Searching Text

How Strings are Stored

SET ANSI_PADDING { ON | OFF }

- Controls the way SQL Server stores values shorter than the defined size of the column, and the way the column stores values that have trailing blanks in **char**, **varchar**, **binary**, and **varbinary** data.
- When padded, **char** columns are padded with blanks, and **binary** columns are padded with zeros. When trimmed, **char** columns have the trailing blanks trimmed, and **binary** columns have the trailing zeros trimmed.

ANSI_PADDING Setting

Setting	char(*n*) NOT NULL or binary(*n*) NOT NULL	char(*n*) NULL or binary(*n*) NULL	varchar(*n*) or varbinary(*n*)
ON	Pad original value (with trailing blanks for **char** columns and with trailing zeros for **binary** columns) to the length of the column.	Follows same rules as for **char(***n***)** or **binary(***n***)** NOT NULL when SET ANSI_PADDING is ON.	Trailing blanks in character values inserted into **varchar** columns are not trimmed. Trailing zeros in binary values inserted into **varbinary** columns are not trimmed. Values are not padded to the length of the column.

ANSI_PADDING Setting

Setting	**char(*n*) NOT NULL or binary(*n*) NOT NULL**	**char(*n*) NULL or binary(*n*) NULL**	**varchar(*n*) or varbinary(*n*)**
OFF	Pad original value (with trailing blanks for **char** columns and with trailing zeros for **binary** columns) to the length of the column.	Follows same rules as for **varchar** or **varbinary** when SET ANSI_PADDING is OFF.	Trailing blanks in character values inserted into a **varchar** column are trimmed. Trailing zeros in binary values inserted into a **varbinary** column are trimmed. Values are not padded to the length of the column.

ANSI_PADDING Setting

- The SET ANSI_PADDING setting does not affect the **nchar**, **nvarchar**, **ntext**, **text**, **image**, and large value. They always display the SET ANSI_PADDING ON behavior. This means trailing spaces and zeros are not trimmed.
- ANSI_PADDING should always be set to ON.

Example

```
PRINT 'Testing with ANSI_PADDING ON'
SET ANSI_PADDING ON;
GO
CREATE TABLE t1 (
   charcol CHAR(16) NULL,
   varcharcol VARCHAR(16) NULL,
   varbinarycol VARBINARY(8)
);
GO
INSERT INTO t1 VALUES ('No blanks', 'No blanks', 0x00ee);
INSERT INTO t1 VALUES ('Trailing blank ', 'Trailing blank ', 0x00ee00);
SELECT 'CHAR' = '>' + charcol + '<', 'VARCHAR'='>' + varcharcol + '<',
   varbinarycol
FROM t1;
GO
```

Output

```
CHAR                VARCHAR             varbinarycol
------------------ ------------------ ------------------
>No blanks        < >No blanks<        0x00EE
>Trailing blank   < >Trailing blank <  0x00EE00

(2 row(s) affected)
```

Example

```
PRINT 'Testing with ANSI_PADDING OFF';
SET ANSI_PADDING OFF;
GO
CREATE TABLE t2 (
  charcol CHAR(16) NULL,
  varcharcol VARCHAR(16) NULL,
  varbinarycol VARBINARY(8)
);
GO
INSERT INTO t2 VALUES ('No blanks', 'No blanks', 0x00ee);
INSERT INTO t2 VALUES ('Trailing blank ', 'Trailing blank ', 0x00ee00);
SELECT 'CHAR' = '>' + charcol + '<', 'VARCHAR'='>' + varcharcol + '<',
  varbinarycol
FROM t2;
GO
```

Output

```
CHAR               VARCHAR            varbinarycol
------------------ ------------------ ------------------
>No blanks<        >No blanks<        0x00EE
>Trailing blank<   >Trailing blank<   0x00EE

(2 row(s) affected)
```

Comparison of Strings

- When you compare character string data, the logical sequence of the characters is defined by the collation of the character data.
- The result of comparison operators such as < and > are controlled by the character sequence defined by the collation.
- The same SQL Collation might have different sorting behavior for Unicode and non-Unicode data.

String Equivalence

- Trailing blanks are ignored in comparisons; for example, these are equivalent:

WHERE LastName = 'White'

WHERE LastName = 'White '

WHERE LastName = 'White   ‘

independently of the type of LastName

LIKE

- Determines whether a specific character string matches a specified pattern.
- A pattern can include regular characters and wildcard characters. During pattern matching, regular characters must exactly match the characters specified in the character string. However, wildcard characters can be matched with arbitrary fragments of the character string.

Syntax

match_expression [NOT] LIKE *pattern* [ESCAPE *escape_character*]

- *match_expression:* is any valid expression of character data type
- *pattern*: Is the specific string of characters to search for in *match_expression*, and can include the wildcard characters. *pattern* can be a maximum of 8,000 bytes.
- Returns true if *match_expression* matches *pattern*

Wildcard Characters

Wildcard character	Description	Example
%	Any string of zero or more characters.	WHERE title LIKE '%computer %' finds all book titles with the word 'computer' anywhere in the book title.
_ (underscore)	Any single character.	WHERE au_fname LIKE '_ean' finds all four-letter first names that end with ean (Dean, Sean, and so on).

Wildcard Characters

Wildcard character	Description	Example
[ ]	Any single character within the specified range ([a-f]) or set ([abcdef]).	WHERE au_lname LIKE '[C-P]arsen' finds author last names ending with arsen and starting with any single character between C and P, for example Carsen, Larsen, Karsen, and so on.
[^]	Any single character not within the specified range ([^a-f]) or set ([^abcdef]).	WHERE au_lname LIKE 'de[^l]%' all author last names starting with de and where the following letter is not l.

Syntax

- *escape_character:* Is a character that is put in front of a wildcard character to indicate that the wildcard should be interpreted as a regular character and not as a wildcard.
- *escape_character* is a character expression that has no default and must evaluate to only one character.

Unicode Pattern Matching

- LIKE supports ASCII pattern matching and Unicode pattern matching.
- When all arguments (*match_expression*, *pattern*, and *escape_character*, if present) are ASCII character data types, ASCII pattern matching is performed.
- If any one of the arguments are of Unicode data type, all arguments are converted to Unicode and Unicode pattern matching is performed.
- When you use Unicode data (**nchar** or **nvarchar** data types) with LIKE, trailing blanks in the expression to be matched are significant; however, for non-Unicode data, trailing blanks are not significant.
- Unicode LIKE is compatible with the SQL-92 standard.

Example

```
-- ASCII pattern matching with char column
CREATE TABLE t (col1 char(30));
INSERT INTO t VALUES ('Robert King');
SELECT *
FROM t
WHERE col1 LIKE '% King'   -- returns 1 row
```

Example

```
-- Unicode pattern matching with nchar column
CREATE TABLE t (col1 nchar(30));
INSERT INTO t VALUES ('Robert King');
SELECT *
FROM t
WHERE col1 LIKE '% King'   -- no rows returned

-- Unicode pattern matching with nchar column and RTRIM
CREATE TABLE t (col1 nchar (30));
INSERT INTO t VALUES ('Robert King');
SELECT *
FROM t
WHERE RTRIM(col1) LIKE '% King'   -- returns 1 row
```

RTRIM

RTRIM **(** *character_expression* **)**

– Returns a character string after truncating all trailing blanks.

Remarks

- When you perform string comparisons by using LIKE, all characters in the pattern string are significant. This includes leading or trailing spaces. If a comparison in a query is to return all rows with a string LIKE 'abc ' (abc followed by a single space), a row in which the value of that column is ‘abc’ (abc without a space) is not returned.
- However, trailing blanks, in the expression to which the pattern is matched, are ignored in ASCII pattern matching. If a comparison in a query is to return all rows with the string LIKE 'abc' (abc without a space), all rows that start with abc and have zero or more trailing blanks are returned.

Remarks

- A string comparison using a pattern that contains **char** and **varchar** data may not pass a LIKE comparison because of how the data is stored.

Example

```
USE AdventureWorks;
GO
CREATE PROCEDURE FindEmployee @EmpLName char(20)
AS
SELECT @EmpLName = RTRIM(@EmpLName) + '%';
SELECT c.FirstName, c.LastName, a.City
FROM Person.Contact c JOIN Person.Address a ON c.ContactID =
    a.AddressID
WHERE c.LastName LIKE @EmpLName;
GO
EXEC FindEmployee @EmpLName = 'Barb';
GO
```

Example

- In the FindEmployee procedure, no rows are returned because the **char** variable (@EmpLName) contains trailing blanks whenever the name contains fewer than 20 characters.
- Because the LastName column is **varchar**, there are no trailing blanks. This procedure fails because the trailing blanks in the pattern are significant
- However, the following example succeeds because trailing blanks are not added to a **varchar** variable.

Example

```
USE AdventureWorks;
GO
CREATE PROCEDURE FindEmployee @EmpLName varchar(20)
AS
SELECT @EmpLName = RTRIM(@EmpLName) + '%';
SELECT c.FirstName, c.LastName, a.City
FROM Person.Contact c JOIN Person.Address a ON c.ContactID =
    a.AddressID
WHERE c.LastName LIKE @EmpLName;
GO
EXEC FindEmployee @EmpLName = 'Barb';
```

Output

```
FirstName         LastName                    City
---------- ----------------------------------------
Angela             Barbariol                  Snohomish
David              Barber                     Snohomish

(2 row(s) affected)
```

NOT LIKE

- If preceeded by NOT, LIKE returns true if the match expression does not match the pattern

ESCAPE Characters

- You can search for character strings that include one or more of the special wildcard characters.
- For example, a sample database contains a column named **comment** that contains the text 30%. To search for any rows that contain the string 30% anywhere in the **comment** column, specify a WHERE clause such as WHERE comment LIKE '%30!%%' ESCAPE '!'. If ESCAPE and the escape character are not specified, the Database Engine returns any rows with the string 30!.

ESCAPE Characters

- The character after the escape character is interpreted literally, not as a wildcard character

Full Text Search

- Full-text search allows fast and flexible indexing for keyword-based query of text data stored in a SQL Server database.
- In contrast to the LIKE predicate, which only works on character patterns, full-text queries perform linguistic searches against this data, by operating on words and phrases based on the rules of a particular language.

Full Text Search

- The performance benefit of using full-text search can be best realized when querying against a large amount of unstructured text data.
- A LIKE query against millions of rows of text data can take minutes to return; whereas a full-text query can take only seconds or less against the same data, depending on the number of rows that are returned
- A full text search uses full text indexes

Full Text Indexes

- You can build full-text indexes on columns that contain **char**, **varchar** and **nvarchar** data.
- Full-text indexes can also be built on columns that contain formatted binary data, such as Microsoft Word documents, stored in a **varbinary(max)** or **image** column. You cannot use the LIKE predicate to query formatted binary data.

Full Text Indexes

- To create a full-text index on a table, the table must have a single, unique not null column.
- For example, consider a full-text index for the **Document** table in AdventureWorks in which the **DocumentID** column is the primary key column.
- A full-text index indicates that the word "instructions" is found at word number 24 and word number 44 in the **DocumentSummary** column for the row associated with a **DocumentID** of 3.

Full-text catalog

- A full-text catalog contains zero or more full-text indexes.
- Full-text catalogs must reside on a local hard drive associated with the instance of SQL Server.
- Each catalog can serve the indexing needs of one or more tables within a database.

Full-Text Engine

- Full-Text Search in Microsoft SQL Server 2005 is powered by the Microsoft Full-Text Engine for SQL Server (MSFTESQL). The MSFTESQL service has two roles, namely indexing support and querying support.
- It is a separate process

Word Breakers and Stemmers

- Word breakers and stemmers perform linguistic analysis on all full-text indexed data.
- Linguistic analysis involves finding word boundaries (word-breaking) and conjugating verbs and nouns (stemming).
- The rules for this analysis differ for different languages, and you can specify a different language for each full-text indexed column.
- Word breakers for each language enable the resulting terms to be more accurate for that language. If no word breaker is available for a particular language, the neutral word breaker is used. With the neutral word breaker, words are broken at neutral characters such as spaces and punctuation marks.

Stemmers

- For a given language, a stemmer generates inflectional forms of a particular word based on the rules of that language. Stemmers are language specific

Filters

- When a cell in a **varbinary(max)**, or **image** column contains a document with a certain file extension, full-text search uses a filter to interpret the binary data.
- The filter extracts the textual information from the document and submits it for indexing.

Filters

- Many document types can be stored in a single **varbinary(max)**, or **image** column. For each document type, SQL Server chooses the correct filter based on the file extension.
- Because the file extension is not visible when the file is stored in a **varbinary(max)**, or **image** column, the file extension must be stored in a separate column in the table, called a type column.
- This type column can be of any character-based data type and contains the document file extension, such as .doc for a Microsoft Word document.

Filters

- In the **Document** table in Adventure Works, the **Document** column is of type **varbinary(max)**, and the **FileExtension** column is of type **nvarchar(8)**.
- When creating a full-text index on a **varbinary(max)**, or **image** column you must identify a corresponding type column that has the extension information so that SQL Server knows which filter to use.

Building a Full Text Index

- Token: a word or a character string identified by the word breaker
- The process of building a full-text index is quite different from building other types of indexes.
- Instead of constructing a B-tree structure based on a value stored in a particular row, MSFTESQL builds an inverted index structure based on individual tokens from the text being indexed.

Full-Text Index Structure

- The following excerpt of the **Document** table in Adventure Works shows three rows from the table and two columns, the **DocumentID** column and the **Title** column.
- For this example, we will assume that a full-text index has been created on the **Title** column

DocumentID	**Title**
1	Crank Arm and Tire Maintenance
2	Front Reflector Bracket and Reflector Assembly 3
3	Front Reflector Bracket Installation

Full Text Index

Keyword	ColId	DocId	Occ
Crank	1	1	1
Arm	1	1	2
Tire	1	1	4
Maintenance	1	1	5
Front	1	2	1
Front	1	3	1
Reflector	1	2	2
Reflector	1	2	5
Reflector	1	3	2
Bracket	1	2	3
Bracket	1	3	3
Assembly	1	2	6
3	1	2	7
Installation	1	3	4

Full Text Index

- The **Keyword** column contains a representation of a single token extracted at indexing time. Word breakers determine what makes up a token
- The **ColId** column contains a value that corresponds to the particular table and column that is full-text indexed.
- The **DocId** column contains the identifier of the document that contains the Keyword

Full Text Index

- The **Occ** column contains an integer value that correspond to the relative word offsets of the particular keyword within that **DocId**.

Index Population

- The process of creating and maintaining a full-text index is called index population. Types of full-text index population:
  - Full population
  - Change tracking-based population
  - Incremental timestamp-based population

Full Population

- Typically occurs when a full-text catalog or full-text index is first populated. The indexes can then be maintained using change tracking or incremental timestamped-based populations.
- During a full population of a full-text catalog, index entries are built for all the rows in all the tables covered by the catalog.
- If a full population is requested for a table, index entries are built for all the rows in that table.

Change Tracking Based Population

- SQL Server maintains a record of the rows that have been modified in a table that has been set up for full-text indexing. These changes are propagated to the full-text index.

Incremental Timestamp Based Population

- Incremental population updates the full-text index for rows added, deleted, or modified after the last population.
- The requirement for incremental population is that the indexed table must have a column of the timestamp data type. If a timestamp column does not exist, incremental population cannot be performed.
- A request for incremental population on a table without a timestamp column results in a full population operation.

Index Creation

- Two steps:
  - Create a full-text catalog to store full-text indexes.
  - Create full-text indexes.

Example

- To create a full-text catalog named AdvWksDocFTCat, use the CREATE FULLTEXT CATALOG statement as shown below.

```
CREATE FULLTEXT CATALOG AdvWksDocFTCat
```

Example

```
CREATE FULLTEXT INDEX ON Production.Document
(
   Document                          --full-text index column name
      TYPE COLUMN FileExtension
          --name of column that contains file type information
      Language 0X0                  --0X0 is LCID for neutral language
)
KEY INDEX PK_Document_DocumentID --Unique index
          ON AdvWksDocFTCat
WITH CHANGE_TRACKING AUTO          --Population type
GO
```

Example

- CHANGE_TRACKING AUTO
  - Specifies that SQL Server automatically updates the full-text index as the data is modified in the associated tables. AUTO is the default.

Full-Text Searching

- To seach full-text use the CONTAINS predicate in the WHERE clause of a query
- Searching for Specific Word or Phrase (Simple Term)
- Searching for the Inflectional Form of a Specific Word (Generation Term)
- Performing Prefix Searches
- Querying varbinary(max) and xml Columns
- Searching for Words or Phrases Close to Another Word or Phrase (Proximity Term)

CONTAINS Syntax

```
CONTAINS      ( { column_name | (column_list) | * } ,
    '< contains_search_condition >'
    [ , LANGUAGE language_term ]       )
< contains_search_condition > ::=
{ < simple_term >     |
   < prefix_term >     |
   < generation_term >     |
   < proximity_term >  }
| ( < contains_search_condition >  )
 { < AND > | < AND NOT > | < OR > }
   <contains_search_condition >   } [ ...n ]
```

CONTAINS Syntax

- { *column_name* | **(***column_list***)** | ***** } indicate in which column to search
- < simple_term > ::= *word* | **"** *phrase* **"**
  - Specifies a match for an exact word or a phrase. Examples of valid simple terms are "blue berry", blueberry, and "Microsoft SQL Server".
  - Phrases should be enclosed in double quotation marks ("").
  - Words in a phrase must appear in the database in the same order as specified in <contains_search_condition>.

Simple Term

- The search for characters in the word or phrase is not case sensitive.
- Noise words (such as a, and, or the) in full-text indexed columns are not stored in the full-text index. If a noise word is used in a single word search, SQL Server returns an error message indicating that the query contains only noise words.
- Punctuation is ignored. Therefore, CONTAINS(testing, '"computer failure"') matches a row with the value, "Where is my computer? Failure to find it would be expensive."

Example of Searching for a Simple Term

- If you want to search the **ProductReview** table in the **AdventureWorks** database to find all comments about a product with the phrase "learning curve", you could use the CONTAINS predicate as follows.

```
SELECT Comments
FROM Production.ProductReview
WHERE CONTAINS(Comments, ' "learning curve" ');
GO
```

Example

```
SELECT Name, ListPrice
FROM Production.Product
WHERE ListPrice = 80.99
  AND CONTAINS(Name, 'Mountain');
GO
```

Example

```
SELECT Name
FROM Production.Product
WHERE CONTAINS(Name, ' "Mountain" OR "Road" ')
GO
```

Prefix Term

< prefix term > ::= { **"***word* *** "** | **"***phrase* ***"** }

- All entries in the column that contain text beginning with the specified prefix will be returned.
- For example, to search for all rows that contain the prefix **top-**, as in **top**ple, **top**ping, and **top** itself, the query looks like this

```
SELECT Description, ProductDescriptionID
FROM Production.ProductDescription
WHERE CONTAINS (Description, ' "top*" ' );
GO
```

Performing Prefix Searches

- When the prefix term is a phrase, each token making up the phrase is considered a separate prefix term. All rows that have words beginning with the prefix terms will be returned. For example, the prefix term "light bread*" will find rows with text of either "light breaded," "lightly breaded," or "light bread", but will not return "Lightly toasted bread".

Generation Term

- < generation_term > ::= FORMSOF **(** { INFLECTIONAL | THESAURUS } **,** < simple_term > [**,**...*n*] **)**
  - Specifies a match of words when the included simple terms include variants of the original word for which to search.

Generation Term

- INFLECTIONAL
  - Specifies that the language-dependent stemmer is to be used on the specified simple term. The column language of the column(s) being queried is used to refer to the desired stemmer. If *language_term* is specified, the stemmer corresponding to that language is used.
- THESAURUS
  - Specifies that the thesaurus corresponding to the column full-text language, or the language specified in the query is used.
  - All synonyms associated with the search term are identified, and searched in the column

Example of Searching for a Generation Term

- You can search for all the different tenses of a verb or both the singular and plural forms of a noun.
- For example, the query shown in this topic searches for any form of "foot" ("foot", "feet", and so on) in the **Comments** column of the **ProductReview** table.

```
SELECT Comments, ReviewerName
FROM Production.ProductReview
WHERE CONTAINS (Comments,
   'FORMSOF(INFLECTIONAL, "foot")');
GO
```

Proximity Term

- < proximity_term > ::=
  { < simple_term > | < prefix_term > }
  { { NEAR | ~ } { < simple_term > | < prefix_term >}
  } [...*n*]
  - Indicates that the word or phrase on the left side of the NEAR or ~ operator should be approximately close to the word or phrase on the right side of the NEAR or ~ operator.
  - Multiple proximity terms can be chained

Example of Searching for Words or Phrases Close to Another Word or Phrase

- The following example returns all product names that have the word "bike" near the word "performance".

```
SELECT Description
FROM Production.ProductDescription
WHERE CONTAINS(Description, 'bike NEAR
   performance');
GO
```

Querying varbinary(max) and xml Columns

- When a **varbinary(max)** or an **xml** column participates in a full-text index, the full-text service looks at the extensions of the documents contained in the **varbinary(max)** column and applies a corresponding filter to interpret the binary data and extract the textual information needed for full-text indexing and querying. For an **xml** column, the xml filter is applied.
- Once indexed, the **varbinary(max)** or **xml** column can be queried like any other column in a table, using the predicates CONTAINS