option for data loading. However, it has limitations for the staging table:

* It always creates a new table on the default file group.
* The newly created table cannot be partitioned.

##### INSERT SELECT

This migration option uses a staging table to maintain the data in Unicode data type. This migration imports data from the original table directly by using SELECT \* as in the following code example:

create table stg\_t1 (c1 nchar(10), c2 nvarchar(20),…)

insert into stg\_t1 select \* from t1

Because it uses a table that already exists for the target, this option allows flexibility when creating the staging table such as using filegroups and partitioning.

##### SQL Server Management Studio

You manually alter the data type to migrate by using SQL Server Management Studio (SSMS). To do this:

1. In Object Explorer, expand the tree of the target database.
2. Right-click the target table, and then click **Design** to open Table Designer.
3. In Table Designer, change the data type of the character columns from the non-Unicode type to a Unicode type, such as from **char** to **nchar**.
4. Close Table Designer.

Basically, this follows the same steps as an INSERT SELECT in a background process. We recommend that you extend the timeout of Table Designer before converting table schemas. For more information on the timeout setting, see [Error message when you try to modify a large table by using SQL Server Management Studio: "Timeout expired"](http://support.microsoft.com/kb/915849/en-us) on the Microsoft Help and Support site.

##### Changing Collation

There are a number of business reasons to change the collation of a database. To read about the impact of changing collation, see [The Impact of Collation Change and Going from Non-Unicode to Unicode](http://download.microsoft.com/download/d/9/4/d948f981-926e-40fa-a026-5bfcf076d9b9/SQL_bestpract_CollationChange.docx) on the SQL Server Best Practices site. The biggest concern about changing collation is the possibility of data loss. As long as the source collation and the target collation are based on the same code page, data loss is not a concern. But data loss is possible when the collations are not compatible and a database table that has multi-byte data in non-Unicode columns is converted to a different collation without changing the columns from non-Unicode data type to Unicode data type. For example, going from Japanese collation to English collation can result in data corruption. This problem is demonstrated in the previous white paper. To avoid data loss, you must alter the data types from non-Unicode to Unicode (such as from **char**/**varchar** to **nchar**/**nvarchar**) before you change collations.

When a migrating database has objects, such as indexes, stored procedures, check constraints, triggers, and so on, that refer to a non-Unicode data type, you must alter or re-create these objects so that they can correctly handle Unicode data after the data is migrated. In addition, because it is impossible to specify more than 4,000 as the length of **nchar** or **nvarchar** data, you must use **nvarchar(max)** for the migration of tables that have **char** or **varchar** columns that are larger than 4,000 characters.

#### Test Objective

The objectives of this test were to characterize the migration scenarios described earlier in this paper and to find out what is the fastest and least intrusive way to achieve the migration.

#### Test Methodology

The goal of this test is to measure the duration and resource utilizations, such as disk space, for each migration scenario. To approximate a real-world scenario, we used a database that had multiple tables, indexes, primary key constraints, and foreign key constraints. Table 2 shows the size of the tables used in tests 1–5. The table schemas, indexes, and constraints are described in [Appendix A](#_Appendix_A:). Disk space utilization was measured by using the **sp\_spaceused** stored procedure.

In the survey we ran prior to this test, we found that most data type migration would be implemented during maintenance windows. Therefore, this test assumed that there was prepared downtime for the system and it was performed with the database recovery model set to simple. Before starting the test, we made a full backup of the database for restoring the initial state.

Table 2: Table Size

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Rows** | **Data (KB)** | **Index (KB)** |
| REGION | 5 | 8 | 8 |
| NATION | 25 | 8 | 24 |
| PART | 3,000,000 | 437,264 | 800 |
| SUPPLIER | 149,984 | 23,736 | 1,856 |
| PARTSUPP | 12,000,000 | 1,874,984 | 134,920 |
| CUSTOMER | 2,249,984 | 389,976 | 728 |
| OREDERS | 22,500,000 | 3,051,592 | 642,576 |
| LINEITEM | 89,987,373 | 14,872,120 | 4,739,640 |

The following table describes the test scenarios that were performed.

Table 3: Test Scenarios

|  |  |
| --- | --- |
| **Test #** | **Test Description** |
| 1 | **ALTER TABLE migration**   1. Measure the total time of the following steps: 2. Drop indexes, primary key constraints, and foreign key constraints for each table 3. Execute an ALTER TABLE statement for each character column on one of the target tables 4. Repeat step 1b for each table 5. Re-create the objects that were dropped in step 1a. 6. Measure the disk space used by data files and log files of the test database before and after the migration and at an intermediate state (just before building the indexes). |
| 2 | **Flat File migration**   1. Measure the total time of the following steps: 2. Export data from the target tables to flat files. 3. Drop foreign key constraints that refer to a target table. 4. Drop the target table. 5. Re-create the table with a Unicode data type. 6. Import data from the flat file into the table. 7. Re-create indexes, primary key constraints, and foreign key constraints on the table. 8. Repeat steps 1b–1e for each table. 9. Measure the total time of the following steps: 10. Export the data from the target tables to flat files. 11. Create staging tables with Unicode data type. 12. Import data from flat files into the staging tables. 13. Drop foreign key constraints for each table. 14. Drop the target tables. 15. Rename staging tables to original table names. 16. Re-create indexes, primary key constraints, and foreign key constraints on the tables. 17. Measure the disk space used by data files and log files of the test database before and after the migration and at an intermediate state (just before building indexes). |
| 3 | **SELECT INTO migration**   1. Measure the total time of the following steps: 2. Load data into the staging table by using SELECT INTO. 3. Repeat step 1a for each table. 4. Drop foreign key constraints. 5. Drop the original tables and rename each staging table to the original table name. 6. Re-create the indexes, primary key constraints, and foreign key constraints on the renamed tables. 7. Measure the disk space used by data files and log files of the test database before and after the migration and at an intermediate state (just before building indexes). |
| 4 | **INSERT SELECT migration**   1. Measure the total time of following steps: 2. Create the staging table. 3. Load data into the staging table by using INSERT SELECT. 4. Repeat steps 1a and 1b for each table. 5. Drop foreign key constraints. 6. Drop the original tables and rename each staging table to the original table name. 7. Re-create the indexes, primary key constraints, and foreign key constraints on the renamed tables. 8. Measure the disk space used by data files and log files of the test database before and after the migration and at an intermediate state (just before building indexes). |
| 5 | **Alter the table by using SSMS**  Use the Table Designer in SQL Server Management Studio to change the data type. Identify what happens in the background by tracing with SQL Server Profiler. |
| 6 | **Change collation**  This test is performed to clarify whether the migration scenarios described in Tests 1–5 can successfully migrate multi-byte data from one collation to another. We used a database that was created in Japanese collation (Japanese\_ CI\_AS), and changed the collation to English one (Latine1\_general\_CI\_AS). |

#### Test Results and Observations

This section describes the results of the tests and observations found.

##### Results

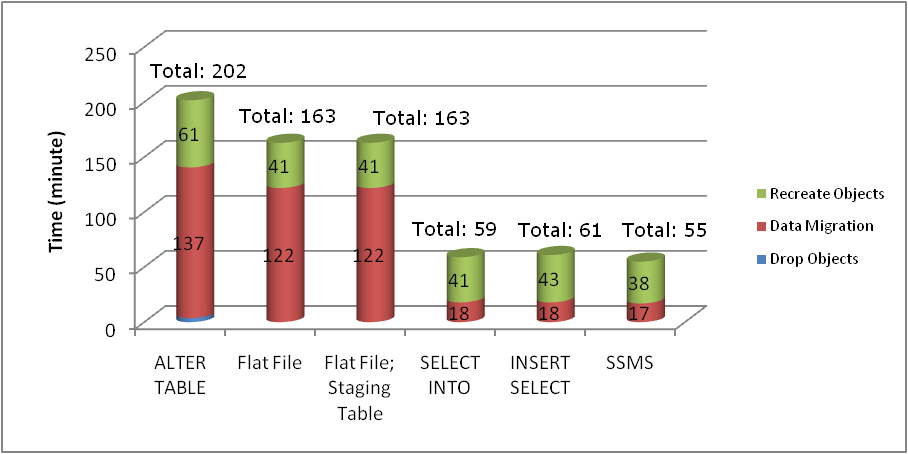
Following are the results from the tests described in the previous section.

###### Duration and Resource Utilization

The following table and graph show the duration of the migration (in minutes) for each scenario.

Table 4: Time for Migration (minute)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Scenario**  **Step** | **Test 1:**  **ALTER TABLE** | **Test 2-1:**  **Flat file** | **Test 2-2: Flat file; staging table** | **Test3:**  **SELECT INTO** | **Test 4:**  **INSERT SELECT** | **Test 5: SSMS[[3]](#footnote-4)** |
| **Drop objects** | 3.7 | <1 | <1 | <1 | <1 | <1 |
| **Data migration** | 136.8 | 121.5[[4]](#footnote-5) | 121.5[[5]](#footnote-6) | 17.9 | 18.1 | 17.2 |
| **Re-create objects[[6]](#footnote-7)** | 61.3 | 40.6 | 40.6 | 41.1 | 43.1 | 38.1 |
| ***Total*** | ***201.7*** | ***162.1*** | ***162.1*** | ***59.0*** | ***61.2*** | ***55.3*** |

Figure 1: Comparing the migration time

As shown in the results, using SSMS is the fastest way to migrate data from non-Unicode type to Unicode type. The internal operations of the SSMS migration are covered in a later section. The SSMS method is followed by SELECT INTO migration, then INSERT SELECT, flat file, and lastly, ALTER TABLE. Following are some observations on the results of this test:

* Flat file migration took more than 2.5 times longer than using SELECT INTO.
* The ALTER TABLE migration took more than three times longer than the SELECT INTO migration.
* The SELECT INTO migration and INSERT SELECT migration took almost the same time.
* For data migration (see the "Data migration" row in Table 4), the SELECT INTO migration and the SSMS migration took almost the same time as INSERT SELECT, followed by the flat file migration and finally, ALTER TABLE.
* Compared to the SELECT INTO process, the flat file process took about 6.5 times longer for data migration while the ALTER TABLE process took about 7.5 times longer.

The following tables and graphs show how much disk space is used by each migration. Table 5 shows space utilization for each table at the initial status and the intermediate status. Table 6 shows the table sizes at the final state of each migration. The space utilization at the intermediate status were captured just after migrating (loading) the data into the Unicode tables. There was no index on the tables at the intermediate state, so they are described as "Heap." Figure 2 shows the space used by the tables for the initial, intermediate, and final states. The scale is normalized by the size of the initial state. You can see the percentage of the data file size change in each status. For the SSMS migration, the table and graph show only the table sizes of the final state because it gets to the final state automatically without stopping at the intermediate state.

The results show that the final disk space used by the tables is the same in all migration scenarios, while disk space utilization of the ALTER TABLE migration at the intermediate state is different in each scenario; the disk space used by ALTER TABLE migration increased up by 60% of the initial disk space at the intermediate state while the disk space used by the flat file and INSERT INTO migrations increased 11% at the intermediate state. The final growth rate was about 37% of the initial state in all of the migration scenarios.

Table 5: Table size (KB) at initial and intermediate states

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Table** | **file** | **Initial**  **(KB)** | **Intermediate (Heap) State** | | | |
| **ALTER TABLE** | **Flat File** | **SELECT INTO** | **INSERT SELECT** |
| **CUSTOMER** | **Data** | 389,976 | 1,121,576 | 712,176 | 712,208 | 712,176 |
| **Index** | 728 | 56 | 56 | 56 | 56 |
| **LINEITEM** | **Data** | 14,872,120 | 27,233,616 | 19,557,400 | 19,557,416 | 19,557,400 |
| **Index** | 4,739,640 | 112 | 112 | 112 | 112 |
| **NATION** | **Data** | 8 | 8 | 8 | 16 | 8 |
| **Index** | 24 | 8 | 8 | 8 | 8 |
| **REGION** | **Data** | 8 | 8 | 8 | 16 | 8 |
| **Index** | 8 | 8 | 8 | 8 | 8 |
| **PART** | **Data** | 437,264 | 1,194,088 | 777,216 | 777,232 | 777,216 |
| **Index** | 800 | 56 | 56 | 56 | 56 |
| **SUPPLIER** | **Data** | 23,736 | 66,128 | 43,128 | 43,128 | 43,128 |
| **Index** | 1,856 | 56 | 56 | 56 | 56 |
| **PARTSUPP** | **Data** | 1,874,984 | 5,116,120 | 3,383,208 | 3,383,800 | 3,383,208 |
| **Index** | 134,920 | 56 | 56 | 56 | 56 |
| **ORDERS** | **Data** | 3,051,592 | 6,941,144 | 4,617,168 | 4,617,144 | 4,617,168 |
| **Index** | 642,576 | 56 | 56 | 56 | 56 |
| ***Total (KB)*** | | ***26,170,240*** | ***41,673,096*** | ***29,090,720*** | ***29,091,368*** | ***29,090,720*** |

Table 6: Table size (KB) at final state

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Table** | **file** | **Final State** | | | | |
| **ALTER TABLE** | **Flat File** | **SELECT INTO** | **INSERT SELECT** | **SSMS** |
| **CUSTOMER** | **Data** | 712,200 | 712,200 | 712,224 | 712,200 | 712,208 |
| **Index** | 1,216 | 1,216 | 1,376 | 1,216 | 1,376 |
| **LINEITEM** | **Data** | 20,667,568 | 20,667,288 | 20,667,376 | 20,667,352 | 20,667,600 |
| **Index** | 4,757,384 | 4,757,376 | 4,757,896 | 4,757,376 | 4,757,856 |
| **NATION** | **Data** | 8 | 8 | 8 | 8 | 8 |
| **Index** | 24 | 24 | 24 | 24 | 24 |
| **REGION** | **Data** | 8 | 8 | 8 | 8 | 8 |
| **Index** | 8 | 8 | 8 | 8 | 8 |
| **PART** | **Data** | 777,224 | 777,224 | 777,240 | 777,224 | 777,256 |
| **Index** | 1,320 | 1,320 | 1,480 | 1,320 | 1,488 |
| **SUPPLIER** | **Data** | 43,120 | 43,120 | 43,144 | 43,120 | 43,160 |
| **Index** | 1,832 | 1,832 | 2,160 | 1,832 | 2,192 |
| **PARTSUPP** | **Data** | 3,383,792 | 3,383,792 | 3,383,800 | 3,383,792 | 3,383,816 |
| **Index** | 137,976 | 137,976 | 138,312 | 137,976 | 138,320 |
| **ORDERS** | **Data** | 4,870,720 | 4,870,672 | 4,870,640 | 4,870,712 | 4,870,792 |
| **Index** | 648,104 | 648,104 | 648,376 | 648,104 | 648,376 |
| ***Total (KB)*** | | ***36,002,504*** | ***36,002,168*** | ***36,004,072*** | ***36,002,272*** | ***36,004,488*** |

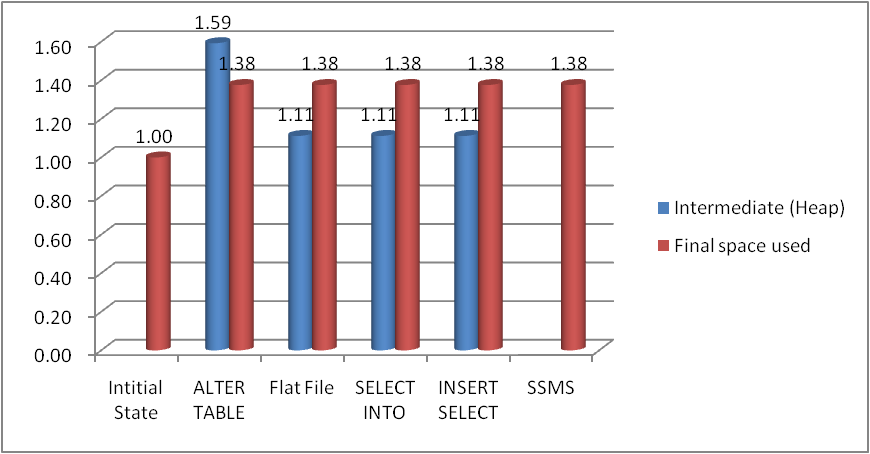
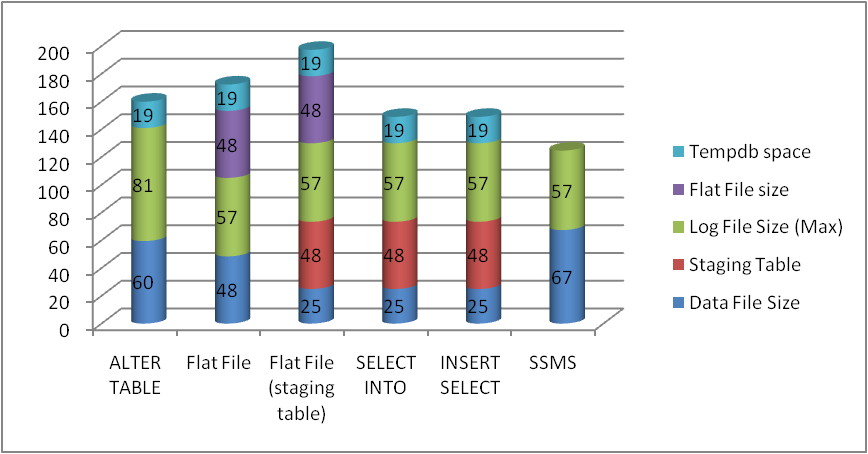
Figure 2: Disk space used by the tables (normalized by the initial state)

Table 7 and Figure 3 show the total disk space used for each migration. It includes data file size, log file size (maximum), staging table size, and flat file size. The data file size for the ALTER TABLE migration, flat file migration, and SSMS migration represents the maximum size of the total data pages. The data file size includes the total size of the heap (intermediate) tables and the size of the largest table in the final state. This is because SQL Server holds both the source and target table during index creation. On the other hand, the flat file migration (with staging tables), SELECT INTO migration, and INSERT SELECT migration show the original data page size as the data file size, while the disk space for staging tables includes the total size of the tables in the final state and the size of the largest heap table. In the case of SSMS migration, **tempdb** was not used for index creation. Instead of using **tempdb**, this method allocates the space for sorting data on the data files where the target data is located.

The test results show that SSMS migration uses the least amount of disk space, followed by SELECT INTO, then INSERT SELECT, ALTER TABLE migration, and lastly the flat file migration without staging tables. Flat file migration with staging tables requires the most disk space.

Table 7: Total Disk Space Used (GB)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **ALTER TABLE (GB)** | **Flat File  (GB)** | **Flat file with staging table (GB)** | **SELECT INTO (GB)** | **INSERT SELECT (GB)** | **SSMS (GB)** |
| **Data file size** | 59.5 | 48.4 | 25.0 | 25.0 | 25.0 | 67.5 |
| **Log file size** | 81.4 | 56.6 | 56.6 | 56.6 | 56.6 | 57.2 |
| **Staging table** | 0 | 0 | 48.4 | 48.4 | 48.4 | 0 |
| **Flat file size** | 0 | 48.3 | 48.3 | 0 | 0 | 0 |
| **temdb space** | 19.0 | 19.0 | 19.0 | 19.0 | 19.0 | 0 |
| ***Total*** | ***159.9*** | ***172.3*** | ***197.3*** | ***149.0*** | ***149.0*** | ***124.7*** |

Figure 3: Total disk space used (GB)

The results do not show any CPU bottleneck during each migration. Table 8 and Table 9 show disk performance during the data migration process (it does not include the operations for index re-creation). The SELECT INTO migration, the INSERT SELECT migration, and the SSMS migration show almost the same value in the average read/write activity on the data disk, while the SELECT INTO migration shows much lower access on the log disk than the INSERT SELECT and SSMS methods.

Table 8: Disk read (MB)/sec

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | **Disk Read MB/sec** | | | | |
| **Disk Usage** | | **ALTER TABLE** | **Flat File** | **SELECT INTO** | **INSERT SELECT** | **SSMS** |
| **Log** | **Average** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| **Maximum** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| **Data** | **Average** | 1.29 | 0.37 | 2.69 | 2.60 | 2.68 |
| **Maximum** | 13.86 | 3.07 | 2.99 | 5.71 | 5.79 |
| **Tempdb** | **Average** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| **Maximum** | 0.22 | 0.22 | 0.01 | 0.00 | 0.00 |
| **Flat File** | **Average** | 0.00 | 6.60 | 0.00 | 0.00 | 0.00 |
| **Maximum** | 0.00 | 11.53 | 0.00 | 0.00 | 0.00 |

Table 9: Disk write (MB)/sec

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | **Disk Write MB/sec** | | | | |
| **Disk Usage** | | **ALTER TABLE** | **Flat File** | **SELECT INTO** | **INSERT SELECT** | **SSMS** |
| **Log** | **Average** | 16.60 | 5.35 | 0.38 | 42.42 | 36.53 |
| **Maximum** | 32.07 | 13.15 | 4.50 | 57.37 | 67.36 |
| **Data** | **Average** | 1.95 | 0.54 | 3.76 | 3.19 | 2.47 |
| **Maximum** | 7.47 | 5.76 | 4.43 | 6.27 | 6.30 |
| **Tempdb** | **Average** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| **Maximum** | 0.21 | 0.21 | 0.02 | 0.01 | 0.00 |
| **Flat File** | **Average** | 0.00 | 6.64 | 0.00 | 0.00 | 0.00 |
| **Maximum** | 0.00 | 44.07 | 0.00 | 0.00 | 0.00 |

###### Inside Operations of SSMS Migration

The following statements were captured by SQL Server Profiler while we changed data types on one of the testing tables by using SSMS.

-- Create a temp table

CREATE TABLE dbo.Tmp\_NATION

(N\_NATIONKEY int NOT NULL,

N\_NAME nchar(25) NOT NULL,

N\_REGIONKEY int NOT NULL,

N\_COMMENT nvarchar(152) NOT NULL)

-- Load data into the temp table

IF EXISTS(SELECT \* FROM dbo.NATION)

EXEC('INSERT INTO dbo.Tmp\_NATION (N\_NATIONKEY, N\_NAME, N\_REGIONKEY,　N\_COMMENT)

SELECT N\_NATIONKEY, CONVERT(nchar(25), N\_NAME), N\_REGIONKEY, CONVERT(nvarchar(152), N\_COMMENT)

FROM dbo.NATION

WITH (HOLDLOCK TABLOCKX)')

-- Drop foreign key constraint

ALTER TABLE dbo.SUPPLIER DROP CONSTRAINT FK\_S\_NATIONKEY

-- Drop the original table

DROP TABLE dbo.NATION

-- Rename the temp table

EXECUTE sp\_rename N'dbo.Tmp\_NATION', N'NATION', 'OBJECT'

-- Create index

ALTER TABLE dbo.NATION ADD CONSTRAINT PK\_N\_NATIONKEY PRIMARY KEY CLUSTERED(N\_NATIONKEY) WITH( STATISTICS\_NORECOMPUTE = OFF, IGNORE\_DUP\_KEY = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON)

CREATE NONCLUSTERED INDEX N\_REGIONKEY\_IDX ON dbo.NATION(N\_REGIONKEY) WITH( PAD\_INDEX = OFF, FILLFACTOR = 100, STATISTICS\_NORECOMPUTE = OFF, IGNORE\_DUP\_KEY = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON)

-- Add constraint

ALTER TABLE dbo.SUPPLIER ADD CONSTRAINT FK\_S\_NATIONKEY FOREIGN KEY(S\_NATIONKEY) REFERENCES dbo.NATION(N\_NATIONKEY) ON UPDATE NO ACTION ON DELETE NO ACTION

The results show that changing the data type by using SSMS consists of the following steps:

1. Create a temporary table.
2. Load data from the original table into the temporary table by using INSERT SELECT.
3. Drop foreign key constraints when the original table has a key referred to by the other table.
4. Drop the original table.
5. Rename the temporary table to the original name.
6. Create table objects such as primary key and index.

###### Changing Collation

In this test, we used the following data, table schema, and collation.

create database JPN collate Japanese\_CI\_AS

use JPN

create table t1 (c1 char(2))

insert into t1 values ('あ')

Figure 4 shows the return value from the query SELECT \* FROM t1.

Figure 4: Returned value from SELECT \* FROM t1

We tested three migration scenarios by using the next steps, which are followed by the results.

ALTER TABLE

1. alter database JPN collate Latin1\_General\_CI\_AS
2. alter table t1 alter column c1 nchar(2) collate Latin1\_General\_CI\_AS

The following query clarifies the collation of the table

select object\_name(o.object\_id) [Table], c.name [Column]

, t.name [Data Type], c.max\_length [Length], c.collation\_name

from sys.columns c join sys.objects o on c.object\_id=o.object\_id

join sys.types t on c.user\_type\_id = t.user\_type\_id

where o.type = 'U'

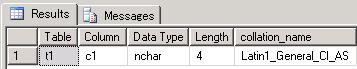
Figure 5: Collation of t1 after ALTER TABLE

Figure 6 shows that the data was migrated to the English collation table successfully.

SELECT \* FROM t1

Figure 6: Returned value from t1 after ALTER TABLE

We got the same results from the staging tables created in the following tests. Table 10 summarizes the results of the collation change test.

Flat File

1. bcp JPN.dbo.t1 out E:\t1.dat –w –Slocalhost –T (from command prompt)
2. alter database JPN collate Latin1\_General\_CI\_AS
3. use JPN
4. create table tmp\_t1 (c1 nchar(2))
5. bulk insert tmp\_t1 from ’E:\t1.dat’ with (DATAFILETYPE=’widechar’)
6. select \* from tmp\_t1

SELECT INTO

1. alter database JPN collate Latin1\_General\_CI\_AS
2. use JPN
3. select cast(c1 as nchar(2)) collate Latin1\_General\_CI\_AS as c1 into tmp\_t1 from t1
4. select \* from tmp\_t1

INSERT SELECT

1. alter database JPN collate Latin1\_General\_CI\_AS
2. use JPN
3. create table tmp\_t1 (c1 nchar(2))
4. insert into tmp\_t1 select \* from t1
5. select \* from tmp\_t1

Table 10: Summary of Collation Change Test

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Scenario | ALTER TABLE | Flat File | SELECT INTO | INSERT SELECT |
| Result | Success | Success | Success | Success |

##### Observations

This section compares migration methodologies according to the results described in the previous section.

###### Comparison of ALTER TABLE, Flat File, and INSERT SELECT Migrations

Considering that SSMS migration uses the same methodology as the INSERT SELECT migration, migration by using INSERT SELECT is best in terms of duration and disk space utilization. This section explores why the other migration methods require more time and disk space.

**ALTER TABLE**

The ALTER TABLE migration took the longest time and used the most disk space. On SQL Server 2005, altering columns by using the ALTER TABLE ALTER COLUMN statement does not actually drop the old column data on the data page. When it executes the statement, it adds a new column at the end of the table and leaves the old column, which is no longer usable, in its original position. This is the reason for the high disk space growth of the migration when using the ALTER TABLE statement. The original data is not cleared until the statement creates or rebuilds a clustered index on the table.

**Flat File**

For migration using flat files, if it is possible to export data in a table into multiple files, we can run multiple importing threads concurrently. In this case, we tested multiple concurrent BULK INSERT statements on the largest table (LINEITEM), which had a **datetime** type column used by the clustered index key (see [Appendix A](#_Appendix_A:)). Because the computer had eight CPU cores, we divided the data into eight files. The example export command lines are as follows:

BCP “select \* from LINEITEM where L\_SHIPDATE < '1992-12-28'” queryout LINEITEM01.dat –w –T

BCP “select \* from LINEITEM where L\_SHIPDATE >= '1992-12-28' and L\_SHIPDATE < '1993-10-25’” queryout LINEITEM02.dat –w –T

:

BCP “select \* from LINEITEM where L\_SHIPDATE >= '1997-12-5'” queryout LINEITEM08.dat –w –T

Table 11 shows the duration of converting LINEITEM table in two cases; one used a single thread and the other used eight threads for the bulk operations.

We see a performance improvement of about 37% when using eight threads, but it still took longer than INSERT SELECT migration (by about 42 minutes as shown in [Table 4](#Table4)).

Table 11: Duration (min) of the Migration for LINEITEM table (19GB)

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Single Thread (min) | Eight Thread (min) | % Difference |
| **bcp out** | 34.7 | 15.2 | -56.2 |
| BULK INSERT | 60.9 | 45.2 | -25.8 |
| Total | 95.6 | 60.4 | -36.8 |

###### Database Availability

When a data type migration requires a maintenance window, we want to know how much time is required for the operation. In this section, we look into database availability during each migration. During the migration using ALTER TABLE, the table specified by the statement was locked. In addition, as the objects, such as indexes and constraints, were dropped, the database was not available. On the other hand, using the staging tables for the migration allowed read access to the original tables until they were dropped. Therefore, the Flat File migration and INSERT SELECT migration require less database downtime. The biggest difference between the INSERT SELECT migration and the SSMS migration is that the SSMS migration requires the exclusive lock on the target table from beginning to end of the data migration, including index re-creation. Therefore, the target table goes offline during the SSMS migration, which results in a long database downtime.

Table 12 shows the availability of the tables during each operation included in the migrations.

Table 12: Database (table) availability during migration

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Scenario** | **Operation** | **Duration (min)** | **Lock mode (Source table)** | **Source table status** |
| ALTER TABLE | Drop index | 4 | Exclusive | Offline |
| ALTER TABLE (Transact‑SQL statement) | 137 | Schema modification | Offline |
| Index creation | 61 | Exclusive/Shared | Offline/Online (option***[[7]](#footnote-8)***) |
| Flat File | bcp out  (bcp utility) | 42 | Shared | Read only |
| Drop and recreate tables | <1 | Exclusive | Offline |
| BULK INSERT (Transact‑SQL statement) | 79 | Exclusive | Offline |
| Index creation | 40 | Exclusive/Shared | Offline/Online (option[[8]](#footnote-9)) |
| Flat File with Staging Table | bcp out  (bcp utility) | 42 | Shared | Read only |
| BULK INSERT (Transact-SQL statement) | 79 | None | Read only |
| Index Creation | 40 | None | Read only |
| Drop and rename | <1 | Exclusive | Offline |
| SELECT INTO | SELECT INTO (Transact‑SQL statement) | 18 | Shared | Read Only |
| Index creation | 41 | None | Read only |
| Drop and rename | <1 | Exclusive | Offline |
| INSERT SELECT | INSERT SELECT (Transact‑SQL statement) | 18 | Shared | Read only |
| Index creation | 43 | None | Read only |
| Drop and rename | <1 | Exclusive | Offline |
| SSMS | Alter Table (GUI) | 55 | Exclusive | Offline |

If using staging tables (flat file migration, SELECT INTO migration, and INSERT SELECT migration), index creation on the new tables (the staging tables) can be done without any impact on the original table. In addition, flat file migration does not affect the source tables when loading data into the new tables, while SELECT INTO migration and INSERT SELECT migration must read data from the original tables. To maintain data consistency between the original table and the new table, the original table should be accessed in read-only mode during the migration. If there is no need to alter the data in the original tables, the outage of the tables in these migrations occurs only when dropping and renaming the tables. In this test, the duration of those operations was less than one minute in total. In comparison, ALTER TABLE, flat file without staging tables, and SSMS migration required much more downtime for the migrations.

Table 13 shows the summary of outage of the database. The SELECT INTO migration provides the highest availability of the database, followed by the INSERT SELECT migration. However, the availability of the database system depends on which tables (data) are used in the operations and what the operations do. For example, if the read-only status allowed for the operation, the flat file with staging table migration also provides the highest availability.

**Table 13: Summary of database availability**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **ALTER TABLE** | **Flat file** | **Flat file with staging table** | **SELECT INTO** | **INSERT SELECT** | **SSMS** |
| **Offline time (min)** | 141 | 79 | <1 | <1 | <1 | 55 |
| **Read-only time (min)** | - | 42 | 161 | 59 | 61 | - |

###### Comparison of INSERT SELECT and SSMS Migration

We confirmed that altering tables by using Table Designer in SSMS executes the same steps as when we tested by using the INSERT SELECT statement. SSMS executes CREATE TABLE, INSERT SELECT, DROP TABLE, RENAME, and CREATE INDEX in the background. As it executes these operations automatically, this is the easiest way to migrate from non-Unicode to Unicode data type. However, the automatic process limits the flexibility of operations such as maintaining both original tables and staging tables, setting options for index creation such as SORT\_IN\_TEMPDB, and so on. So, this method may cause a database availability problem. As it does not use **tempdb** during index creation, you must prepare enough space on the filegroup of the table for the operation when you want to use SSMS migration. For a table that has no index, this comparison does not apply. On the other hand, INSERT SELECT has the flexibility of allowing creating table in a new or different filegroup if this is required.

The other concern is the possibility of human error in a GUI operation. The larger the database system is, the more difficult the operations become.

#### Recommendations

After conducting the tests by using sample tables and sample data, we offer the following general recommendations. Note that these are general recommendations that apply to the tests used in this study. They may not apply to your particular environment and therefore we encourage you to use them as base recommendation.

* Prepare an appropriate maintenance window for the migration and make sure the system has a valid backup.
* Use SELECT INTO migration when you must complete the migration as quickly as possible and you can ignore the restrictions of the migration.
* Use SSMS migration when database availability or index creation is not the first priority and you can ignore the human error potential in GUI operations.
* Use INSERT SELECT migration when you need the flexibility for the filegroup or the table partitioning of the migrated tables.
* Use SELECT INTO or INSERT SELECT migration when you must complete the migration with the least possible outage of the database. These migration methods provide read-only access to the target tables during the data loading.
* Make sure the system has sufficient disk space for staging tables, indexes, logging, and **tempdb** when you use SELECT INTO or INSERT SELECT migration. Additional disk space is required for flat files when you use flat file migration.
* Use flat file migration when you must verify and/or clean the data before the migration or when you want to operate the data export and import separately.
* Use the same number of flat files with CPU (core) for exporting and importing data if you use flat file migration.
* Make sure the target tables have the correct schema (data type). When changing the collation and code page, if the table has multi-byte character data in a non-Unicode data type column, the target table (of a new collation) must store the data in a Unicode data type column.

#### Conclusion

We tested a number of methods for migrating database systems from non-Unicode data type to Unicode data type. Based on the results of our testing, we conclude that the best practice recommendation for this migration is to use staging tables and to load data by using an INSERT SELECT statement, followed by re-creating indexes and other table objects.

**References**

* [SQL Server 2005 Unicode Considerations](http://blogs.msdn.com/mssqlisv/archive/2006/07/07/659374.aspx)
* [The Impacts of Collation Change and Going from Non-Unicode to Unicode](http://download.microsoft.com/download/d/9/4/d948f981-926e-40fa-a026-5bfcf076d9b9/SQL_bestpract_CollationChange.docx)

For more information:

[SQL Server Web site](http://www.microsoft.com/sql/default.mspx)

[SQL Server TechCenter](http://technet.microsoft.com/en-us/sqlserver/default.aspx)

[SQL Server DevCenter](http://msdn2.microsoft.com/en-us/sqlserver/default.aspx)

For more information:

[http://www.microsoft.com/sql/](http://www.microsoft.com/exchange/)!href([http://www.microsoft.com/sql/](http://www.microsoft.com/exchange/))

Did this paper help you? Please give us your feedback. On a scale of 1 (poor) to 5 (excellent), [how would you rate this paper?](mailto:sqlfback@microsoft.com?subject=Feedback:%20%3cinsert%20paper%20Title%3e)!href(mailto: sqlfback@microsoft.com?subject=Feedback: Developing Custom Components in SQL Server Integration Services for Project REAL)

#### Appendix A

Table Schema

* CREATE TABLE REGION

(R\_REGIONKEY int not null,

R\_NAME char(25) not null,

R\_COMMENT varchar(152) not null)

* CREATE TABLE NATION

(N\_NATIONKEY int not null,

N\_NAME char(25) not null,

N\_REGIONKEY int not null,

N\_COMMENT varchar(152) not null)

* CREATE TABLE PART

(P\_PARTKEY int not null,

P\_TYPE varchar(25) not null,

P\_SIZE int not null,

P\_BRAND char(10) not null,

P\_NAME varchar(55) not null,

P\_CONTAINER char(10) not null,

P\_MFGR char(25) not null,

P\_RETAILPRICE float not null,

P\_COMMENT varchar(23) not null)

* CREATE TABLE SUPPLIER

(S\_SUPPKEY int not null,

S\_NATIONKEY int not null,

S\_NAME char(25) not null,

S\_ADDRESS varchar(40) not null,

S\_PHONE char(15) not null,

S\_ACCTBAL float not null,

S\_COMMENT varchar(101) not null)

* PARTSUPP

(PS\_PARTKEY int not null,

PS\_SUPPKEY int not null,

PS\_SUPPLYCOST float not null,

PS\_AVAILQTY int not null,

PS\_COMMENT varchar(199) not null)

* CREATE TABLE CUSTOMER

(C\_CUSTKEY int not null,

C\_MKTSEGMENT char(10) not null,

C\_NATIONKEY int not null,

C\_NAME varchar(25) not null,

C\_ADDRESS varchar(40) not null,

C\_PHONE char(15) not null,

C\_ACCTBAL float not null,

C\_COMMENT varchar(117) not null)

* CREATE TABLE ORDERS

(O\_ORDERDATE datetime not null,

O\_ORDERKEY bigint not null,

O\_CUSTKEY int not null,

O\_ORDERPRIORITY char(15) not null,

O\_SHIPPRIORITY int not null,

O\_CLERK char(15) not null,

O\_ORDERSTATUS char(1) not null,

O\_TOTALPRICE float not null,

O\_COMMENT varchar(79) not null)

* CREATE TABLE LINEITEM

(L\_SHIPDATE datetime not null,

L\_ORDERKEY bigint not null,

L\_DISCOUNT float not null,

L\_EXTENDEDPRICE float not null,

L\_SUPPKEY int not null,

L\_QUANTITY float not null,

L\_RETURNFLAG char(1) not null,

L\_PARTKEY int not null,

L\_LINESTATUS char(1) not null,

L\_TAX float not null,

L\_COMMITDATE datetime not null,

L\_RECEIPTDATE datetime not null,

L\_SHIPMODE char(10) not null,

L\_LINENUMBER int not null,

L\_SHIPINSTRUCT char(25) not null,

L\_COMMENT varchar(44) not null)

Constraints and Indexes

* ALTER TABLE NATION ADD CONSTRAINTS PK\_N\_NATIONKEY PRIMARY KEY (N\_NATIONKEY)
* ALTER TABLE REGION ADD CONSTRAINTS PK\_R\_REGIONKEY PRIMARY KEY (R\_REGIONKEY)
* ALTER TABLE PART ADD CONSTRAINTS PK\_P\_PARTKEY PRIMARY KEY (P\_PARTKEY)
* ALTER TABLE SUPPLIER ADD CONSTRAINTS PK\_S\_SUPPKEY PRIMARY KEY (S\_SUPPKEY)
* ALTER TABLE CUSTOMER ADD CONSTRAINTS PK\_C\_CUSTKEY PRIMARY KEY (C\_CUSTKEY)
* ALTER TABLE PARTSUPP ADD CONSTRAINTS PK\_PS\_PARTKEY\_PS\_SUPPKEY PRIMARY KEY (PS\_PARTKEY, PS\_SUPPKEY)
* ALTER TABLE ORDERS ADD CONSTRAINTS PK\_O\_ORDERKEY PRIMARY KEY (O\_ORDERKEY)
* CREATE INDEX S\_NATIONKEY\_IDX ON SUPPLIER(S\_NATIONKEY)
* CREATE INDEX PS\_SUPPKEY\_IDX ON PARTSUPP(PS\_SUPPKEY)
* CREATE CLUSTERED INDEX O\_ORDERDATE\_CLUIDX ON ORDERS(O\_ORDERDATE)
* CREATE CLUSTERED INDEX L\_SHIPDATE\_CLUIDX ON LINEITEM(L\_SHIPDATE)
* CREATE INDEX L\_ORDERKEY\_IDX ON LINEITEM(L\_ORDERKEY)
* CREATE INDEX L\_PARTKEY\_IDX ON LINEITEM(L\_PARTKEY)
* ALTER TABLE SUPPLIER ADD CONSTRAINT FK\_S\_NATIONKEY FOREIGN KEY (S\_NATIONKEY) REFERENCES NATION(N\_NATIONKEY)
* ALTER TABLE PARTSUPP ADD CONSTRAINT FK\_PS\_PARTKEY FOREIGN KEY (PS\_PARTKEY) REFERENCES PART(P\_PARTKEY)
* ALTER TABLE PARTSUPP ADD CONSTRAINT FK\_PS\_SUPPKEY FOREIGN KEY (PS\_SUPPKEY) REFERENCES SUPPLIER(S\_SUPPKEY)
* ALTER TABLE CUSTOMER ADD CONSTRAINT FK\_C\_NATIONKEY FOREIGN KEY (C\_NATIONKEY) REFERENCES NATION(N\_NATIONKEY)
* ALTER TABLE OREDERS ADD CONSTRAINT FK\_O\_CUSTKEY FOREIGN KEY (O\_CUSTKEY) REFERENCES CUSTOMER(C\_CUSTKEY)
* ALTER TABLE NATION ADD CONSTRAINT FK\_N\_REGIONKEY FOREIGN KEY (N\_REGIONKEY) REFERENCES REGION(R\_REGIONKEY)
* ALTER TABLE LINEITEM ADD CONSTRAINT FK\_L\_ORDERKEY FOREIGN KEY (L\_ORDERKEY) REFERENCES ORDERS(O\_ORDERKEY)
* ALTER TABLE LINEITEM ADD CONSTRAINT FK\_L\_PARTKEY FOREIGN KEY (L\_PARTKEY) REFERENCES PART(P\_PARTKEY)
* ALTER TABLE LINEITEM ADD CONSTRAINT FK\_L\_SUPPKEY FOREIGN KEY (L\_SUPPKEY) REFERENCES SUPPLIER(S\_SUPPKEY)
* ALTER TABLE LINEITEM ADD CONSTRAINT FK\_L\_PARTKEY\_SUPPKEY FOREIGN KEY (L\_PARTKEY, L\_SUPPKEY) REFERENCES PARTSUPP(PS\_PARTKEY, PS\_SUPPKEY)

#### Appendix B

**Test environment information**

Database server CPU: 2-way (Quad core; eight cores), 2.33 GHz

Database server memory: 16-GB RAM

Operating System: The 64-bit edition of Windows Server® 2003 Enterprise Japanese (SP2)

SQL Server: SQL Server 2005 Enterprise Edition for x64 (SP2)

1. Use NVARCHAR(MAX) instead of NTEXT because NTEXT will be deprecated in future. [↑](#footnote-ref-2)
2. The **bcp** utility also provides the **–N** option as an alternative to **-w**. However, this option cannot be used for this migration because the option limits **bcp** data to databases that have same character sets and sort order. [↑](#footnote-ref-3)
3. Does not include think time, key time, and so on caused by manual operation. [↑](#footnote-ref-4)
4. Includes the time of **bcp out** and BULK INSERT. [↑](#footnote-ref-5)
5. Includes the time of **bcp out** and BULK INSERT. [↑](#footnote-ref-6)
6. Includes the time of dropping the old tables and renaming the staging tables. [↑](#footnote-ref-7)
7. SQL Server 2005 enables you to create the index online by setting the option ONLINE = ON in the clause in a CREATE INDEX Transact‑SQL statement. [↑](#footnote-ref-8)
8. SQL Server 2005 enables you to create the index online by setting the option ONLINE = ON in the clause in a CREATE INDEX Transact‑SQL statement. [↑](#footnote-ref-9)