

Study Group 70-432

Tables

Designing SQL Server Indexes

Full Text Search

Łukasz Łysik

llysik@gmail.com

Agenda

- ◆ Tables
 - ◆ Creating tables (with Data Types)
 - ◆ Constraint
- ◆ Indexes
 - ◆ Architectur
 - ◆ Designing
 - ◆ Maintenance
- ◆ Full text indexes
 - ◆ Creating and populating
 - ◆ Querying
 - ◆ Managing

Oznaczenia

- ◆ **Note** – informacja z okienka „Note”
- ◆ **Exam Tip** – informacja z okienka „Exam Tip”
- ◆ **Tip** – zalecenia z tekstu

Creating Tables / Schemas

- ◆ Used for:
 - ◆ security reasons
 - ◆ grouping tables
- ◆ Container that owns all object within a database
- ◆ **Tip:** it is better to create tables within multiple schemas than in multiple databases

```
CREATE SCHEMA <schema name> AUTHORIZATION <owner name>
```

- ◆ <owner name> - **SQL Server principle**

Creating Tables / Schemas

- ◆ **Note:** It is recommended that you do not create tables and view or assign permissions within a CREATE SCHEMA statement. Any CREATE SCHEMA statement that is executed must be in a separate batch.

```
CREATE SCHEMA Sprockets AUTHORIZATION Annik
CREATE TABLE NineProngs (source int, cost int)
GRANT SELECT TO Mandar
DENY SELECT TO Prasanna;
```

Creating Tables / Data Types / Numeric Data Types

Numeric Data Types

DATA TYPE	RANGE OF VALUES	STORAGE SPACE
<i>TINYINT</i>	0 to 255	1 byte
<i>SMALLINT</i>	-32,768 to 32,767	2 bytes
<i>INT</i>	-231 to 231-1	4 bytes
<i>BIGINT</i>	-263 to 263-1	8 bytes
<i>DECIMAL(P,S)</i> and <i>NUMERIC(P,S)</i>	-1038+1 to 1038-1	5 to 17 bytes
<i>SMALLMONEY</i>	-214,748.3648 to 214,748.3647	4 bytes
<i>MONEY</i>	-922,337,203,685,477.5808 to 922,337,203,685,477.5807	8 bytes
<i>REAL</i>	-3.438 to -1.1838, 0, and 1.1838 to 3.438	4 bytes
<i>FLOAT(N)</i>	-1.79308 to -2.23308, 0, and 2.23308 to 1.79308	4 bytes or 8 bytes

Creating Tables / Data Types / **Numeric Data Types**

- ◆ **Note:** NUMERIC and DECIMAL are exactly equivalent
- ◆ The MONEY and SMALLMONEY data types are designed specifically to store monetary values with a maximum of four decimal places.
- ◆ FLOAT: n = mantissa (the number of digits stored after the decimal)
 - ◆ mantissa between 1 and 24 → 4 bytes of storage.
 - ◆ mantissa between 25 and 53 → 8 bytes of storage.
- ◆ **Note:** FLOAT and REAL are classified as approximate numerics. Moving between Intel chipset ↔ AMD chipset can produce different results.
- ◆ DECIMAL: P – precision, S - scale

PRECISION	STORAGE SPACE
1 to 9	5 bytes
10 to 19	9 bytes
20 to 28	13 bytes
29 to 38	17 bytes

Creating Tables / Data Types / **Character Data Types**

DATA TYPE	STORAGE SPACE
<i>CHAR(n)</i>	Non-Unicode, 1 byte per character defined by <i>n</i> , up to a maximum of 8,000 bytes.
<i>VARCHAR(n)</i>	Non-Unicode, 1 byte per character stored up to a maximum of 8,000 bytes
<i>NCHAR(n)</i>	Unicode, 2 bytes per character defined by <i>n</i> , up to a maximum of 4,000 bytes
<i>NVARCHAR(n)</i>	Unicode, 2 bytes per character stored up to a maximum of 4,000 bytes

- ◆ **VARCHAR(MAX), NVARCHAR(MAX)** – up to 2 GB of data

Creating Tables / Data Types / **Date and Time Data**

DATA TYPE	RANGE OF VALUES	ACCURACY	STORAGE SPACE
<i>SMALLDATETIME</i>	01/01/1900 to 06/06/2079	1 minute	4 bytes
<i>DATETIME</i>	01/01/1753 to 12/31/9999	0.00333 seconds	8 bytes
<i>DATETIME2</i>	01/01/0001 to 12/31/9999	100 nanoseconds	6 to 8 bytes
<i>DATETIMEOFFSET</i>	01/01/0001 to 12/31/9999	100 nanoseconds	8 to 10 bytes
<i>DATE</i>	01/01/0001 to 12/31/9999	1 day	3 bytes
<i>TIME</i>	00:00:00.0000000 to 23:59:59.9999999	100 nanoseconds	3 to 5 bytes

- ◆ *DATETIME2* provides better precision
- ◆ *DATETIMEOFFSET* allows you to store a time zone for applications that need to localize dates and times.

Creating Tables / Data Types / **Binary Data**

DATA TYPE	RANGE OF VALUES	STORAGE SPACE
<i>BIT</i>	Null, 0, and 1	1 bit
<i>BINARY</i>	Fixed-length binary data	Up to 8,000 bytes
<i>VARBINARY</i>	Variable-length binary data	Up to 8,000 bytes

- ◆ **VARBINARY(MAX)** – up to 2 BG of data storage

Creating Tables / Data Types / **XML Data Type**

- ◆ Limitations:
 - ◆ 2 GB
 - ◆ 128 levels within a document
- ◆ XML data type natively understands the structure of XML data
- ◆ Additional validation – XML schema
- ◆ XML schemas are stored in *schema collection*

```
CREATE XML SCHEMA COLLECTION ProductAttributes AS  
'<xsd:schema . . . >'
```

Creating Tables / Data Types / **Spatial Data Types**

INSTANCE	DESCRIPTION
<i>Point</i>	Has <i>x</i> and <i>y</i> coordinates, with optional elevation and measure values.
<i>LineString</i>	A series of points that defines the start, end, and any bends in the line, with optional elevation and measure values.
<i>Polygon</i>	A surface defined as a sequence of points that defines an exterior boundary, along with zero or more interior rings. A polygon has at least three distinct points.
<i>GeometryCollection</i>	Contains one or more instances of other geometry shapes, such as a <i>Point</i> and a <i>LineString</i> .
<i>MultiPolygon</i>	Contains the coordinates for multiple <i>Polygons</i> .
<i>MultiLineString</i>	Contains the coordinates of multiple <i>LineStrings</i> .
<i>MultiPoint</i>	Contains the coordinates of multiple <i>Points</i> .

- ◆ GEOMETRY (table above)
- ◆ GOGRAPHY (latitude, longitude)

Creating Tables / Data Types / **HIERARCHYID** Data Type

- ◆ The *HIERARCHYID* data type is used to organize hierarchical data.

```
SELECT OrgNode.ToString() AS Text_OrgNode,  
OrgNode, OrgLevel, EmployeeID, EmpName, Title  
FROM HumanResources.EmployeeOrg
```

	Text_OrgNode	OrgNode	OrgLevel	EmployeeID	EmpName	Title
1	/	0x	0	6	David	Marketing Manager
2	/1/	0x58	1	46	Sariya	Marketing Specialist
3	/1/1/	0x5AC0	2	269	Wanida	Marketing Assistant
4	/2/	0x68	1	271	John	Marketing Specialist
5	/2/1/	0x6AC0	2	272	Mary	Marketing Assistant
6	/3/	0x78	1	119	Jill	Marketing Specialist

Creating Tables / Data Types / **ntext**, **text**, **image**

- ◆ „**ntext**, **text**, and **image** data types will be removed in a future version of Microsoft SQL Server. Avoid using these data types in new development work, and plan to modify applications that currently use them. Use **nvarchar(max)**, **varchar(max)**, and **varbinary(max)** instead.”



Creating Tables / **Column Properties**

Creating Tables / Column Properties / **Nullability**

- ◆ NULL / NOT NULL
- ◆ Domyślna wartość ANSI_NULL_DEFAULT

Creating Tables / Column Properties / **COLLATE**

- ◆ Instance level
- ◆ Database level
- ◆ Table level
- ◆ Column level

Creating Tables / Column Properties / **IDENTITY**

- ◆ You cannot update a column with the identity property.
 - ◆ *SET IDENTITY_INSERT* <table name> ON
- ◆ Columns with any numeric data type, except float and real
- ◆ Seed and increment
 - ◆ Seed can be changed using *DBCC CHECKIDENT* command
- ◆ Only a single identity column in a table is allowed
- ◆ Frequently are unique, but they do not have to be

Creating Tables / Column Properties / **ROWGUIDCOL**

- ◆ Used mainly by merge replication
- ◆ To ensure that only a single column of this type exists and that the column has a *UNIQUEIDENTIFIER* data type.

Creating Tables / Column Properties / **FILESTREAM**

- ◆ You are no longer restricted to the 2-GB limit on BLOBs.
- ◆ When you back up the database, all the files are backed up at the same time
- ◆ Only for VARBINARY(MAX)

```
FILEGROUP GrupaFileStream CONTAINS FILESTREAM  
( FILENAME = N'D:\TECHNET\FileStreamCatalog',
```

```
FILESTREAM_ON FileStreamGroup1
```

- ◆ ***EXAM TIP*** A FILEGROUP designated for FILESTREAM storage is off-line and inaccessible within a Database Snapshot. Database Mirroring cannot be applied.

Creating Tables / Column Properties / **NOT FOR REPLICATION**

- ◆ For column with IDENTITY
- ◆ SQL Server does not reseed the identity column when the replication engine is applying changes

Creating Tables / Column Properties / **SPARSE**

- ◆ Designed to optimize storage space for columns with a large percentage of NULLs.
- ◆ NULL in the column – no storage space is consumed
- ◆ Non-NULL in the column – 4 extra bytes of storage
- ◆ You cannot apply the *SPARSE* property to
 - ◆ Columns with the *ROWGUIDCOL* or *IDENTITY* property
 - ◆ *TEXT*, *NTEXT*, *IMAGE*, *TIMESTAMP*, *GEOMETRY*, *GEOGRAPHY*, or user-defined data types
 - ◆ A *VARBINARY(MAX)* with the *FILESTREAM* property
 - ◆ A computed column of a column with a rule or default bound to it
 - ◆ Columns that are part of either a clustered index or a primary key
 - ◆ A column within an *ALTER TABLE* statement

Creating Tables / Column Properties / **SPARSE**

- ◆ **NOTE** If the maximum size of a row exceeds 4,009 bytes, you cannot issue an ALTER statement to either change a column to SPARSE or add an additional SPARSE column. During the ALTER, each row is recomputed by writing a second copy of the row on the same data page.

Possible workarounds:

- ◆ Reduce the data within a row so that the maximum row size is greater than 4,009 bytes
- ◆ Create a new table, copy all the data to the new table, drop the old table, and then rename the newly created table
- ◆ Export the data, truncate the existing table, make the changes, and import the data back into the table

Creating Tables / **Computed Columns**

- ◆ Columns that are calculated based on other columns in the row.
- ◆ Only the definition of the calculation is stored
- ◆ *PERSISTED*
 - ◆ the result of the calculation is stored in the row
 - ◆ The value is updated anytime data that the calculation relies upon is changed.

Creating Tables / Row and Page Compression

- ◆ Row-level compression - more rows on a page – less pages
- ◆ Page-level compression
 - ◆ Entire page is compressed
 - ◆ When SQL Server applies page compression to a heap (a table without a clustered index), it compresses only the pages that currently exist in the table.
 - ◆ Compression: BULK INSERT or INSERT INTO. . .WITH (TABLOCK)
 - ◆ No-compression: BCP or an INSERT
 - ◆ ALTER TABLE. . .REBUILD with PAGE compression
- ◆ VARCHAR(MAX), NVARCHAR(MAX), and VARBINARY(MAX) cannot be compressed.

Creating Tables / **Creating Tables**

- ◆ Maximum 1024 columns (but using column set definition and sparse columns it can be 30 000 columns)
- ◆ Temporary tables:
 - ◆ Local #table and global ##table
 - ◆ Stored in tempdb, dropped when connection that created is finished

Implementing Constraints / **Primary Keys**

- ◆ Only a single primary key for a table
- ◆ Column(s) that uniquely identify every row
- ◆ All columns within the primary key are NOT NULL.
- ◆ Clustered or nonclustered
- ◆ ***Exam Tip:*** The default option for a primary key is clustered. When a clustered primary key is created on a table that is compressed, the compression option is applied to the primary key when the table is rebuilt.

Implementing Constraints / **Foreign Keys**

- ◆ For data integrity
- ◆ For multicolumn primary key all the columns must exist in the child table
- ◆ CASCADE - extremely bad idea

Implementing Constraints / **Unique Constraints**

- ◆ Column or columns for which the values must be unique within the table
- ◆ NULLs are allowed
- ◆ *Exam Tip:* single row within the table is allowed to have a NULL within unique column

Implementing Constraints / **Default Constraints**

- ◆ New rows added with an INSERT, BCP, or BULK INSERT statement.
- ◆ For either NULL or NOT NULL columns

Implementing Constraints / **Check Constraints**

- ◆ Limit the range of values within a column
- ◆ Column level – not allowed to reference other columns in the table
- ◆ Table level – only columns in the table

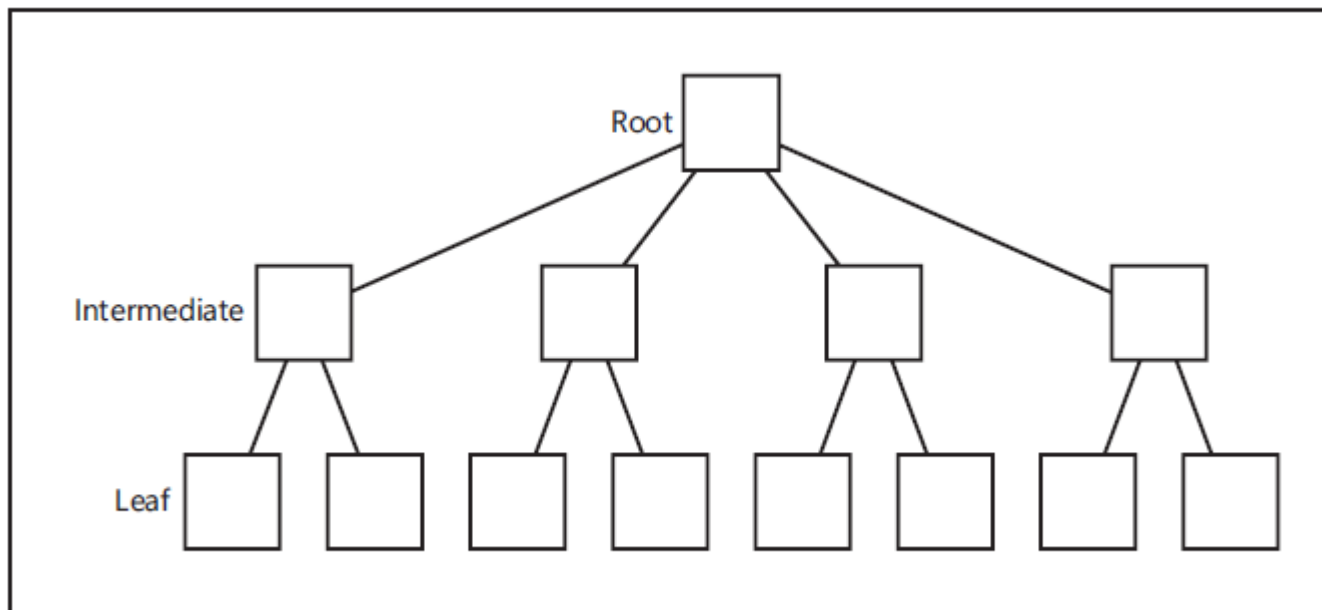
```
CHECK (Column1 LIKE ' [0-9] [0-9] [0-9] - [0-9] [0-9] - [0-9] [0-9] [0-9] [0-9] ')
```



Designing SQL Server Indexes

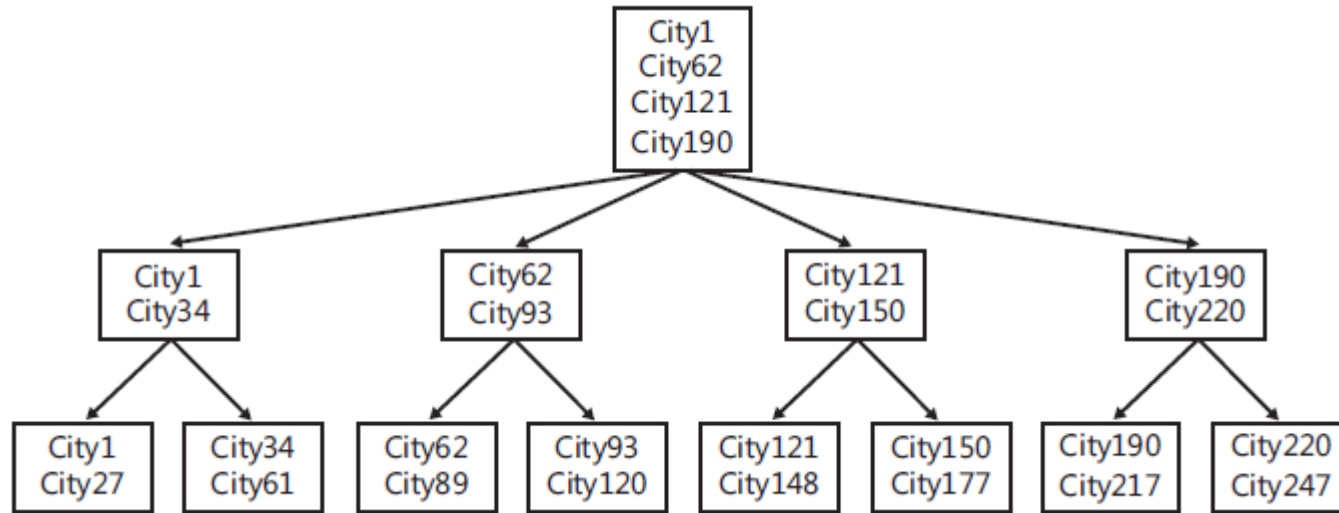
Designing SQL Server Indexes / **Index Architecture**

- ◆ **Balanced tree (B-tree) – always symmetrical**



- ◆ **NOTE** Pages in SQL Server can store up to 8,060 bytes of data. An index with an **INT** data type can store 2,015 values on a single page.

Designing SQL Server Indexes / Index Architecture



Example: City132

As soon as it read the entry for City133, the query returns with no results found.

Designing SQL Server Indexes / **Index Architecture**

- ◆ Index levels: The number of levels is determined by simple mathematics.
- ◆ Data page size: 8,192 bytes (8,060 bytes of actual user data).
- ◆ Example:
 - INT column: 4 bytes
 - 1 200 rows of data = 4 800 bytes of storage = only single page
 - The page can store 2 015 rows
 - On 2 016th: two more pages are used (no intermediate level).
 - Next level: 4,060,225 rows
- ◆ **NOTE** SQL Server writes any new entries into the correct sorted location when page splitting. This can cause rows to move between pages, and page splits can occur at any level within the storage structure.

Designing SQL Server Indexes / **Index Architecture**

- ◆ ***EXAM TIP*** If you are creating an index on a sparse column, you should use a filtered index to create the most compact and efficient index possible.

Designing SQL Server Indexes / Designing Indexes / **Clustered Indexes**

- ◆ With a maximum of 16 columns
- ◆ The maximum size of the index key is 900 bytes.
- ◆ Rearranges data (only 1 clustered index allowed)
- ◆ Provide sort order but does not provide a physical sort order.
- ◆ Table without a clustered index is referred to as a *heap*.
- ◆ **NOTE** If you specify a different filegroup in the FILESTREAM_ON clause than where the FILESTREAM data is currently located, all the FILESTREAM data will be moved to the newly specified filegroup during the creation of the clustered index.
- ◆ Primary Key = unique, clustered index
- ◆ ON <partition> - leaf level with data on the same partition

Designing SQL Server Indexes / Designing Indexes / **Nonclustered Indexes**

- ◆ Multiple indexes can be created (max. 1000)
- ◆ The same restriction as clustered index.
- ◆ If clustered index exists
 - ◆ index points to clustered index
 - ◆ Else index points to row of data in the table
- ◆ A rule of thumb: no more than 5 indexes for OLTP databases (more for data warehouses)
- ◆ Included columns - become part of the index at only the leaf level

Designing SQL Server Indexes / Designing Indexes / Nonclustered Indexes

- ◆ Query optimizer – decides if to use index
- ◆ Uses *histogram*
- ◆ *Selectivity* :
 - ◆ Big – big number of unique values
 - ◆ Small – small number of unique values
- ◆ QO chooses most selective indexes.
- ◆ Filtered index - keys matching the WHERE clause are added to the index
 - ◆ Nonclustered index
 - ◆ Cannot be created on computed columns
 - ◆ Cannot undergo data type conversions

Designing SQL Server Indexes / Designing Indexes / Nonclustered Indexes / Index Options

- ◆ FILLFACTOR – space left on the **leaf** level during creation or rebuild
- ◆ PAD_INDEX – FILLFACTOR for intermediate-level page(s) and root page
- ◆ SORT_IN_TEMPDB – sort operations on temporary tables are performed in *tempdb*
- ◆ IGNORE_DUP_KEY – rows that produce duplicate generate a warning (not an error) and only those are rejected.
- ◆ WITH ONLINE – rebuild indexes
 - ◆ OFF – lock on the table (default)
 - ◆ ON – version store in *tempdb* (**EXAM TIP** Only SS 2008 Enterprise)

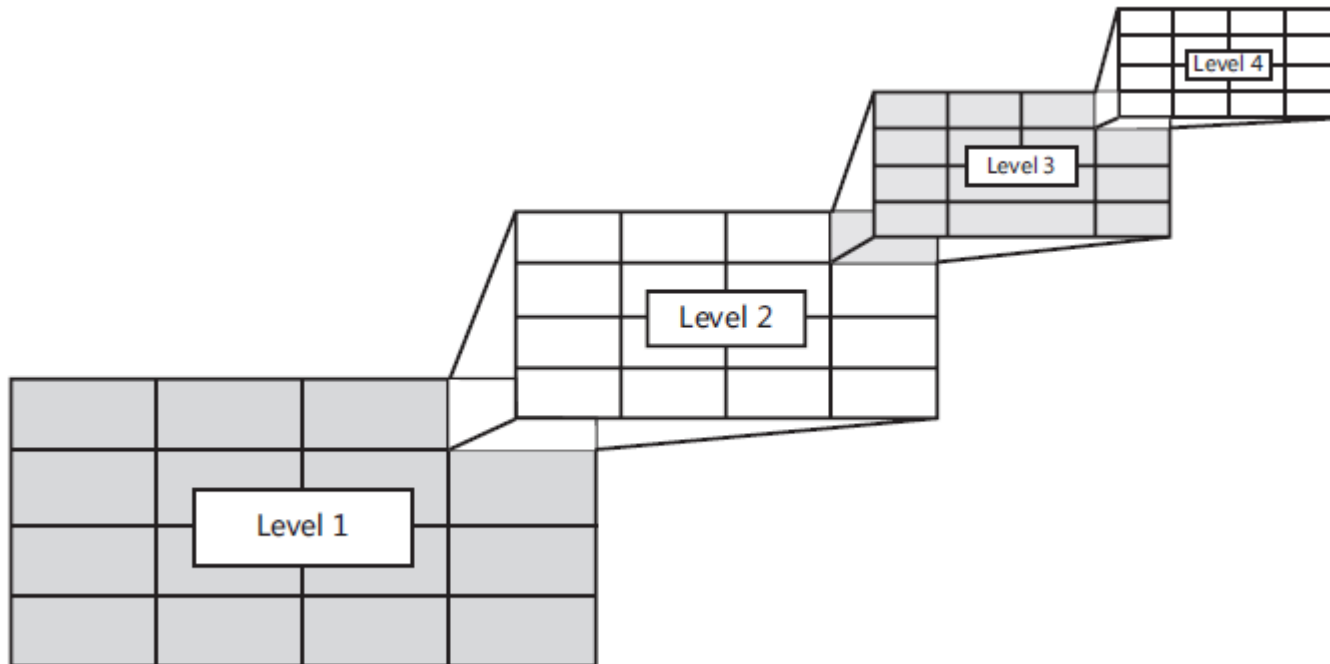
Designing SQL Server Indexes / Designing Indexes / Nonclustered Indexes / XML Indexes

- ◆ **Primary**
 - ◆ Against all the nodes within XML column
 - ◆ Tied to clustered index
 - ◆ Required for secondary index
- ◆ **Secondary**
 - ◆ Can be create on PATH, VALUE or PROPERTY

```
CREATE [ PRIMARY ] XML INDEX index_name
ON <object> ( xml_column_name )
[ USING XML INDEX xml_index_name
[ FOR { VALUE | PATH | PROPERTY } ] ]
[ WITH ( <xml_index_option> [ ,...n ] ) ][ ; ]
```

Designing SQL Server Indexes / Designing Indexes / Nonclustered Indexes / Spatial Indexes

- ◆ Tessellation – transformation of two-dimensional system to linear chain



Designing SQL Server Indexes / Designing Indexes / Nonclustered Indexes / Spatial Indexes

- ◆ Rules
 - ◆ Covering – in an object completely covers a cell the cell is not thesaled
 - ◆ Cell-per-object - This rule enforces the *CELLS_PER_OBJECT* parameter for the *spatial* index for levels 2, 3, and 4 of the grid hierarchy.
 - ◆ Deepest cell – records only the lowest-level cells that have been thesalled
- ◆ Bounding box

Designing SQL Server Indexes / **Maintaining Indexes**

- ◆ Index fragmentation
 - ◆ The removal of the value creates a „hole”
 - ◆ Data changes creates another hole
- ◆ FILLFACTOR – the percentage of free space reserved
 - ◆ 75% means that 25% is left empty
- ◆ Defragmenting an Index
 - ◆ Rebuild – rebuild all levels (uses FILLFACTOR)
 - ◆ Reorganize – only at the leaf level (always an online operation)
- ◆ Disabling an index
 - ◆ If clustered index is disabled, the entire table is inaccessible
 - ◆ To enable it must be rebuilt

Question

4.3.1. You are in charge of building the process that loads approximately 150 GB of data into the enterprise data warehouse every month. Every table in your data warehouse has at least eight indexes to support data analysis routines. You want to load the data directly into the tables as quickly as possible. Which operation provides the best performance improvement with the least amount of administrative effort?

- A.** Use a *BULK INSERT* command.
- B.** Drop and re-create the indexes.
- C.** Disable and enable the indexes.
- D.** Use Integration Services to import the data.



Full Text Indexing

Full Text Indexing / Full Text Catalogs

- ◆ Full text indexes are stored in *full text catalogs*.
- ◆ Each full text catalog contains one or more full text indexes

```
CREATE FULLTEXT CATALOG catalog_name
[ON FILEGROUP filegroup ]
[IN PATH 'rootpath']
[WITH <catalog_option>]
[AS DEFAULT]
[AUTHORIZATION owner_name ]
```

```
<catalog_option>::=
```

```
ACCENT_SENSITIVITY = {ON|OFF}
```

Full Text Indexing / Full Text Catalogs

- ◆ *EXAM TIP*
 - ◆ Prior to SQL Server 2008 – stored in a directory structure on the operating system
 - ◆ In SQL Server 2008 – stored within the database
- ◆ FILEGROUP – which filegroup to use to store full text indexes
- ◆ IN PATH – deprecated
- ◆ ACCENT_SENSITIVITY - For example, 'a' is not equal to 'á'. If you change this option, you need to rebuild all the full text indexes within the catalog.
- ◆ AS DEFAULT – if the catalog is default
- ◆ AUTHORIZATION – owner of the catalog. Owner must have TAKE OWNERSHIP permission.

Full Text Indexing / **Full Text Catalogs**

- ◆ **NOTE** Full text catalog and relational data in the same catalog – possible but not recommended.

Full Text Indexing / Full Text Indexes

- ◆ CHAR/VARCHAR: full text engine parse the data directly and build an appropriate index.
- ◆ XML a special processor is loaded
- ◆ VARBINARY:
 - ◆ Extra column that designates the type of document
 - ◆ SQL Server 2008 ships with 50 filters
- ◆ Helper services
 - ◆ Word breakers
 - ◆ Stemmers
 - ◆ Stop words (*the, a, an*)

Full Text Indexing / Full Text Indexes

```
CREATE FULLTEXT INDEX ON table_name
  [ ( { column_name
  [ TYPE COLUMN type_column_name ]
  [ LANGUAGE language_term ]
  } [ ,...n]
  ) ]
KEY INDEX index_name
  [ ON <catalog_filegroup_option> ]
  [ WITH [ ( ) <with_option> [ ,...n] [ ) ] ]
[;]

<with_option> ::=
  {CHANGE_TRACKING [ = ] { MANUAL | AUTO | OFF [, NO
  POPULATION ] }
  | STOPLIST [ = ] { OFF | SYSTEM | stoplist_name }}
```

Full Text Indexing / **Language, Work Breakers, Stemmers**

- ◆ For example, the English language breaks words with a space, whereas languages such as German and French can combine words.
- ◆ The language specification is used to control the specific word breaker and stemmer loaded by the full text indexing engine.
- ◆ The same word breaker and stemmer for the entire full text index and cannot change dynamically based on a type column like you can apply to a *VARBINARY* column.
- ◆ When you have data spanning languages, you should specify a language setting for the most complicated language.
- ◆ When the languages vary widely such as Arabic, Chinese, English, and Icelandic, you should split the data into separate columns based on language.

Full Text Indexing / Full Text Indexes

```
CREATE FULLTEXT INDEX ON table_name
  [ ( { column_name
  [ TYPE COLUMN type_column_name ]
  [ LANGUAGE language_term ]
  } [ ,...n]
  ) ]
KEY INDEX index_name
  [ ON <catalog_filegroup_option> ]
  [ WITH [ ( ] <with_option> [ ,...n] [ ) ] ]
[;]

<with_option> ::=
  {CHANGE_TRACKING [ = ] { MANUAL | AUTO | OFF [, NO
  POPULATION ] }
  | STOPLIST [ = ] { OFF | SYSTEM | stoplist_name }}
```

Question

- 5.1.1.** You are the database administrator at your company. You need to enable the sales support team to perform fuzzy searches on product descriptions. Which actions do you need to perform to satisfy user needs with the least amount of effort? (Choose two. Each forms part of the correct answer.)
- A.** Create a full text catalog specifying the filegroup for backup purposes and the root path to store the contents of the catalog on the file system.
 - B.** Create a full text catalog and specify the filegroup to store the contents of the catalog.
 - C.** Create a full text index on the table of product descriptions for the description column and specify NO POPULATION.
 - D.** Create a full text index on the table of product descriptions for the description column and specify CHANGE_TRACKING AUTO.

Question

- 5.1.2.** You want to configure your full text indexes such that SQL Server migrates changes into the index as quickly as possible with the minimum amount of administrator effort. Which command should you execute?
- A.** ALTER FULLTEXT INDEX ON *<table_name>* START FULL POPULATION
 - B.** ALTER FULLTEXT INDEX ON *<table_name>* START INCREMENTAL POPULATION
 - C.** ALTER FULLTEXT INDEX ON *<table_name>* SET CHANGE_TRACKING AUTO
 - D.** ALTER FULLTEXT INDEX ON *<table_name>* START UPDATE POPULATION

Full Text Indexing / **Querying Full Text Data**

- ◆ CONTAINS
- ◆ FREETEXT
- ◆ CONTAINSTABLE
- ◆ FREETEXTTABLE

Full Text Indexing / Querying Full Text Data / **FREETEXT**

```
FREETEXT ( { column_name | (column_list) | * }  
, 'freetext_string' [ , LANGUAGE language_term ] )
```

```
SELECT ProductDescriptionID, Description  
FROM Production.ProductDescription  
WHERE FREETEXT(Description,N'bike')
```

- ◆ **NOTE** different language will not automatically improve the accuracy of a full text search.
- ◆ **EXAM TIP** All search terms used with full text are Unicode strings. If you pass in a non-Unicode string, the query still works, but it is much less efficient.

Full Text Indexing / Querying Full Text Data / **FREETEXTTABLE**

```
FREETEXTTABLE ( table , { column_name | (column_list) |  
* } , 'freetext_string' [ , LANGUAGE language_term ]  
[ , top_n_by_rank ] )
```

```
SELECT a.ProductDescriptionID, a.Description, b.*  
FROM Production.ProductDescription a  
INNER JOIN FREETEXTTABLE(Production.ProductDescription,  
Description,N'bike') b ON a.ProductDescriptionID = b.  
[Key]  
ORDER BY b.[Rank]
```

Full Text Indexing / Querying Full Text Data / **CONTAINS**

```
SELECT ProductDescriptionID, Description
FROM Production.ProductDescription
WHERE CONTAINS(Description,N' FORMSOF (INFLECTIONAL,ride) ')
```

```
SELECT ProductDescriptionID, Description
FROM Production.ProductDescription
WHERE CONTAINS(Description,N' FORMSOF (THESAURUS,metal) ')
```

- ◆ **NOTE** All thesaurus files are XML files stored in the FTDATA directory underneath your default SQL Server installation path. The thesaurus files are not populated.

Full Text Indexing / Querying Full Text Data / **CONTAINSTABLE**

```
SELECT a.ProductDescriptionID, a.Description, b.*
FROM Production.ProductDescription a INNER JOIN
CONTAINSTABLE (Production.ProductDescription, Description,
N'bike NEAR performance') b ON a.ProductDescriptionID = b.[Key]
ORDER BY b.[Rank]
```

```
SELECT a.ProductDescriptionID, a.Description, b.*
FROM Production.ProductDescription a INNER JOIN
CONTAINSTABLE (Production.ProductDescription, Description,
N'ISABOUT (performance WEIGHT (.8), comfortable WEIGHT (.6),
smooth WEIGHT (.2) , safe WEIGHT (.5), competition WEIGHT
(.5)) ', 10)
b ON a.ProductDescriptionID = b.[Key]
ORDER BY b.[Rank] DESC
```

Question

1. You want to search for two terms based on proximity within a row. Which full text predicates can be used to perform proximity searches? (Choose two. Each forms a separate answer.)
 - A. *CONTAINS*
 - B. *FREETEXT*
 - C. *CONTAINSTABLE*
 - D. *FREETEXTTABLE*

Question

2. You want to perform a proximity search based on a weighting value for the search

arguments. Which options for the *CONTAINSTABLE* predicate should you use?

- A. *FORMSOF* with the *THESAURUS* keyword
- B. *FORMSOF* with the *INFLECTIONAL* keyword
- C. *ISABOUT*
- D. *ISABOUT* with the *WEIGHT* keyword

Full Text Indexing / Querying Full Text Data / **Thesaurus**

- ◆ In FTDATA directory
- ◆ The thesaurus is used only for *CONTAINS* and *CONTAINSTABLE* queries when you specify the *FORMSOF THESAURUS* option.
- ◆ **Replacement** - term or terms that are replaced within the search argument prior to the word breaker tokenizing the argument list.
- ◆ **Expansion** set defines a set of terms that are used to expand upon a search argument.

Full Text Indexing / Querying Full Text Data / Thesaurus

```
<XML ID="Microsoft Search Thesaurus">
<!-- Commented out
  <thesaurus xmlns="x-schema:tsSchema.xml">
    <diacritics_sensitive>0</diacritics_sensitive>
    <expansion>
      <sub>Internet Explorer</sub>
      <sub>IE</sub>
      <sub>IE5</sub>
    </expansion>
    <replacement>
      <pat>NT5</pat>
      <pat>W2K</pat>
      <sub>Windows 2000</sub>
    </replacement>
    <expansion>
      <sub>run</sub>
      <sub>jog</sub>
    </expansion>
  </thesaurus>
-->
</XML>
```

Full Text Indexing / Querying Full Text Data / **Stop Lists**

- ◆ Noise word list
- ◆ The word breaker categorizes the term as uninteresting and removes it.
- ◆ ***EXAM TIP*** You can assume that you will have questions on an exam which are designed to test whether you know the change in behavior for the new version. In SQL Server 2005 and prior versions, you configured noise word files that were in the FTDATA directory. In SQL Server 2008, you configure stop lists that are contained within a database in SQL Server.

Full Text Indexing / Querying Full Text Data / **Populate Full Text Indexes**

- ◆ **FULL** Reprocesses every row from the underlying data to rebuild the full text index completely
- ◆ **INCREMENTAL** Processes only the rows that have changed since the last population; requires a timestamp column on the table
- ◆ **UPDATE** Processes any changes since the last time the index was updated; requires that the `CHANGE_TRACKING` option is enabled for the index and set to `MANUAL`

To initiate population of a full text index, you would execute the *ALTER FULLTEXT INDEX* statement.

Question

- 5.3.1.** You have a list of words that should be excluded from search arguments. Which action should you perform in SQL Server 2008 to meet your requirements with the least amount of effort?
- A.** Create a stop list and associate the stop list to the full text index.
 - B.** Create a noise word file and associate the noise word file to the full text index.
 - C.** Populate a thesaurus file and associate the thesaurus file to the full text index.
 - D.** Parse the inbound query and remove any common words from the search arguments.

Źródło

Mike Hotek

„Microsoft SQL Server 2008 – Implementation and Maintenance”

Training Kit

Dziękuję za uwagę

Łukasz Łysik
llysik@gmail.com