

The background of the slide is a deep blue with a high-quality photograph of a single water droplet hitting a surface, creating concentric ripples that spread outwards. The droplet is captured at the moment of impact, appearing as a clear, spherical bubble. The ripples are lighter blue and white, contrasting with the darker blue background.

Designing for Performance

Understanding
SPARSE Columns

SQL
skills

Efficient Storage of Heterogenous Data

- ◆ Problem: how to design an efficient schema that allows an extremely large number of data fields?
 - ◆ Supporting an application that allows user-defined properties on multiple entity types, such as product catalog or document store
- ◆ Think of Sharepoint Server – storing tens of thousands of document types, each with hundreds or thousands of distinct attributes
- ◆ Given that SQL Server only supports up to 1024 columns in versions up to and including SQL Server 2005, what are the schema choices?

Conceptually, this is what your data looks like... but, no one does this! (*right?*)

DocID	DocName	DocType	DocSize	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19	P20	P21	P22	P23	P24	P25	P26
1	abcx	1	1,234	5	6	8	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	
2	defy	1	54	8	3	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	
3	ghiz	1	480	3	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	
4	jklb	2	340	null	null	null	a	c	r	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	
5	mnoa	2	45,896	null	null	null	z	null	o	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	
6	pqrz	2	123	null	null	null	b	r	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	
7	stuy	3	4,589	null	null	null	null	null	null	n	8	null	4	e	null	null	null	null	null	null	null	null	null	null	null	null	null	null	
8	vwxx	3	1,024	null	null	null	null	null	null	5	5	2	g	d	null	null	null	null	null	null	null	null	null	null	null	null	null	null	
9	yzau	3	56	null	null	null	null	null	null	3	z	null	8	f	null	null	null	null	null	null	null	null	null	null	null	null	null	null	
10	bcdt	4	237	null	null	null	null	null	null	null	null	null	null	a	b	c	d	e	null	null	null	null	null	null	null	null	null	null	
11	efgs	4	593	null	null	null	null	null	null	null	null	null	null	f	g	h	i	j	null	null	null	null	null	null	null	null	null	null	
12	hijr	4	457	null	null	null	null	null	null	null	null	null	null	k	l	m	n	o	null	null	null	null	null	null	null	null	null	null	
13	klmq	4	135	null	null	null	null	null	null	null	null	null	null	p	q	r	s	t	null	null	null	null	null	null	null	null	null	null	
14	nopp	5	75	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	7	null	null	null	null	null	null	null	
15	qrso	5	1,345	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	3	5	null	null	null	null	null	null	null	
16	tuvn	5	576,457	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	5	null	null	null	null	null	null	null	
17	wxym	6	24	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	1	2	null	null	null	null	null	
18	zabl	6	65	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	4	3	null	null	null	null	null	null	
19	cdek	7	23,523	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	z	e	null	null	null	null	
20	fghj	7	56	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	b	r	null	null		
21	ijki	7	24	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	a	s			
22	lmnh	7	6,457	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	a	r	e	null	b	l			
23	opqg	7	23	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	a	c	k	null	w	h			
24	rstf	7	757	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	i	t	e	null	h	o			
25	uvwe	7	234,234	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	r	s	e	s	null	null			

Why don't we have a single "flat" table today?

- ◆ The rows will be really wide
 - ◆ All fixed-width columns take that fixed amount of space regardless of whether or not they are NULL
 - ◆ Row size of 7.0/2000 was limited to a maximum of 8060 bytes (2005 does NOT have this limitation)
 - ◆ The maximum number of columns that SQL Server 7.0/2000/2005 supports is 1024
- ◆ Performance will be compromised
 - ◆ The maximum number of indexes that SQL Server 7.0/2000/2005 supports is 250 (generally you don't even want dozens, actually)
 - ◆ Performance is abysmal (even just for the metadata)
- ◆ It just looks wrong... ☺

Flexible Schema Choices

- ◆ Name/Value pairs table
- ◆ Normalized tables
- ◆ Properties as an XML structure
- ◆ Properties as a BLOB structure
- ◆ No flexibility...
 - ◆ Easy...
 - ◆ Not very flexible... ☺

Name/Value Pairs

- ◆ Core attributes in one table
- ◆ Attributes specific to each type – probably described in a metadata table
- ◆ Table gets very large, very quickly
- ◆ Rows x attributes
 - ◆ Rows without a value do not need to be stored at all (crossed out rows)
- ◆ Hard to index (especially the CL key)
 - ◆ Add an ever-incrementing key?
 - ◆ Nonclustered are almost as large as the table
 - fragmented and complex to administer
- ◆ “Value” column is of what type – sql_variant?
 - ◆ Comparison operators can be complex on this type – without conversions – these are costly as well...
 - ◆ See “Using sql_variant” in the BOL for a discussion on sql_variant issues including those around the data type hierarchy
 - ◆ char value ‘123’ < int value 111 (yes, ‘123’ is less than 111)

DocID	Property	Value
1	1	5
1	2	6
1	3	8
2	1	8
2	2	3
2	3	null
3	1	3
3	2	null
3	3	null
4	4	a
4	5	c
4	6	r
5	4	z
5	5	null
5	6	o
6	4	b
6	5	r
6	6	null
7	7	n
7	8	8
7	9	null
7	10	4
7	11	e

Normalized Tables

- Each “type” requires a new table – with only the attributes specific to that type
- Metadata for which table to join to needs to be included so the code is directed to correct table
- Lots of joins and/or dynamic string execution
- Hard to administer
 - Lots of tables to manage
 - Lots of indexes to manage
- But, probably the easiest for many to understand...

DocID	P7	P8	P9	P10	P11
7	n	8	null	4	e
8	5	5	2	g	d
9	3	z	null	8	f

DocID	P4	P5	P6
4	a	c	r
5	z	null	o
6	b	r	null

DocID	P1	P2	P3
1	5	6	8
2	8	3	null
3	3	null	null

DocID	P17	P18
14	null	7
15	3	5
16	null	5

DocID	P12	P13	P14	P15	P16
10	a	b	c	d	e
11	f	g	h	i	j
12	k	l	m	n	o
13	p	q	r	s	t

Properties in XML

- ◆ Core attributes in normal columns
- ◆ All attributes targeting a specific type are put into an XML structure
- ◆ XML structure can be strongly typed
 - ◆ XML schema by type – reference in your metadata layer or separate column (something like XMLSchemaID)
- ◆ Structure *can* be indexed:
 - ◆ Primary Index is EXPENSIVE
 - ◆ See: XML Indexes in SQL Server, Bob Beauchemin, SQLskills.com
http://msdn.microsoft.com/en-us/library/ms345121.aspx#xmlindexes_topic2
 - ◆ Primary Index requires that PK be clustered...
 - ◆ Secondary indexes can be very beneficial (however, you MUST create a PRIMARY index before you can create any secondary indexes)

DocID	ALL Values in a XML structure
1	<properties collection> <P1>5</P1> <P2>6</P2> <P3>8</P3> </properties collection>
2	properties collection> <P1>8</P1> <P2>3</P2> </properties collection>
3	properties collection> <P1>3</P1> </properties collection>
4	<properties collection> <P4>'a' </P4> <P5>'c' </P5> <P6>'r' </P6> </properties collection>
5	properties collection> <P4>'z' </P4> <P6>'o' </P6> </properties collection>
6	<properties collection> <P4>'b' </P4> <P5>'r' </P5> </properties collection>
7	<properties collection> <P7>'n' </P7> <P8>8</P8> <P10>4</P10> <P11>'e' </P11> </properties collection>

Quote from referenced article

Although having the primary XML index is a vast improvement over creating it afresh during each query, ***the size of the node table is usually around three times that of the XML data type in the base table***. The actual size depends upon the XML instances in the XML column—if they contain many tags and small values, more rows are created in the primary XML index and the index size is relatively larger; if there are few tags and large values, then few rows are created in the primary XML index and the index size is closer to the data size. Take this into consideration when planning disk space. This is because the node table contains explicit representations of information (such as the path and node number) that is a different representation of information inherent in the structure of the XML document itself.

Properties in BLOB

- ◆ Core attributes in normal columns
- ◆ All attributes for each type are stored in a customized format (varbinary(max))
- ◆ BLOB manipulation is *likely* to be done with SQLCLR
- ◆ Structure *CANNOT* be indexed with relational indexes
- ◆ Finding specific properties require finding the property within an offset table or you need an offset array – which can get overly complicated (hence SQLCLR)

DocID	ALL Values in a varchar(max)
1	Customizable format...
2	Customizable format...
3	Customizable format...
4	Customizable format...
5	Customizable format...
6	Customizable format...
7	Customizable format...

So, here's what I want you to have!

(wait, I thought this was bad?)

DocID	DocName	DocType	DocSize	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19	P20	P21	P22	P23	P24	P25	P26
1	abcx	1	1,234	5	6	8	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	
2	defy	1	54	8	3	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	
3	ghiz	1	480	3	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	
4	jklb	2	340	null	null	null	a	c	r	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	
5	mnoa	2	45,896	null	null	null	z	null	o	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	
6	pqrz	2	123	null	null	null	b	r	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	
7	stuy	3	4,589	null	null	null	null	null	null	n	8	null	4	e	null	null	null	null	null	null	null	null	null	null	null	null	null	null	
8	vwxx	3	1,024	null	null	null	null	null	null	5	5	2	g	d	null	null	null	null	null	null	null	null	null	null	null	null	null	null	
9	yzau	3	56	null	null	null	null	null	null	3	z	null	8	f	null	null	null	null	null	null	null	null	null	null	null	null	null	null	
10	bcdt	4	237	null	null	null	null	null	null	null	null	null	null	a	b	c	d	e	null	null	null	null	null	null	null	null	null	null	
11	efgs	4	593	null	null	null	null	null	null	null	null	null	null	f	g	h	i	j	null	null	null	null	null	null	null	null	null	null	
12	hijr	4	457	null	null	null	null	null	null	null	null	null	null	k	l	m	n	o	null	null	null	null	null	null	null	null	null	null	
13	klmq	4	135	null	null	null	null	null	null	null	null	null	null	p	q	r	s	t	null	null	null	null	null	null	null	null	null	null	
14	nopp	5	75	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	7	null	null	null	null	null	null	null	
15	qrso	5	1,345	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	3	5	null	null	null	null	null	null	null	
16	tuvn	5	576,457	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	5	null	null	null	null	null	null	null	
17	wxym	6	24	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	1	2	null	null	null	null	null	
18	zabl	6	65	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	4	3	null	null	null	null	null	null	
19	cdek	7	23,523	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	z	e	null	null	null	null	
20	fghj	7	56	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	b	r	null	null		
21	ijki	7	24	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	a	s			
22	lmnh	7	6,457	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	a	r	e	null	b	l		
23	opqg	7	23	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	a	c	k	null	w	h		
24	rstf	7	757	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	i	t	e	null	h	o		
25	uvwe	7	234,234	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	r	s	e	s	null	null			

Why is this OK now?

- ◆ The rows will **NOT** be really wide
 - ◆ Columns of type sparse take *0* bytes if the value is NULL
 - ◆ The maximum number of columns in SQL Server 2008 supports is 30,000
- ◆ Performance will **NOT** be compromised
 - ◆ The maximum number of indexes in SQL Server 2008 supports is 1,000 (in fact, you might *want* even more – if you're using sparse columns) but these indexes are true relational indexes and very lean
 - ◆ Performance (with the right design → using an XML column set) can be excellent
- ◆ It still *looks* wrong but sparse columns (and filtered indexes) **solve** all of the old problems...

Sparse Columns: What are they?

- ◆ SPARSE is a new column attribute in SQL Server 2008
- ◆ NULL values in a column defined as sparse require **zero** bytes of storage
 - ◆ Compared to *at least* 1-bit in SQL Server 2005 (and probably more depending on the column type and nullability)
- ◆ Trade-offs:
 - ◆ Access non-null SPARSE columns is more costly
 - ◆ Storing non-null SPARSE columns takes 4 extra bytes per value
- ◆ Sparse columns can be grouped together into a column set for faster access (returned as XML)
- ◆ Can be efficiently accessed using filtered indexes

Sparse Column: When To Use?

- ◆ When you need a flexible schema (the ability to add new “attributes”)
- ◆ When the large majority of attributes have NULL values for the large majority of rows
(remember the picture – doesn't have to be so mutually exclusive but make sure to run the numbers)
- ◆ Recommendation is to use when net space savings can be 20-40%
- ◆ Books Online has a table showing how much data would have to be null for a particular type to achieve 40% saving
 - ◆ See ‘Sparse columns’ in Books Online index

Sparse Columns: Column Sets

- ◆ A way to return ALL of the sparse columns as a unit to the application (*much* faster)
- ◆ Added as a non-persisted computed column to the table and returned as an XML column
(Note: this column does not limit ONLINE operations for the clustered index as a traditional LOB column will. This LOB column is a virtual column.)
- ◆ *Only one can exist per table – and it must be created before any sparse columns are added or at the time of table creation (you cannot add this later)*
- ◆ In this release, the column set for a table ALWAYS operates over all sparse columns in the table and new sparse columns are automatically added into the column set

Sparse Columns: Column Sets

- ◆ Selecting *the* column set returns an XML value containing all non-null sparse column values for that row
- ◆ Inserts and updates to multiple sparse columns can be done using either the XML format, or by specifying a regular column list
- ◆ Check which tables have column sets with:

```
SELECT OBJECTPROPERTY  
      (OBJECT_ID ( 'TableName' ),  
       'TableHasColumnSet ' )
```

Sparse Columns: Storage Internals

- ◆ Row with no sparse columns has no storage for them
 - ◆ No entry in the NULL bitmap or fixed length portion
- ◆ Row with sparse values
 - ◆ Adds 6 bytes to the row size + 4 bytes per non-null column + the values
- ◆ Compared to other column types:
 - ◆ Fixed-width columns are always complete storage (plus 1 bit in the NULL block):
 - ◆ INT = 4 bytes (plus 1 bit)
 - ◆ CHAR(10) = 10 bytes (plus 1 bit)

Sparse Columns: Storage Internals

- ◆ Non-NULL variable-width columns are actual length + 2 bytes (plus 1 bit in the NULL block)
NOTE: NULL variable-width columns may take 0 bytes but only in very special cases. This can happen when the NULL values are at the end of the variable block array; then it does not need to be full populated.
- ◆ See these posts on storage internals:
 - ◆ <http://www.sqlskills.com/BLOGS/KIMBERLY/post/Column-order-doesnt-matter-generally-but-IT-DEPENDS!.aspx>
 - ◆ <http://www.sqlskills.com/BLOGS/PAUL/category/On-Disk-Structures.aspx>
 - ◆ [Search Engine Q&A #27: How does the storage engine find variable-length columns?](#)

Sparse Columns: Limitations

- ◆ These types cannot be SPARSE:
 - ◆ Auto-populated columns:
 - ◆ ROWGUIDCOL columns
 - ◆ IDENTITY columns
 - ◆ Timestamp
 - ◆ Columns with DEFAULT values
 - ◆ Computed columns (although they can contain sparse columns)
 - ◆ Special types:
 - ◆ Spatial data types (*maybe in a future release?*)
 - ◆ UDTs (*maybe in a future release?*)
 - ◆ varbinary(max) FILESTREAM (*not internal anyway*)
 - ◆ Legacy LOB (n/text, image) (*eventually to be removed*)

Sparse Columns: Other Limitations

- ◆ A sparse column cannot be:
 - ◆ Part of a clustered index key
 - ◆ Part of a primary key constraint
 - ◆ Part of a unique constraint
(however, you can create a unique index on one)
 - ◆ The partition key of a table (w/clustered index or heap)
- ◆ Table/Row Conversions
 - ◆ What you really need to know:
 - ◆ If you want an EXISTING table to have sparse columns DO NOT ADD THEM without some planning!
 - ◆ Why?
 - ◆ You cannot add an XML column set to an existing table that already has sparse columns
 - ◆ You might create some massive fragmentation and some weird results with the row structure conversion
 - ◆ A better approach, create a new table – and INSERT/SELECT

Sparse Columns & Other Features

- ◆ Data Compression
 - ◆ Data compression cannot be used with sparse columns
- ◆ Replication
 - ◆ Transactional replication supports sparse columns but not column sets
 - ◆ Merge replication does not support sparse columns at all
- ◆ Change Tracking
 - ◆ Supports both sparse columns and column sets
 - ◆ Does not track which columns were updated for column sets
- ◆ Change Data Capture
 - ◆ Supports sparse columns but not column sets

What about searching?

- ◆ Name/Value Pairs
 - ◆ Quickly becomes a very large table/index – requires a lot of maintenance and becomes very fragmented
- ◆ Normalized Tables
 - ◆ Indexes for each table must be created
 - ◆ Application needs to know which tables to search
- ◆ Properties in XML
 - ◆ Can add indexes but XML indexes are very expensive
- ◆ Properties in BLOB
 - ◆ No way to index these properties using SQL Server indexes – can be very slow to search/analyze

Searching Sparse Columns

- ◆ Using nonclustered indexes!
- ◆ Won't there be a lot of wasted space in an index or can index columns be sparse as well?
 - ◆ Index columns do support the SPARSE attribute...
 - ◆ NULL values in a SPARSE column that's indexed – are “materialized” in the index...
 - ◆ SQL Server 2008 allows filters on index definitions
 - ◆ WHERE SparseColumn IS NOT NULL
- ◆ Will we be able to index every property?
 - ◆ NO, but SQL Server increased the nonclustered index limit from 249 (SQL 2005) to 999 (SQL 2008)
 - ◆ This still may not be enough but... it gives you hundreds of “properties” which you can VERY efficiently index

Indexing Sparse Columns

Background: Understanding nonclustered indexes

- ◆ Traditionally, a nonclustered index stores (in the leaf level) *something* for every row of the base table
 - ◆ If the table has 10,000 rows then EVERY nonclustered index (leaf-level) has 10,000 “rows”
- ◆ There are many benefits (i.e. covering) to this approach but for a column with a significant number of NULLs and where we’re creating a “lookup” index – we just don’t want the NULL values to be represented... enter, filtered indexes