

Title: Performance Considerations of Data Types
Author: Michelle Ufford, <http://sqlfool.com>
Technical Reviewer: Paul Randal, <http://sqlskills.com/AboutPaulSRandal.asp>
Paul Nielsen, <http://www.sqlserverbible.com/>
Date Published: 13 May 2009

Selecting inappropriate data types, especially on large tables with millions or billions of rows, can have significant performance implications. In this article, I'll explain why and offer suggestions on how to select the most appropriate data type for your needs. The primary focus will be on common data types in SQL Server 2005 and 2008, but I'll also discuss some aspects of clustered indexes and column properties. Most importantly, I'll show some examples of common data-type misuse.

About Data Types

[Data types](#)¹ in SQL Server refer to one of the attributes of a column, variable, or parameter. For the purposes of this article, we will focus on data types as they relate to columns in a table. A data type is required² when defining a column in a table and tells SQL Server which values may and may not be stored in the column. In many ways, data types are a form of CHECK constraint and can be useful for enforcing business logic. For example, an **int** column cannot contain alphabetic characters, and a **bit** column can only contain values equal to 0, 1, or NULL.

Let's take a look at some common data types and their storage requirements:

Data Type	Values Stored	Storage Requirements
bit	0, 1, NULL	1 byte per 8 bits
tinyint	0 to 255	1 byte
smallint	up to +/- 32,767	2 bytes
int	up to +/- 2,147,483,647	4 bytes
bigint	up to +/- 9,223,372,036,854,775,807	8 bytes
smalldatetime	1900-01-01 to 2079-06-06, minute precision	4 bytes
datetime	1753-01-01 to 9999-12-13, sub-second precision	8 bytes
date (SS 2008)	0001-01-01 to 9999-12-31	3 bytes

Do not underestimate the importance of selecting appropriate data types during table creation. One Fortune 500 company I've worked with learned this the hard way. For various reasons, the company

¹ <http://msdn.microsoft.com/en-us/library/ms187752.aspx>

² The exception to this requirement is computed columns; SQL Server infers the data type for computed columns from the source columns' data types.

decided to use **char** columns for certain critical columns, such as order and customer number, when initially designing its systems. As the company grew and their databases increased in size and complexity, the data types became increasingly more painful to maintain. One systems issue that occurred was a **char(4)** column running out of space to store new values. A **char(4)** column will consume 4 bytes of storage and can store a max numeric value of 9999. The problem was solved by switching to an **int**, which also consumes 4 bytes of storage but can store up to a positive or negative value of 2,147,483,647. Another issue was the decision to allow alphanumeric values in order numbers. Account managers and customers often confused letters and numbers, such as zero and “o,” which sometimes led to costly mistakes.

The issues this company faced are too numerous for this paper, but suffice it to say that the company incurred a significant amount of time and expense (and errors!) because of its poor data type choices. Eventually, the company spent a small fortune to hire consultants to fix the problem.

Wasted Space and IO

It's unfortunately very common for developers and even database administrators to define an inappropriate data type when creating a table. In fact, I've met many developers who habitually use an **int** data type for every numeric column, a **nvarchar** data type for every alphanumeric column, and a **datetime** data type for every timestamp column. While there is certainly nothing wrong with these 3 data types—indeed, they are some of the most commonly used data types in SQL Server—a smaller data type may suffice in many cases. This is important because using a larger data type than needed can waste space and consume unnecessary IO.

Let's look at an example of how this happens. Assume you have the following table:

```
CREATE TABLE myBigTable
(
    myID          INT IDENTITY(1,1)
    , myNumber    INT
    , myDate      DATETIME

    CONSTRAINT PK_myBigTable
        PRIMARY KEY CLUSTERED (myID)
);
```

For the purposes of this example, let's assume `myBigTable` will hold 1 million records. `myID` is an auto-incrementing column; because of the number of rows we're storing, an **int** makes the most sense. Let's also assume that `myNumber` will hold a value between 1 and 100, and `myDate` will hold the date and time of the record insert.

A data page can hold 8192 bytes, with 96 bytes reserved for the header. This leaves 8096 bytes available for row data. In Table 1, you saw that an **int** column consumes 4 bytes and a **datetime** column consumes 8 bytes. You also need to be aware that there is 9 bytes of overhead for each row. The amount of overhead per row can also increase if other factors exist, such as variable-length and nullable columns. For now, we can use this information to determine how much space is required to store 1 million rows.

Bytes per row:	4 + 4 + 8 + 9 (myID + myNumber + myDate+ overhead)	= 25 bytes
Rows per page:	8096 / 25	= 323 rows ³
Pages per 1m rows:	1,000,000 / 323	= 3096 pages ⁴

So `myBigTable` will consume 3096 pages in the leaf level of the clustered index to store 1 million rows.

Now let's look at the same table, but this time I'll use more appropriate data types:

```
CREATE TABLE mySmallTable
(
    myID          INT IDENTITY(1,1)
    , myNumber    TINYINT
    , myDate      SMALLDATETIME

    CONSTRAINT PK_mySmallTable
        PRIMARY KEY CLUSTERED (myID)
);
```

I've made two changes. Earlier, I mentioned that `myNumber` will contain values from 1 to 100. If you refer to Table 1, you'll see that a **tinyint** consumes just 1 byte and can more than accommodate these values. Also, we've decided that the seconds precision of a **datetime** data type is unnecessary. Instead, a **smalldatetime** column will store the date of the insert down to the minute, while only consuming 4 bytes.

Let's see how much space our smaller table consumes:

Bytes per row:	4 + 1 + 4 + 9 (myID + myNumber + myDate+ overhead)	= 18 bytes
Rows per page:	8096 / 18	= 449 rows
Pages per 1mm rows:	1,000,000 / 424	= 2228 pages

³ Rounded down to the nearest whole number

⁴ Rounded up to the nearest whole number; this number only includes leaf-level pages

`mySmallTable` only consumes 2228 pages, compared to the 3096 pages required by `myBigTable` to store the same information⁵. This is a space savings of 27%. Also, because a single IO can now return 126 more rows, there's an IO performance improvement of 39%. As you can see, selecting the most appropriate data type for your needs can have a pretty significant impact on performance.

Unicode Data Types

Another common mistake involves Unicode data types, such as **nchar** and **nvarchar**. If you have a definite need to support Unicode characters, such as an international company or a global website, then you should certainly use **nvarchar** or **nchar** as appropriate. I would advise against using it frivolously, however. I've seen business requirements that specify every character column in every table support Unicode. This is potentially wasteful unless really necessary.

Unicode character data types require twice as much storage space as their non-Unicode equivalent. Imagine a large table with a comment column, perhaps a **varchar(1000)** column. Let's assume that the column averages a 30% fill rate, and there are 10 million rows in the table. The **varchar** column will require, on average, 300 bytes per rows; the **nvarchar** equivalent, by contrast, will require 600 bytes per row. For 10 million rows, the **nvarchar** column requires 2.8GB more space than its **varchar** equivalent. For this reason, evaluate the use of each column and only use **nchar**, **nvarchar**, and **ntext** in the columns that may actually contain international data.

CHAR vs VARCHAR

In a similar vein, careful consideration should be given when deciding whether to use **char** or **varchar**. A **char(n)** column is fixed length. This means SQL Server will consume *n* bytes for storage, regardless of the amount of actual data stored. The **varchar** datatype, by contrast, consumes only the amount of actual space used plus 2 bytes for overhead⁶. So at what point does it make more sense to use a **char** column? Roughly, when the difference between the character length and the average number of characters is less than or equal to 2 bytes. If you're not sure what the average length of the column will be, then I suggest using a **char** for column sizes less than 5 and a **varchar** for column sizes greater than 10.⁷ Think about the use case for column sizes between 5 and 10 and select the data type most appropriate to your requirements.

⁵ These numbers only refer to the amount of data pages in the leaf level of the clustered index

⁶ In addition to the 2-byte offset per variable column, there is also a 2-byte overhead per row to maintain the count of variable columns. This is only incurred if one or more variable columns exist in the table.

⁷ These are general recommendations. For frequently updated columns, a char could possibly perform faster and result in less fragmentation because it is a fixed-length data type.

Wide Clustered Indexes

Clustered indexes are a performance tuning topic of its own and are largely outside the scope of this article. However, as clustered indexes are typically defined at table creation, this topic warrants some discussion.

In general, it is best practice to create a clustered index on narrow, static, unique, and ever-increasing columns. This is for numerous reasons. First, using an updateable column as the clustering key can be expensive, as updates to the key value could require the data to be moved to another page. This can result in slower writes and updates, and you can expect higher levels of fragmentation. Secondly, the clustered key value is used in non-clustered indexes as a pointer back into the leaf level of the clustered index. This means that the overhead of a wide clustered key is incurred in every index created.

A good example of a poor clustering choice would be a **uniqueidentifier**, also known as a GUID. **Uniqueidentifier** columns are frequently encountered in database requirements, as GUIDs are a common and often beneficial data type in .NET applications. **Uniqueidentifiers** are an expensive pointer because they are typically not sequential and because they consume 16 bytes. One environment in which I've worked contained two tables, each with over a billion rows, clustered on a non-sequential **uniqueidentifier**, and 99.998% fragmented. By clustering instead on a **bigint identity**, I drastically reduced the time required for inserts. This also resulted in smaller indexes, saving over 100GB of disk space. In environments where the use of a **uniqueidentifier** is required, consider using an internal surrogate key, such as an **int identity**, as a clustering and foreign key to avoid the overhead and poor performance associated with GUIDs.

Unique Clustered Indexes

A clustered index created as part of a primary key will, by definition, be unique. However, a clustered index created with the following syntax,

```
CREATE CLUSTERED INDEX <index_name>  
    ON <schema>.<table_name> (<key columns>);
```

will not be unique unless **unique** is explicitly declared, i.e.

```
CREATE UNIQUE CLUSTERED INDEX <index_name>  
    ON <schema>.<table_name> (<key columns>);
```

In order for SQL Server to ensure it navigates to the appropriate record, for example when navigating the B-tree structure of a non-clustered index, SQL Server requires every row to have an internally unique id. In the case of unique clustered index, this unique row id is simply the clustered index key value. However, as SQL Server will not require a clustered index to be unique - that is, it will not prevent a clustered index

from accepting duplicate values - it will ensure uniqueness internally by adding a 4-byte uniquifier to any row with a duplicate key value.

In many cases, creating a non-unique clustered index on a unique or mostly unique column will have little-to-no impact. This is because the 4-byte overhead is only added to duplicate instances of an existing clustered key value. An example of this would be creating a non-unique clustered index on an identity column. However, creating a non-unique clustered index on a column with many duplicate values, perhaps on a column of **date** data type where you might have thousands of records with the same clustered key value, could result in a significant amount of internal overhead.

Moreover, SQL Server will store this 4-byte uniquifier as a variable-length column. This is significant in that a table with all fixed columns and a large number of duplicate clustered values will actually incur 8 bytes of overhead per row, because SQL Server requires 4 bytes to manage this variable column (2 bytes for the count of variable-length columns in the row and 2 bytes for the offset of the the variable-length column of the uniquifier column). If there are already variable-length columns in the row, the overhead is only 6 bytes—two for the offset and four for the uniquifier value. Also, this value will be present in all nonclustered indexes too, as it is part of the clustered index key.

Conclusion

Performance tuning is a vast subject, of which data types is just a small piece. Hopefully, you now have a better understanding of how data types can affect disk space and IO. As I've discussed, making poor data type choices in a large or busy database can have a trickle-down effect and ultimately impact the entire system negatively. It's much less expensive to spend some time upfront during the design phase to thoroughly think through data requirements than it is to modify a large database after data is loaded and problems start.

References

SQL Server Books Online