

# Reducing Database Size by Using Vardecimal Storage Format

SQL Server Technical Article

Writers: Sunil Agarwal and Hermann Daeubler

Technical Reviewers: Paul Randal and Kangrong Yan

Published: May 2007

Applies To: SQL Server 2005 SP2

Summary: This white paper describes the new storage format for storing decimal and numeric data in SQL Server 2005 SP2. Details include when and how to enable vardecimal storage format, restrictions, space savings, and its impact on the performance of your workload.

### 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.

 2007 Microsoft Corporation. All rights reserved.

Microsoft is a registered trademark of Microsoft Corporation in the United States and/or other countries.

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

### Table of Contents

[Introduction 1](#_Toc167472812)

[The Problem 1](#_Toc167472813)

[The Solution: Vardecimal Storage Format 2](#_Toc167472814)

[Data Storage Format 2](#_Toc167472815)

[Estimating Space Savings 4](#_Toc167472816)

[Enabling Vardecimal Storage Format 6](#_Toc167472817)

[Restrictions 8](#_Toc167472818)

[Disabling Vardecimal Storage Format 11](#_Toc167472819)

[Impact on Database Migration 12](#_Toc167472820)

[Impact on Log Shipping 14](#_Toc167472821)

[Impact on Database Mirroring 14](#_Toc167472822)

[Impact on Replication 15](#_Toc167472823)

[Performance 15](#_Toc167472824)

[Table Scans 15](#_Toc167472825)

[Variable-Length Row and Forward Pointers 15](#_Toc167472826)

[CPU-Bound Scans 16](#_Toc167472827)

[OLTP-Like Workload 16](#_Toc167472828)

[Mass Updates, Inserts, and Deletes 17](#_Toc167472829)

[Production Workload SAP BI 17](#_Toc167472830)

[Production Workload Microsoft Internal Application 17](#_Toc167472831)

[Conclusion 18](#_Toc167472832)

#### Introduction

The vardecimal storage format became available starting with Microsoft® SQL Server™ 2005 Service Pack 2 (SP2) in the Enterprise Edition. This storage format can be enabled at a table-level granularity. When enabled, SQL Server stores decimal and numeric data in the variable portion of the row instead the fixed portion. You can use vardecimal storage format to reduce the size of your database if you have tables with decimal and numeric data types. How much space you save depends on the number of decimal or numeric columns, the data distributions, and the size of the table(s). Since this is a new storage format but not a new data type, you do not need to apply any application changes to benefit from it. This white paper covers the impact of enabling vardecimal storage format on different workloads.

#### The Problem

SQL Server defines decimal and numeric data types as fixed-length data types. These data types have a format of (*p*, *s*) where *p* is the precision (number of decimal digits) and *s* is the scale representing the number of digits after the decimal. These two data types are functionally equivalent. Depending on the precision (1 to 38), a decimal value can take anywhere from 5 bytes to 17 bytes of storage space as shown in the following table.

| **Precision** | **Storage Bytes** |
| --- | --- |
| 1-9 | 5 |
| 10-19 | 9 |
| 20-28 | 13 |
| 29-38 | 17 |

Storage requirements become especially challenging when you have small decimal values for a column that has a high precision requirement as this wastes many bytes storing the record. This is magnified if you have a large table(s) with many decimal/numeric columns. The size of your database can become much larger than necessary (you need a bigger database to store the same information). This is similar to the existing storage format for character data; **char** (17) versus **varchar** (17). If most of your character data is 1 or 2 characters but the maximum value is 17 characters, you can reduce the size of a row by declaring the column to be of data type **varchar** (17) instead of **char** (17). For example, to store one ASCII character in variable-length format, SQL Server needs three bytes (two bytes of fixed overhead for the column offset in the offset table and one byte to store one character), which is more than 80% less space than the 17 bytes required to store **char** (17).

The size of the database has performance and cost implications beyond increased requirements for disk space. Following are some less obvious costs associated with the size of a database:

* In a typical IT group, the production data may include multiple copies of the database for fault tolerance and availability purposes. For example, replicating a database proportionally increases the cost of the hardware.
* The size of backups are larger. It takes more time to back up your database, more disk space to store backups, and more time to restore the database.
* Depending on the data access pattern of your workload, a server that is running SQL Server may incur additional memory pressure and physical I/Os. This ultimately impacts transaction throughput and the response time of your workload.
* A larger database size means more logging, which can impact the performance of transactional replication, log shipping, and database mirroring, for example.

A smaller database may be preferable over a larger one. But how can the database size be reduced? And what is the impact of a smaller database on the workload?

#### The Solution: Vardecimal Storage Format

Starting with SQL Server 2005 SP2, you can use a new storage format to store decimal/numeric values in a variable-length storage format. By eliminating leading and trailing zeros and by storing only the minimum number of digits that are required, SQL Server stores decimal/numeric data efficiently. Using vardecimal storage format can result in significant space savings, depending on the schema of the table, number of rows, and the data distribution of decimal and numeric data.

The vardecimal storage format is actually an alternate storage format for storing decimal and numeric data and is NOT a new data type. This means that application changes are not required to take advantage of this new storage format. You can choose to enable this storage format at a table granularity by using the **sp\_tableoption**stored procedure. When a table is enabled to use vardecimal storage format, all decimal and numeric columns in the table are stored by using the new storage format. This includes indexes that have decimal or numeric columns as keys or as included columns. The following sections provide details on how decimal or numeric data is stored in the new storage format, how to estimate space savings before enabling this option on a table, vardecimal storage format limitations, and the impact of vardecimal storage on CPU utilization and I/Os on your workload.

##### Data Storage Format

The vardecimal storage format consists of three parts to store decimal or numeric values: sign (1 bit), exponent (7 bits), and mantissa (1–19 bytes).

The decimal or numeric value in vardecimal storage format is computed as: (sign) mantissa \* 10exponent. The bits and bytes of the three parts of the vardecimal storage format are interpreted as follows:

* **Sign** **bit**: 1 represents positive, and 0 represents negative.
* **Exponent**: Range from -38 to 37. The exponent is biased by 64, so the value stored is 26 to 101 (with 26 representing -38 and 101 representing +37). For example, for -16.1, the exponent value is 1. After biasing, it becomes 65; that is, 100 0001. Similarly, the sign/exponent byte for -161.0 is 0100 0010.
* **Mantissa**: SQL Server stores the integer value in mantissa; the decimal point is assumed to be after the first digit. The mantissa consists of multiple 10-bit chunks. Every 10 bits represents three decimal digits. The order is from the most significant chunks to the least significant chunks. Note that since it uses 10-bit chunks, the total number of bits used might not be a multiple of 8 (bits per byte). In that case, SQL Server pads the bit stream to byte boundary, potentially wasting a maximum of 6 bits.

For example, if the number 12.3 is stored in a numeric (18, 4) column, the value that is stored is actually 123000. With the column metadata, SQL Server knows where to put the decimal point according to its scale. This aligns all the data to the same scale, making it possible to perform a comparison of the data without adjusting scale.

The vardecimal storage format will work on the value 123000, with a column type of numeric (18, 4). First, SQL Server converts it to scientific notion; that is, 1.23 \* 105. Since the column scale is 4, it subtracts 4 from the exponent, and stores 1 as an exponent, and 1.23 as mantissa. Since in vardecimal storage format, the decimal point is always after the first digit, the decimal point is ignored, and treated only as an integer value 123. Note that SQL Server truncates the trailing zeros when storing this value.

Following is the actual byte representation of the numeric value 12.3 in vardecimal storage format:

First byte:   
Bit 7: 1 (positive)  
Bit 0-6: 1000001; that is, 1 (exponent)

Mantissa (2 bytes): 0001111011000000. The first 10 bits represent 123; the remaining 6 bits are just zero-padding to achieve the full byte length.

Full representation: 10000001 00011110 11000000

So, in vardecimal storage format, the value 12.3 uses a total of 3 bytes of storage. In comparison, the same value takes 9 bytes in SQL Server 2005 or earlier versions. The savings are even greater if the column type is (38,4); this requires 3 bytes under the new format but would take 17 bytes in SQL Server 2005 or earlier. In this example, the row size reduction is approximately 85%.

The storage bytes for vardecimal storage format are determined only by the number of digits, independent of the precision that is declared for the column. A column that is declared with a higher precision results in an even greater reduction of row size when using vardecimal storage format. Note that for the value of 0, the storage format is 0 bytes (a zero value does not have an exponent or mantissa). Thus, the overhead is only 2 bytes (used to store column offset in the offset table). The same applies to NULL values. NULL values require only 2 bytes and those are for storing the column offset in the offset table.

The following table lists the number of bytes stored when using vardecimal storage format and all digits in the precision are non-zero (worst-case scenario). In comparison, fixed-decimal storage format does not depend on the actual value to determine the number of bytes to store; it uses the column max precision. Note that in the worst-case scenario (when the number of digits are the same as the precision declared and are non-zero), vardecimal storage format takes more space than fixed-decimal storage format in almost all cases. If the decimal or numeric column is the only variable-length column in the row, an additional overhead of 2 bytes (the number of variable length columns) is incurred, as this will become the first variable-length column. This overhead is not accounted for in the following table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Column precision** | **Original fixed decimal size (bytes)** | **Maximum vardecimal data area (bytes)** | **Overhead to store offset (bytes)** | **Maximum vardecimal storage used (bytes)** |
| 1-3 | 5 | 3 | 2 | 5 |
| 4-6 | 5 | 4 | 2 | 6 |
| 7-9 | 5 | 5 | 2 | 7 |
| 10-12 | 9 | 6 | 2 | 8 |
| 13-15 | 9 | 8 | 2 | 10 |
| 16-18 | 9 | 9 | 2 | 11 |
| 19 | 9 | 10 | 2 | 12 |
| 20-21 | 13 | 10 | 2 | 12 |
| 22-24 | 13 | 11 | 2 | 13 |
| 25-27 | 13 | 13 | 2 | 15 |
| 28 | 13 | 14 | 2 | 16 |
| 29-30 | 17 | 14 | 2 | 16 |
| 31-33 | 17 | 15 | 2 | 17 |
| 34-36 | 17 | 16 | 2 | 18 |
| 37-38 | 17 | 18 | 2 | 20 |

As the table shows, storing data in vardecimal storage format does not necessarily mean that the storage size of a value will be smaller. The storage size depends on both the declared precision of the column and the value. For example, if you have a decimal column declared with precision 18 and the common value for this column has only 3 digits, SQL Server uses only 5 bytes, which is almost 50% less than the 9 bytes used by the fixed format. But, if the common value for the column has 18 digits, SQL Server uses 11 bytes, which is 20% more than the corresponding size in the fixed-length format.

##### Estimating Space Savings

Before you enable the vardecimal storage format, you may want to know the potential reduction in the size of the table. Clearly, if a table has no decimal/numeric columns, there will be no savings. Even if a table has decimal/numeric column types, there is no guarantee that the size of the table will be reduced if you enable vardecimal storage format. Again, this is similar to using **varchar** (17) versus **char** (17). For example, if all the values in the column have 17 characters, the average row length will be larger with **varchar** (17), because it is stored in the variable portion of the record structure. Recall that you need 2 bytes to store the column offset in the offset table for the variable-length column. Also, if **varchar** (17) is the only variable-length column in the table, an overhead of 2 bytes to store number of variable length columns in the row is incurred. In this worst-case scenario, declaring the column type as **varchar** (17) may cost you 4 bytes more for each row than a **char** (17) column type.

SQL Server 2005 SP2 provides a stored procedure for estimating the ‘reduction in row size’ for vardecimal storage format. This stored procedure runs single-threaded and samples only a small set of rows from the target table to estimate the space savings when vardecimal storage format is enabled. You do not need to enable your database to vardecimal storage format to estimate the space savings. Based on our testing and customer feedback, the savings estimated by the stored procedure are close to the actual savings.

The following example illustrates the reduction in row size for two tables that have same scheme but different data, t\_bestcase being the best case and t\_worstcase being the worst case (where each decimal value has max 38 digits as allowed by the declared precision).

CREATE TABLE t\_bestcase (c1 int, c2 decimal(10,2),

c3 decimal (38,2), c4 varchar(10))

GO

CREATE TABLE t\_worstcase (c1 int, c2 decimal(10,2),

c3 decimal (38,2), c4 varchar(10))

GO

-- insert rows into these tables.

DECLARE @i int

SELECT @i = 0

WHILE (@i < 1000)

BEGIN

INSERT INTO t\_bestcase values (1, 0.0,0.0, 'hello')

INSERT INTO t\_worsecase values

(1,12345678.99,123456789012345678901234567890123499.99, 'hello')

set @i = @i + 1

END

Now let us find the potential space savings for each of these tables.

-- -- This is the best case

EXEC sys.sp\_estimated\_rowsize\_reduction\_for\_vardecimal 't\_bestcase'

Here is the output. Note that in this case you can reduce the size of the row by almost 50%. If you have more decimal/numeric columns, the savings are proportionally larger.

avg\_rowlen\_fixed\_format avg\_rowlen\_vardecimal\_format row\_count

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

46.00 24.00 1000

-- This is worst case.

EXEC sys.sp\_estimated\_rowsize\_reduction\_for\_vardecimal 't\_worstcase'

Here is the output. Note that in this case, the size of the average row actually increased in size.

avg\_rowlen\_fixed\_format avg\_rowlen\_vardecimal\_format row\_count

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

46.00 48.00 1000

Before you enable a table for vardecimal storage format, it is recommended that you run this stored procedure to estimate space savings. You may wonder why it is not always best to gain any storage saving, even a small amount. This is because the space savings come at the cost of additional CPU cycles. With the new storage format, SQL Server must first convert the fixed-length storage format into the variable-length storage format, and then store it in the variable portion of the row during INSERT and UPDATE operations. A similar conversion cost is paid when a new decimal/numeric value is read during a SELECT operation. Performance tradeoffs are discussed in a later section.

##### Enabling Vardecimal Storage Format

Enabling vardecimal storage format on a table is a two-step process.

1.  Enable the database for vardecimal storage format.

To enable the database for vardecimal storage format, use the stored procedure **sp\_db\_**vardecimal**\_storage\_format***.* The command is as follows:

EXEC sp\_db\_vardecimal\_storage\_format '<database-name>', 'ON'

**Note**   If there is another active session in the database, you will see the following error message. Close other concurrent sessions before you execute the command.

"Msg 5061, Level 16, State 1, Line 1  
ALTER DATABASE failed because a lock could not be placed on database   
'compression'. Try again later.  
sp\_db\_vardecimal\_storage\_format statement failed."

When the command is executed, SQL Server internally bumps the database version number, but no tables are enabled for vardecimal storage format. You must enable vardecimal storage format on each table separately. The database version must be bumped to indicate that the data in this database can potentially have a different storage format (the vardecimal storage format). This prevents attaching a vardecimal-enabled database to earlier versions of SQL Server 2005, as those versions are unable to interpret the new storage format. You can only enable vardecimal storage format on user databases; not on system databases such as **master** and **tempdb**. One exception to this is the distribution database that is used for replication. SQL Server does not allow enabling vardecimal storage format on a distribution database.

To find out which databases are enabled for vardecimal storage format, you can use the following command:

EXEC sp\_db\_vardecimal\_storage\_format

The output of this stored procedure is:

Database Name Vardecimal State

Master OFF

Tempdb OFF

Model OFF

Msdb OFF

<userdb-1> ON

<userdb-2> OFF

...

Note that the vardecimal state for system databases such as **master** and **tempdb** is always OFF. You cannot enable these databases for vardecimal storage format.

You can use the following command to see if a specific database has been enabled for vardecimal storage format:

EXEC sp\_db\_vardecimal\_storage\_format <database-name>

The output of this stored procedure is:

Database Name Vardecimal State

<database—name> ON

Changing the database vardecimal storage format state requires an exclusive lock on the database during the entire process. This command will fail if there are active users in the database because this prevents the exclusive lock on the database. Note that this command waits only a short time to acquire an exclusive lock before it times out. This short wait is defined internally by SQL Server and is not configurable.

2.  Determine which tables to enable for vardecimal storage format.

Choose one or more tables to enable with vardecimal storage format but be aware of the additional CPU overhead. (To determine whether to use vardecimal storage format, estimate the potential disk savings by using the stored procedure described in [Estimating Space Savings](#_Estimating_the_Space).)

sp\_tableoption '<table-name>', 'vardecimal storage format', 1

This command, potentially an expensive one (on the same order as creating an index), converts all the rows in the table that contain columns of type decimal/numeric to vardecimal storage format. Similar to creating or rebuilding indexes, enabling vardecimal storage format can generate log records depending on the recovery model of your database. Converting to vardecimal storage format is not an ONLINE operation. During conversion, the table is locked exclusively by using SCH-M lock and is not available for concurrent operations. If a table does not have a clustered index (it is a heap), an index row on the leaf page in a nonclustered index refers to the data rows by using a RID (a combination of file-ID, page-ID, and the index of the row offset array on the data page). When a table that does not have a clustered index is enabled for vardecimal storage format, it is rebuilt, thereby causing the RIDs of all data rows to change. It forces a rebuild of all nonclustered indexes as well. However, if a table has a clustered index, only the nonclustered indexes containing a decimal or numeric column as a key or included column need to be rebuilt. You can use the following command to find out which tables are enabled for vardecimal storage format in your database:

SELECT name, object\_id, type\_desc

FROM sys.objects

WHERE objectproperty(object\_id,

N'TableHasVarDecimalStorageFormat') = 1

Following is an example output showing that two user tables are enabled with the new storage option:

name object\_id type\_desc

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

t\_decimal1 34102012 USER\_TABLE

t\_decimal2 1074102867 USER\_TABLE

You can also query a specific table to see if it has been enabled for vardecimal storage format. Here is one example:

SELECT objectproperty(object\_id('<table-name>’),

'TableHasVarDecimalStorageFormat')

This query returns 1 if the specified table has been enabled for vardecimal storage format.

###### Restrictions

When vardecimal storage format is enabled, SQL Server must provide two guarantees. First, an update on decimal and numeric columns should never fail as a result of the row size limitation of 8,060 bytes. This is because, for applications, decimal and numeric data types are fixed-length data types, so it is assumed that the space to store all possible values of decimal and numeric type with declared precision has already been created. Of course, an update of decimal/numeric data might fail if it violates user-defined constraints but it should never fail as a result of the row size exceeding the 8,060 limit after the update.

Second, disabling vardecimal storage format must never fail as the result of row size limitations. That is, there must never be a case where the row size with vardecimal storage format enabled is less than or equal to 8,060, but the row cannot fit in 8,060 bytes when the vardecimal storage format is disabled. Of course, disabling vardecimal storage format can fail for other reasons, such as when the server does not have the additional disk space required when indexes are created or rebuilt.

The following code is one example that illustrates this point:

-- create a table that has row size that can potentially exceed 8060 limit

CREATE TABLE boundary (c1 char(8000), c2 char(20),

c3 varchar(23), c4 decimal(38,2))

When you create this table, it gives you the following warning:

"Warning: The table "boundary" has been created, but its maximum row size exceeds the allowed maximum of 8060 bytes. INSERT or UPDATE to this table will fail if the resulting row exceeds the size limit."

However if you create the following table:

CREATE TABLE t\_boundary (c1 char(8000), c2 char(200),

c3 varchar(23), c4 decimal(38,2))

It will fail with the following error:

"Msg 1701, Level 16, State 1, Line 1

Creating or altering table 't\_boundary' failed because the minimum row size would be 8224, including 7 bytes of internal overhead. This exceeds the maximum allowable table row size of 8060 bytes."

CREATE TABLE fails in the latter case because the minimum row length exceeds 8,060. In the former case, the minimum row length is less than or equal to 8,060 but not to the maximum length. This means that some, but not all, rows can be inserted or updated successfully. For example, the following row insert will succeed:

-- this is the max allowed value

INSERT INTO boundary values ('a', 'b', replicate ('1', 12), 0.0)

But the following update will fail because with this update, the row length exceeds 8,060-byte limit:

UPDATE boundary SET c3 = replicate('1', 13)

This update fails with the error:

"Msg 511, Level 16, State 1, Line 1

Cannot create a row of size 8061 which is greater than the allowable maximum of 8060.The statement has been terminated."

If you update a fixed-length column value, you will never get error 511 because the fixed-length value, by definition, retains the same size regardless of the value. Most applications that update a fixed-length column value do not check for error 511.

What happens when you enable vardecimal storage format on a table? Since the numeric/decimal data is now stored by using variable-length storage, an update to a decimal/numeric value can potentially fail just as it failed when updating the **varchar** column in the previous example. Since applications do not expect updates to decimal/numeric columns to fail (unless of course there are constraints defined on the decimal/numeric value), the application may encounter an unexpected failure. To prevent this, SQL Server allows enabling vardecimal storage format on a table if and only if it can gurantee that updates to decimal/numeric values will never fail with error 511. So, for the table in the previous example, enabling vardecimal storage format:

sp\_tableoption 'boundary', 'vardecimal storage format', 1

GO

Returns the following error:

"Msg 1721, Level 16, State 2, Procedure sp\_tableoption, Line 129

Altering table 'boundary' has failed because the row size using vardecimal storage format exceeds the maximum allowed table row size of 8060 bytes."

An example helps explain the reason for this failure. Assume that we insert the row with values ('a', 'b', replicate ('1', 12), 0.0. The decimal column with a 0.0 value uses only 2 bytes (offset array) of storage in vardecimal storage format. Thus, the total current row size is 8,045. Had SQL Server allowed vardecimal storage format on the boundary table in the preceding code example, the following steps for updating decimal data would fail.

1. Update column c3 to ‘1234567890124567890’ (change its size from 12 bytes to 20 bytes). The current row size becomes 8,053 bytes.
2. Now update decimal column c4 to max digits (38) allowed, which requires 18 additional bytes of storage. This update cannot succeed because we have only space for (8060 - 8053) = 7 additional bytes.

Similar restrictions apply when you create an index and one or more key columns are decimal or numeric data type. In SQL Server, the cumulative size of index key columns cannot exceed 900 bytes. Note that the 900-byte limit excludes non-key columns that are included in the definition of nonclustered indexes. The following example illustrates the size limitation and the vardecimal storage format:

CREATE TABLE boundary(c1 char(883), c2 decimal(38,2))

GO

CREATE NONCLUSTERED INDEX nci\_boundary on boundary(c1, c2)

GO

The index creation succeeds because the cumulative size of the index key columns is (883 + 17) = 900. However, if vardecimal storage format were enabled on this table, the index creation would fail because SQL Server would require 18 bytes to store key column c2 in the worst case.

sp\_tableoption 'boundary', 'vardecimal storage format', 1

GO

-- Msg 10617, Level 16, State 1, Procedure sp\_tableoption, Line 12 :

-- Index 'nci\_boundary' could not be created or rebuilt. This index has a

-- maximum key length of 901 bytes with vardecimal storage format. This

-- length exceeds the maximum allowed key length of 900 bytes.

You may not be able to enable vardecimal storage format on tables that have a data row length of around 8,060 bytes and an index key length of around 900 bytes. This is not a serious issue because the row size of tables created in most application schemas is much less than 8,060 and the cumulative index key length is less than 900. However, it is reassuring to know that your application will not have problems when vardecimal storage format is enabled.

##### Disabling Vardecimal Storage Format

Disabling vardecimal storage format is expensive. The cost and implications are identical to those resulting from enabling vardecimal storage format. SQL Server guarantees that you can always revert back to fixed-length storage format unless you run out of disk space during the conversion. In this case, the operation will fail. This is similar to a failure that is due to not having enough disk space when creating or rebuilding an index.

To disable vardecimal storage format on a table, you can use the following command:

sp\_tableoption '<table-name>', 'vardecimal storage format', 0

To disable vardecimal storage format on the database, you must first disable vardecimal storage format on all tables in the database. For example, you have a database with one or more tables enabled for vardecimal storage format and now you want to disable this option. To disable it on the database, execute the following command (which will fail—the purpose of this is test to see if tables in the database have vardecimal storage format enabled):

EXEC sp\_db\_vardecimal\_storage\_format '<database-name>', 'OFF'

This command will fail with the following error:

"Msg 5086, Level 16, State 1, Line 1

There are one or more tables with vardecimal storage format enabled. sp\_db\_vardecimal\_storage\_format statement failed."

If there is another active session in the database when you run the command, you will get the following error. You must close other concurrent sessions before executing the command.

"Msg 5061, Level 16, State 1, Line 1

ALTER DATABASE failed because a lock could not be placed on database 'compression'. Try again later. sp\_db\_vardecimal\_storage\_format statement failed."

Now that you know that there are one or more tables that have vardecimal storage format enabled, you can use the same query used in a previous example to find the names of these tables as in the following code. After that, you can disable vardecimal storage format on each table individually.

SELECT name, object\_id, type\_desc

FROM sys.objects

WHERE objectproperty(object\_id,

N'TableHasVarDecimalStorageFormat') = 1

Here is the sample output that shows that there are two user tables that have vardecimal storage format enabled:

name object\_id type\_desc

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

t\_decimal1 34102012 USER\_TABLE

t\_decimal2 1074102867 USER\_TABLE

One more requirement is necessary before you can disable vardecimal storage format on the database—set the recovery model of the database to simple recovery. Because this breaks the log backup chain, you must perform another full database backup. This could potentially limit point-in-time recovery, so it must be planned very carefully. Also, you need to strongly warn people to make sure they are aware that enabling vardecimal breaks the continuity of the log. You do not want people enabling vardecimal storage format just to experiment with it.

##### Impact on Database Migration

This section describes the impact on database migration in the context of vardecimal storage format. There are essentially three kinds of migration for a database:

* **Migrating a database that was created in an earlier version of SQL Server to a later version.** For example, you want to migrate a SQL Server 2000 database to the SQL Server 2005 instance. This is a typical database upgrade and is always supported. There are no special considerations when upgrading your database to an instance of SQL Server 2000 SP2 in any environment, including environments that use database mirroring or clustering. Like SQL Server 2005 SP1, SP2 supports rolling upgrades. For information on rolling upgrades, see SQL Server 2005 Books Online. By default, the database is not enabled for vardecimal storage format when you upgrade it to SQL Server 2005/SP2 instance.

Note that you cannot migrate a database that was created in a later version of SQL Server to an earlier version of SQL Server. For example, you cannot migrate a SQL Server 2005 database to a SQL Server 2000 instance. This is because SQL Server 2000 does not support the new physical storage structures in SQL Server 2005. In theory, if there were no changes in the physical structures between the two versions of SQL Server, you would be able to migrate your database between them any number of times freely. However, in reality this is never the case.

* **Migrating a SQL Server database across service packs for the same SQL Server version.** An example of this is migrating a database from SQL Server 2005 to SQL Server 2005 SP1. SQL Server also allows you to migrate a SQL Server database created on a SQL Server 2005 SP1 instance to a SQL Server 2005 instance. This is possible because, historically, the physical storage structures have not changed between different service packs of the same SQL Server version. However, SQL Server 2005 SP2 is an exception. Since the vardecimal storage format changes the physical storage format of decimal or numeric data types, you cannot migrate a vardecimal-enabled database to an instance of SQL Server 2005 or SQL Server 2005 SP1. However, if you do not enable vardecimal storage format in the database in the SQL Server 2005 SP2 instance, there are no restrictions on migrating it to earlier service packs of SQL Server 2005.

SQL Server prevents migration of a vardecimal-enabled database to earlier versions of SQL Server 2005 by incrementing the database version when the vardecimal storage format is enabled. To migrate to an earlier version, you must disable vardecimal storage format on the database. When vardecimal storage format is disabled, the database version is decremented to the same version as in SQL Server 2005 and then the database can be attached to an instance of SQL Server 2005 or SQL Server 2005 SP1. You must also set the database to simple recovery model before disabling vardecimal storage format on the database. The reason for this is to break the log chain; otherwise, the log backup may have records with the new storage format which are not supported by earlier versions of SQL Server. The following scenario illustrates this point.

1. Do a full physical backup (DB) on SQL Server 2005 SP1. This backup will have the database version that is supported by SQL Server 2005 SP1.
2. Upgrade the instance to SQL Server 2005 SP2.
3. Enable vardecimal storage format on the database.
4. Create a table and enable it for vardecimal storage format
5. Insert one row. The log records generated by this will have data in vardecimal storage format
6. Disable vardecimal storage format on the database.
7. Do the log backup (L). This log backup will have the database version that is supported by SQL Server 2005 SP1.

Now if you restore (DB + L) on SQL Server 2005 SP1, it will not detect that there were log records with vardecimal storage format and will potentially fail unpredictably.

To prevent this, set the database to simple recovery model before you disable vardecimal storage format on the database. Setting the recovery model to simple breaks the log chain and the problem is prevented. You are not required to set the database to simple recovery model when you disable vardecimal storage format on individual table(s). This is required only when disabling vardecimal storage format at the database level.

* **Migrating SQL Server databases to different editions of SQL Server 2005 SP2.** The vardecimal storage format is supported only in the SQL Server 2005 Enterprise Edition and the Development Edition. If you have a database enabled with vardecimal storage format, it cannot be migrated to a SQL Server edition that does not support this format. This restriction is similar to the one for table partitioning. If you want to migrate to an edition of SQL Server that does not support vardecimal storage format, attach the database to the Enterprise or the Development edition and remove vardecimal storage format from the database. Only then can the database be attached to an edition that does not support vardecimal.

##### Impact on Log Shipping

Log shipping is based on log backup/restore, so it behaves similarly to backup/recovery. If vardecimal storage format is not enabled, there are no restrictions. However, if you will enable vardecimal storage format, consider the following two combinations:

* **Primary and secondary servers are both running SP2** (both with vardecimal storage format enabled)

Allowed. You can enable vardecimal storage format on the primary. This is a logged operation. When the transaction log backup is restored on the secondary, vardecimal storage format will automatically be enabled.

* **Primary server is running SP2 and secondary is running SP1** (vardecimal storage format is enabled on the primary)

Not allowed. Restoring log records or physical backup on the secondary instance will fail.

##### Impact on Database Mirroring

If a database has not been enabled for vardecimal storage format, no changes are required in the mirroring configuration. You can mix and match SQL Server SP1 and SP2 for the principal and mirrored instances. If you do not want vardecimal storage format, you can install SP2 on mirror without setting ‘partner off.’ In a rolling upgrade scenario, you can upgrade both the principal and the mirror one at a time. Note that this behavior is the same as you would experience with SP1 and SQL Server 2005.

If you want to enable vardecimal storage format under database mirroring, you must run both the principal and the mirrored instance in SQL Server SP2.

To enable vardecimal on the database in a mirrored environment

1. Remove database mirroring by executing:

ALTER DATABASE <database\_name> SET PARTNER OFF

1. Upgrade both the primary and mirrored instance to SQL Server 2005 SP2.
2. Enable vardecimal on the principal database by executing:

sp\_db\_vardecimal\_storage\_format ‘<database\_name>', 'ON'

1. Re-establish database mirroring. For more information on database mirroring, see SQL Server 2005 Books Online.

Once the mirror is re-established, a user may choose to enable vardecimal storage format on individual tables. Since enabling vardecimal storage format on a non-empty table must update each data row and possibly one or more nonclustered indexes, enabling vardecimal storage format will generate log records proportional to the size of the table. You may want to consider enabling vardecimal storage format on one or more tables before re-establishing database mirroring.

To disable vardecimal storage format in a database in mirroring environment, you must first remove database mirroring, and then put the principal in the simple recovery model, which breaks the log chain. After the principal database has been disabled for vardecimal storage format, database mirroring can be re-enabled.

##### Impact on Replication

Changing the database vardecimal storage format state requires an exclusive lock on the database for the duration of the operation. The command will fail if there are active users in the database as that will prevent acquiring the exclusive lock on the database. This command waits only a short time to acquire the exclusive lock before timing out. If the database is being replicated, stop the replication agents before enabling vardecimal storage format on the database.

##### Performance

The only reason to use vardecimal storage format is to reduce the size of the table and thereby of the database. Reducing the size of the database has benefits other than the obvious savings of disk space. A smaller database leads to fewer I/Os, reduces memory pressure on the buffer pool (you can fit more *information* in the buffer pool), and reduces the size of a backup and thereby the time required to back up a database.

It sounds all good, but there is a catch. The catch is that it takes more CPU cycles to convert decimal values from fixed length to vardecimal storage format and then back. That means that if your workload is CPU bound, you may see degradation in the performance of your workload. If your workload is I/O bound, you may see a performance gain.

How do you determine the impact of vardecimal storage format on your workload? The answer is, of course, “It depends.” To remove some of the uncertainties of the performance impact, we tested many scenarios in our performance lab and also validated the tests with production workloads. This section describes these scenarios and test results. This will help you to better understand the performance implications of vardecimal storage format.

###### Table Scans

To test table scans, in a very extreme test case, a table with 20 decimal (38, 0) columns was used. After 10 million rows were inserted with only the value '0' in all columns, the table used about 450,000 pages on disk. After activating the new vardecimal format, the table used only about 14,000 pages—roughly 30 times less. The same factor was then measured for a simple sequential scan, which was also about 30 times faster after vardecimal storage format was enabled. No doubt that is a "best case" scenario that might never happen in a real-life application. But it is certain that I/O-bound scans are faster if there is a significant reduction in space due to the vardecimal storage format.

###### Variable-Length Row and Forward Pointers

Using vardecimal storage format may convert a row that contains only fixed-length columns to one that contains variable-length columns. (Any column with decimal or numeric type will now be stored in the variable portion of the row.) For example, a table that consists only of decimal and integer columns has a fixed row length. After enabling vardecimal storage format, rows contain variable-length columns. There is a well-known side effect of having variable-length columns in a table that does not have a clustered index. In this case, if you update a "small" value (for example, 1-digit : '1' ) to a "big" value (for example, 20-digit : '12345678901234567890'), the data row may not fit on the current page and must be moved to the new page. SQL Server handles this by using a forwarding record that points to the moved row (called the *forwarded record*) from the old row location, thereby removing the necessity of updating all nonclustered indexes. The disadvantage is that you may need up to two physical I/Os to accomplish this: one to access the ‘forwarding row’ and a second to access the ‘forwarded row.’ This can hurt the performance of your application. You can address this performance issue by creating a clustered index; this eliminates the forwarded rows.

###### CPU-Bound Scans

As previously indicated, no application changes are required to take advantage of the vardecimal storage format. One important aspect of this format is that it is used only for the storage of pages on disk and in memory (that is, the buffer pool) but not for internal query processing. Decimal and numeric values in the row are converted during fetch or scan operations to the fixed-length storage format during fetch before returning them to the query execution engine in SQL Server. Thus, the cost of converting from vardecimal storage format to fixed-length storage format is incurred only by scan operations.

Similarly, fixed-length decimal values are converted to vardecimal storage format during insert and update operations. This conversion overhead combined with storing and retrieving column values from the variable portion of the row can add to CPU overhead. So if your workload is CPU bound, you will see a degradation in performance after vardecimal storage format is enabled.

The worst-case scenario for vardecimal storage format is when the table fits in the buffer pool and you are doing an aggregate operation on a decimal column over all rows. In this case, there is no performance to be gained by reduced size, as the table already fits in memory. As you can imagine, this workload is CPU-bound. In our internal testing, we found that in this worst case, the performance of the workload was twice as slow after vardecimal storage format was enabled. This kind of workload is rare, but it does emphasize that CPU-bound workloads will see degradation in performance.

###### OLTP-Like Workload

In our internal test environment, we did not find a case where the response time of OLTP-like queries increased after vardecimal storage format was enabled. There were two major reasons for this. First, the cost of query execution was tilted towards relational operators other than scan, which potentially compensated the CPU impact of the conversion between fixed-length and variable-length storage formats. Second, since the size of the table was reduced, the chances of finding the page in the buffer pool increased proportionally.

###### Mass Updates, Inserts, and Deletes

Internal tests showed results similar to CPU-bound scans and OLTP-like queries. In certain cases "mass updates” and "mass inserts" were over 20% slower when the vardecimal storage format was enabled. There was no disadvantage when running single-row updates or single-row inserts in a loop. The situation is more complicated with updates and deletes, as SQL Server must first search for the rows that must be changed. The search operation could benefit from the reduced table size and the additional CPU cost would be paid only for conversion to vardecimal storage format.

###### Production Workload SAP BI

SAP BI is the primary workload targeted by vardecimal storage format. To measure the impact on an SQL BI application, we worked with one of our SAP BI customers in Germany to test the impact of vardecimal storage format both in terms of space savings and the impact on the performance of the real-life workload. This customer was very interested in the new storage format to reduce the size of fact tables within SAP BI info cubes that often included a large number of decimal columns. The customer was able to reduce the size of their fact tables up to 80% in some cases without any noticeable impact on the performance of their workload in spite of the extra CPU cycles required to convert decimal data between fixed-length format and vardecimal storage format. This was because the customer’s workload was running complex queries involving many joins, aggregates, and other expensive relational operators. These constituted the significant portion of the cost of the query as compared to the cost of scan operators, which pay the additional cost of conversion from vardecimal storage format to fixed-length format. Another way to look at this is that if the 95% CPU cost of your query is in executing relational operators other than scan, even if the CPU cost of scanning goes up by 40%, it is still 2% (that is, 40% of the 5% query cost) of the overall query cost. Here is a summary of the results:

* For the majority of the fact tables, the disk space savings was between 20 and 50%.
* As expected, the space savings were dependent on the table schema (the relative number of decimal and numeric columns), the size of the table, and the data distribution. Thus, for one table that had just one decimal column out of nine total columns, the saving was only 4.76%. Another table, which had 94 decimal columns out of 109 total columns, became 80% smaller in size.
* There were no space savings for indexes because the indexes defined on the SAP BI fact tables did not have any decimal or numeric key columns
* As mentioned earlier, there was no noticeable impact on the query or load performance with vardecimal storage format.

###### Production Workload Microsoft Internal Application

This internal Microsoft data warehouse application runs critical business functions for the company on a very large database around 1 terabyte. We enabled vardecimal storage format on this database and evaluated the space savings and the impact on performance. A test with a mix of over 100 different ad-hoc queries showed an average improvement of about 15% in query response time when we ran these queries sequentially. Interestingly, some of the queries that were I/O-bound became almost twice as fast after enabling vardecimal storage format. Based on our analysis of the application, the main reason is that this application runs significant number of ad‑hoc queries on very large tables (hundreds of gigabytes (GB)) with millions of rows where it is not possible to use any kind of aggregation. In this scenario, it is almost impossible to keep everything in memory. So it was no surprise that this application had queries that were I/O bound. Since this database had tables with many decimal columns, we were able to reduce the size of the some of the tables significantly by using vardecimal storage format, which led to a reduction in the I/O, thereby helping to improve the performance of the queries.

The following table shows the space savings when using vardecimal storage format in this test.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Table Name | Before Size (MB) | After Size (MB) | Diff (MB) | Savings % | Rows |
| TDSTINVSL00 | 397.46 | 360.34 | 37.12 | 9 | 206379 |
| TDSTPURSL00 | 12305.91 | 11059.36 | 1246.55 | 1 | 1666744 |
| TDSTSALSL00 | 17295.14 | 15600.46 | 1694.68 | 1 | 2376630 |
| TECSBUDSL00 | 2633.17 | 1365.32 | 1267.85 | 48 | 352065 |
| TECSFORSL00 | 1556.45 | 668.42 | 888.03 | 57 | 310278 |
| TECSPURSL00 | 125864.06 | 100472.57 | 25391.49 | 2 | 11287803 |
| TMSFBUDLL00 | 1487.64 | 952.91 | 534.73 | 36 | 138045 |
| TMSFBUDSL00 | 1284.21 | 836.22 | 447.99 | 35 | 119536 |
| TMSFFORSL00 | 786.28 | 417.05 | 369.23 | 47 | 190540 |
| TMSFSALLL00 | 155987.04 | 120095.39 | 35891.65 | 23 | 7124813 |
| TMSFSALSL00 | 104934.5 | 81824.08 | 23110.42 | 22 | 5782899 |
| TRSLINVSL00 | 12629.58 | 11212.8 | 1416.78 | 11 | 8203836 |
| TRSLPURSL00 | 39845.78 | 36330.93 | 3514.85 | 9 | 5239574 |
| TRSLSALSL00 | 72123.42 | 65796.36 | 6327.06 | 9 | 10927014 |
| TVOLSALSL00 | 108013.48 | 101114.59 | 6898.89 | 6 | 5121061 |
| **Total** | 657144.12 | 548106.8 | 109037.32 | 17 |  |

#### Conclusion

The important question is whether to enable vardecimal storage format. If the answer is yes, the next question is: on which tables? As explained in this white paper, there are multiple factors to consider before enabling vardecimal storage format:

* First evaluate the space savings that you can get by enabling vardecimal storage format. For smaller tables (< 1% the size of the database), it may not buy you much in terms of space savings while still incurring the CPU cost of data conversion to vardecimal storage format.
* You will incur some CPU overheard for converting data between vardecimal storage format and fixed storage format. If your workload is already CPU bound, you will incur degradation in the performance of the workload. The degradation in the performance will be proportional to the relative cost of scan operators with respect to the overall CPU cost of the query.
* If you have an I/O-bound workload, you are likely to see an improvement in the performance of your workload.
* Evaluate the implications of vardecimal format on database migration.

As is true with any significant change, test the impact of vardecimal storage format in your test lab before deploying it to a production environment.

For more information:

[SQL Server Developer Center](http://msdn2.microsoft.com/en-us/sql/default.aspx) on MSDN

[SQL Server TechCenter](~$rdecimalStorage.docx) on TechNet

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 white papers we release. [Send feedback](mailto:sqlfback@microsoft.com?subject=White%20Paper%20Feedback:%20%5bTitle%20of%20white%20paper%5d).