SQL Server:
Why Physical Database Design Matters

Module 2: Data Types and Row Size

Kimberly L. Tripp
Kimberly@SQLskills.com
http://www.SQLskills.com/blogs/Kimberly

pluralsight
hardcore developer training

Introduction: Does Data Type Choice Matter?

- **An int is 4 bytes and a bigint is 8 bytes – who cares, right?**
  - It's just disk space
- **Imagine a table where every date column is defined using:**
  - datetime: 8 bytes
    - January 1, 1753, through December 31, 9999
    - Precision to a time tick (3.33 ms)
  - datetime2: size varies from 6 to 8 bytes depending on the chosen precision
    - January 1, 0001 through December 31, 9999
    - Maximum precision to 100 ns
- **With four datetime (or, datetime2(7)) columns you are requiring 32 bytes of fixed-width storage to be allocated**
- **If even just two of those columns are stored as smalldatetime (4 bytes, January 1, 1900 through June 6, 2079) then you can save 8 bytes**
- **Will that help? Does it matter?**

It's Not Just About That Single Column Value

- For a large table, even just saving 8 bytes can make a difference…
- You must think about scale:

Bytes Saved	Rows	MB Saved	
8	10,000,000	76	
8	100,000,000	763	
8	1,000,000,000	7629	7.6 GB
16	10,000,000	153	
16	100,000,000	1526	
16	1,000,000,000	15259	15.3 GB
32	10,000,000	305	
32	100,000,000	3052	
32	1,000,000,000	30518	30.5 GB

- This is disk space, backups, maintenance, caching, logging, replication and HA feature performance and scalability, etc.

Use Common Sense: Don't Take It Too Far…

- **Don't use a tinyint for your customer ID because you currently only have 187 customers**
  - Changing data types later can be difficult
- **Use the smallest, most reasonable data type**
  - Choose a bigint over an int for a large table's ID because changing a KEY can be very difficult
    - bigint: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
    - int: -2,147,483,648 to 2,147,483,647
  - Choose tinyint for something like:
    - Age of people who are alive
    - Status codes (if you only have 15 status codes and you're not likely to add a lot more of these)
- **The key point – don't think that this stuff is irrelevant!**
- **It's not hard, it doesn't take a lot of time, and it does make a difference!**

Column Size and Number of Columns

- **Have you ever sat in a room with a bunch of stakeholders…**
  - *Gee, I think we should add a column called **abc** because we might, someday, consider storing something about how many widgets they considered buying when they were a teenager*
  - *Hey, that's a great idea! If you're going to add a column **abc** for that then I think we should also add a column **def** for (blah, blah, blah)*
  - And, this goes on
  - And on
- **Until your table has a large number of columns that fall into a variety of categories:**
  - Core, critical columns that you actually use regularly
  - Columns that are often queried
  - Columns that are often updated
  - Columns that no one even knows exist
  - Columns that no one uses, plans to use, or has any use for

Understanding Rows, Pages, and Storage

- **Tables have attributes (a.k.a. columns)**
- **Together these columns make up your data records (a.k.a. rows)**
- **A row can have different types of data structures associated with it:**
  - IN_ROW_DATA
    - Every row has an IN_ROW_DATA portion of the row
    - All fixed-width columns must be stored in the IN_ROW_DATA portion of the row
    - IN_ROW_DATA rows cannot span pages
  - ROW_OVERFLOW_DATA
    - Tables with the potential for wider rows might have one or more columns that have *overflowed* to the ROW_OVERFLOW_DATA structure
    - We'll discuss which column types can be stored there in a few slides
  - LOB_DATA
    - Tables with the potential for extremely wide rows might have one or more columns that have been stored in the LOB_DATA structure
    - We'll discuss which column types can be stored there in a few slides

Data Storage Structure: IN_ROW_DATA

- **Maximum storage per data/index row for IN_ROW_DATA is 8,060 bytes**
  - This limit is based on the following:
    - 8,192 bytes per page minus the page header of 96 bytes = 8,096 bytes for data
    - 2 bytes are used for the row's pointer within the slot array = 8,094 bytes for data
    - 34 additional bytes were reserved for "future use" (when SQL Server 7.0 was being designed) = 8,060 bytes
  - 8,060 bytes was the maximum row size in SQL Server versions 7.0 and 2000, SQL Server 2005 added the ability for a row to overflow to additional pages
- **Fixed-width columns are always part of the IN_ROW_DATA structure**
  - You cannot create a table with 10 char(2000) columns
  - You can create a table with 10 varchar(2000) columns even if ALL columns are filled to capacity (some data will "overflow" and be stored off page)
- **A table is limited to only 1,024 columns unless you are using SPARSE attributes (and, an XML column-set)**
  - With SPARSE columns you can have up to 30,000 columns (in SQL Server 2008+)

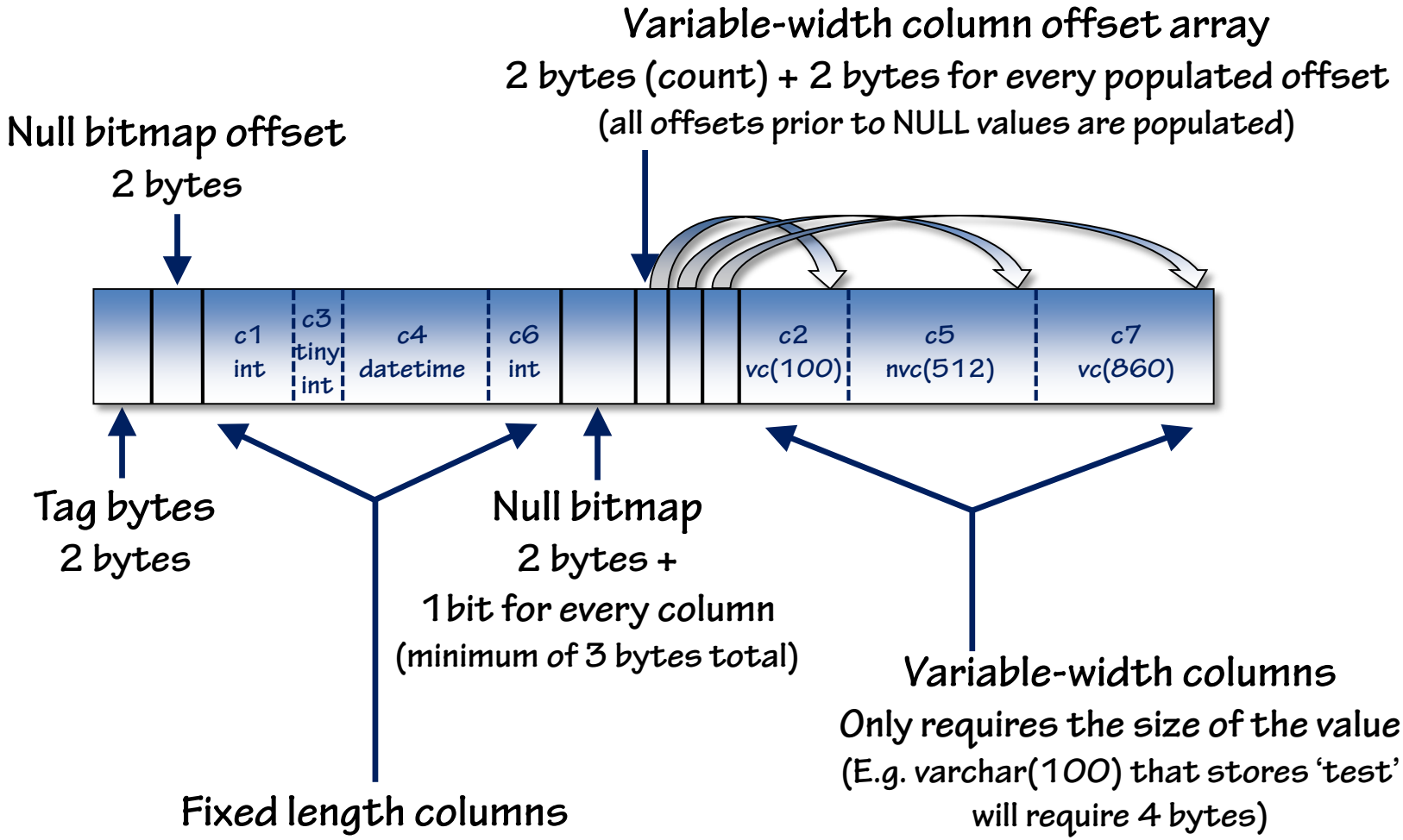
Data Storage Structure: ROW_OVERFLOW_DATA

- **Maximum storage per column for overflow data is 8,000 bytes**
  - Limited (n) LOB columns
  - Data types in this category
    - varchar(n) / nvarchar(n) / varbinary(n) / sqlvariant
- **Columns only overflow when the total row length is greater than 8,060 bytes**
  - Which columns overflow is not predictable (this can lead to poor performance)
    - If a particularly important column overflows then this adds an extra I/O when the row is accessed
    - If large ranges of rows are accessed and one of the columns has overflowed to other pages then this creates very inefficient I/O patterns
- **MISTAKE: Letting a lot of your columns overflow to additional pages without any thought for usage patterns and the negative effect on I/O**
- **SOLUTION: Column placement can be better controlled through vertical partitioning**

Data Storage Structure: LOB_DATA

- **Consists of one these types:**
  - Pre-SQL Server 2005 types (generally should not be used in new applications): text, ntext, and image
  - Current LOB types: varchar(max), nvarchar(max), varbinary(max), XML, and CLR user-defined types
- **LOB data is stored off-page when the LOB column itself exceeds 8,000 bytes or (usually) when row size exceeds 8,060 bytes**
  - Can lead to the same I/O problems as ROW_OVERFLOW_DATA storage
  - Can restrict the use of online operations prior to SQL Server 2012
- **MISTAKE: Designing without understanding both the impact to maintenance operations (pre-2012) and performance (all versions)**
- **SOLUTION (all versions): Performance and maintenance operations can be better controlled through vertical partitioning (even in 2012)**

Record Structure



NOTE: This structure is only for IN_ROW_DATA without any SPARSE columns or data compression enabled

Record Structure: Key Points (1)

- **Fixed-width columns require exactly that – a fixed amount of space**
  - **MISTAKE**: Using the wrong data type for a column
  - **SOLUTION**: Take time to choose the right data type
- **Row size is impacted by the data types chosen, not necessarily the data that is stored**
  - **MISTAKE**: Thinking that NULL values do not require any space/overhead
  - **SOLUTION**: Take time to choose the right data type
- **Internal storage for 100-character data value stored in a varchar(8000) column is the same as when it is stored in a varchar(500) column, but the data integrity implications are far more complex**
  - **MISTAKE**: Thinking that all columns should just be varchar(8000) because "it doesn't really matter"
  - **SOLUTION**: Always analyze the domain of legal values for an attribute and limit the column appropriately

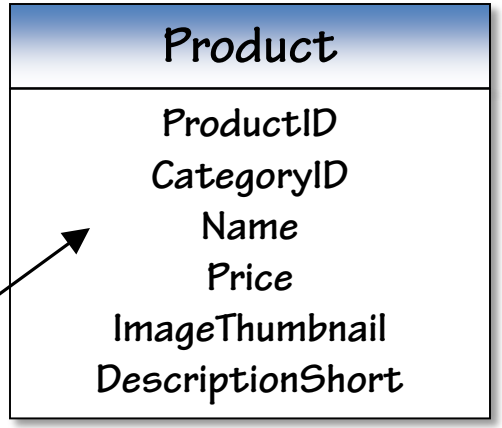
Record Structure: Key Points (2)

- **IN_ROW_DATA cannot span pages**
  - Fixed-width columns defined for a row must fit within the 8,060 byte row limit
- **This does not include the columns that are stored off-page**
  - A column that's overflowed will go to one additional page
    - For every column that overflows, an 8-byte pointer is added to the variable-width portion (and an extra I/O will be required to access that data)
  - A column of type LOB can go to a tree of pages
    - For every LOB column that is stored off-page, a pointer (usually 16 or 24 bytes, but up to 72 bytes) is added to the variable-width portion (and at least one extra I/O will be required to access that off-page data, possibly many more)
- **MISTAKE: Creating inefficient row structures will cause wasted disk space, wasted cache, and additional I/O overhead**
- **SOLUTION: Use vertical partitioning to better isolate columns and taking time to choose appropriate data types**

Vertical Partitioning (Limited LOB and LOB)

- **Imagine a table with a large number of wide, descriptor columns**
  - Table: Product

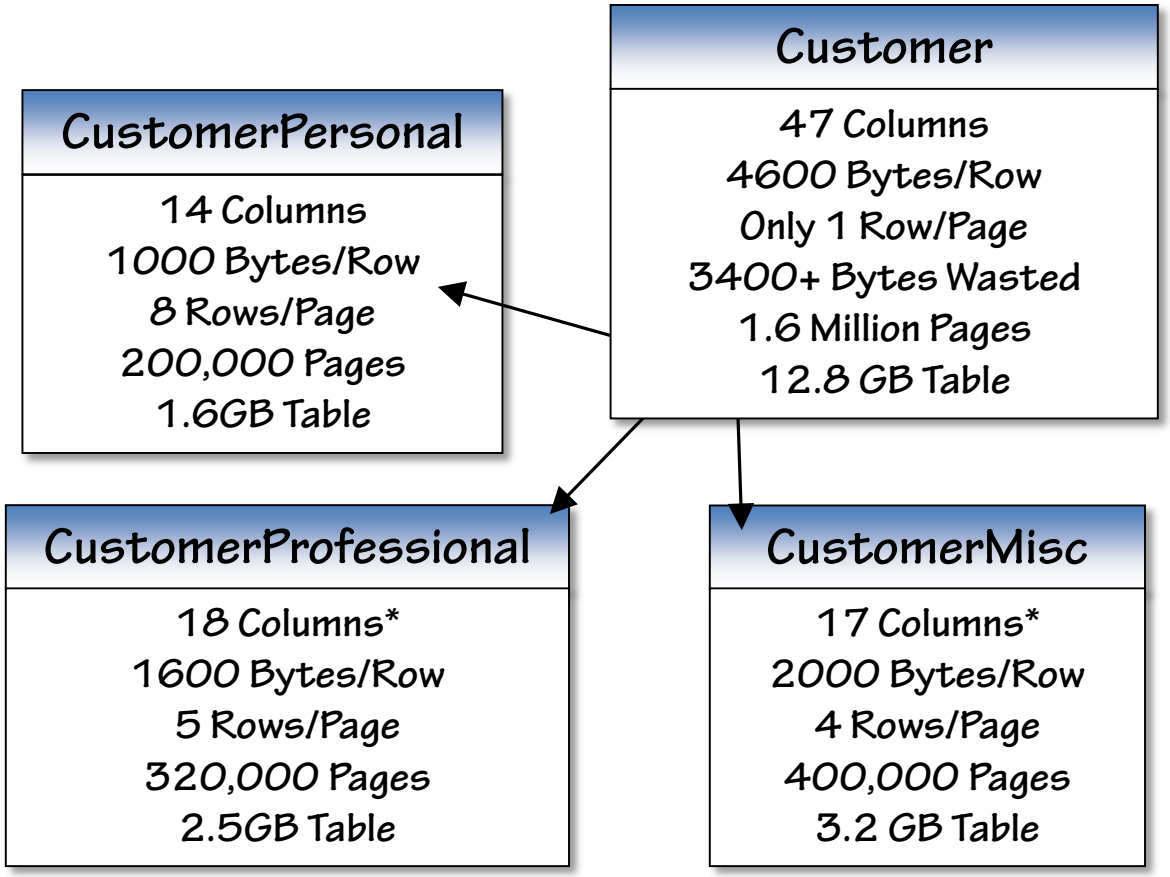
This table has very wide rows and poor page density. Which columns end up in the IN_ROW_DATA structure, and which overflow, is non-deterministic and can lead to poor performance (e.g. trying to display all Products in a Category).



This table will have narrow rows and therefore more rows per page (better data density). Queries that display large numbers of rows can access this data more efficiently and more of the critical data can fit in cache.

IF most requests for product details are singleton requests (drilling down from a Product list), then these "lookups" are very inexpensive.

Vertical Partitioning: Customer Table



* The PRIMARY KEY column(s) must be made redundant for the additional tables. Above: 47 columns in the original Customer table; 49 columns total between the 3 vertically partitioned tables.

- Customer = 12.8 GB
  - 1,600,000 rows
- Partitioned tables = 7.3 GB
- Savings in overall disk space (5.5 GB saved)
- Not reading data into cache when not necessary
- Overflow data can be isolated to reduce I/O costs for critical columns
- LOB data can be isolated to support online index operations of more critical data, for systems prior to 2012
- Locks are table specific therefore less contention across the customer data, when vertically partitioned

Vertical Partitioning Strategies

- **Understand the column usage above all else**
- **Usage defines vertical "partitions" or "sets"**
  - Keep columns often queried together in the same table
    - Logically group columns to minimize joins
  - Consider read only vs. OLTP columns
  - Consider LOB separate from OLTP to allow online index maintenance prior to SQL Server 2012 for the critical/OLTP part of the table
- **Optimizes row size for:**
  - Caching: better page density means less memory required
  - Locking: only locking the set of columns that are of interest reduces what would have been row-level conflicts
- **If every query requires a join, this isn't as optimal as it could be but should still be considered**

Record Structure and Compression

- **SQL Server 2008 Enterprise Edition introduced data compression**
  - Row-level compression, where every column, whether variable-width or not, is treated as variable width
  - Page-level compression, which does row-level compression, then per-column prefix compression, and per-page data dictionary compression
- **Is row-level or page-level compression going to give you a gain?**
  - Use: sp_estimate_data_compression_savings
- **These forms of compression do not require application changes to leverage but they do require overhead; they don't eliminate the need for good design!**
- **Whitepaper that will help you**
  - Data Compression: Strategy, Capacity Planning and Best Practices
  - http://bit.ly/1SBLqV

Summary: Column Size and Row Size Matter!

- **A minimal amount of analysis can yield more efficient column choices**
- **Better column choices can lead to increased scalability and longer application life**
  - Smaller rows means more rows per page and better page density
  - Fewer pages means less disk space AND less memory is required
  - Fewer pages means fewer pages to compress (and less compression overall)
- **Rows CAN span pages from SQL Server 2005 onward**
  - Is that always the best choice?
    - What will the impact be to the I/O patterns
    - Table access/usage patterns should dictate structure
- **Keep active/critical columns in core table**
- **Separate LOB types:**
  - For better management/placement
  - Prior to SQL Server 2012, to allow online index maintenance