Microsoft SQL Server 9.0 Technical Articles

SQL Server 2005 Full-Text Search: Internals and Enhancements

Andrew Cencini
Microsoft Corporation

December 2003
revised May 2006

Applies to:
Microsoft® SQL Server 2005
Transact-SQL (T-SQL) Language

Summary: See the benefits and new features of SQL Server 2005 full-text search for both developers and database administrators. (23 printed pages)

Contents

Introduction

The full-text search feature of Microsoft SQL Server 2005 delivers enterprise search functionality integrated into the database. Significant enhancements in the areas of performance, manageability, and functionality deliver best-of-breed search capabilities for applications of any size.

This paper is for database administrators and developers alike. An introduction to full-text search and its architecture is followed by details about important enhancements and new features.

What Can Full-Text Search Do for Me?

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

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 (for example, '%cencini%') against millions of rows of text data can take minutes to return; whereas a full-text query (for 'cencini') can take only seconds or less against the same data, depending on the number of rows that are returned.

Full-text indexes may be built not just on columns that contain text data, but also against formatted binary data, such as Microsoft® Word documents, stored in a BLOB-type column; in these cases, it is not possible to use the LIKE predicate for keyword queries.

With the growing popularity of storing and managing textual data in a database, demand has risen for full-text search capabilities in a wide variety of applications. Common uses of full-text search include Web-based applications (searching websites, product catalogs, news items, and other data), document management systems, and custom applications that need to provide text search capabilities over data stored in a SQL Server database.

SQL Server 2005 full-text search can scale from small mobile or personal deployments with relatively few

and simple queries, up to complex mission-critical applications with high query volume over huge amounts of textual data. Full-text search provides integrated management capabilities and easy-to-use Transact-SQL query syntax; building applications that expose search capabilities is quick and easy.

Full-text search is also highly extensible. The full-text engine can support additional languages for index and query (by adding additional word breakers or stemmers from a third-party vendor), as well as filtering of additional document formats (there are a number of third-party filters to choose from). A set of well-known, published interfaces provide the framework for full-text engine extensibility For more information, see the MSDN topics IFilter [http://technet.microsoft.com/en-us/library/ms691105(printer).aspx] , IWordBreaker [http://technet.microsoft.com/en-us/library/ms691079(printer).aspx] , and Istemmer [http://technet.microsoft.com/en-us/library/ms690983(printer).aspx] .

In short, if there is an application that manages textual data stored in SQL Server, chances are that full-text search will add great value by providing fast, robust, enterprise-class search functionality integrated into the database platform.

Full-Text Search Architecture

The Microsoft Full-Text Engine for SQL Server (MSFTESQL) powers full-text search in SQL Server. This engine is in turn built on Microsoft Search (MSSearch) technology and is integrated into the SQL Server 2005 Database Engine more tightly than ever. One major architectural improvement made in the SQL Server 2005 release has been the implementation of side-by-side installs of the full-text engine. This means that for each instance of SQL Server 2005, there is a dedicated instance of MSFTESQL, including dedicated components (such as word breakers and filters), resources (such as memory), and configuration (such as service-level settings like resource_usage) at the instance level.

(The full-text search engine in previous versions of SQL Server was shared by all instances of SQL Server, as well as the operating system and other Microsoft server products such as Exchange and SharePoint™ Portal Server. A side effect of this approach was that updates to the operating system or other products that used MSSearch had wide and sometimes undesired effects on all other products that used MSSearch.)

Let's take a closer look at SQL Server 2005 full-text search.

Indexing

Full-text indexes are a special type of token-based functional index that is built and maintained by the full-text engine. The process of building a full-text index is quite different from building other types of indexes. Rather than construct a B-tree structure based on a value stored in a particular row, MSFTESQL builds an inverted, stacked, compressed index structure based on individual tokens from the text being indexed.

Index Structure

A good understanding of the structure of a full-text index helps one appreciate how the full-text engine works. Let's take a look at a basic example of a full-text index rendered in tabular form:

> **Note** Actual full-text indexes contain information in addition to what is presented in this table. The table below is provided for explanatory purposes.

Keyword	ColId	DocId	Occ
token1	1	1	1
token1	1	1	5
token1	1	2	2
token1	2	1	1
token2	1	1	5
token2	2	4	11
(etc.) ...	(etc.) ...	(etc.) ...	(etc.) ...

The **Keyword** column corresponds to some representation of a single token extracted at indexing time. What makes up a token is determined by a component of the full-text engine called a "word breaker." Each language supported by full-text search has a word breaker component that breaks a text stream into individual tokens based on the rules for word boundaries of that particular language.

The **ColId** column is a numeric value that corresponds to a particular table and column that is full-text indexed. Since multiple tables and columns may be stored in a single full-text catalog, the **ColId** value is used to narrow queries down to only occurrences of a keyword that come from the table and column specified in a full-text query.

Each **DocId** value for a **Keyword**/**ColId** pair maps to a particular full-text key value in a full-text indexed table. **DocId** values that satisfy a search condition are passed from the full-text engine to the database engine, where they are mapped to full-text key values from the base table being queried.

> **Note** **DocId** is represented internally as a compressible 4-byte integer value, which therefore may be different from the data type of the full-text key. As such, the upper limit to the number of full-text indexed rows in a full-text catalog in SQL Server 2005 is roughly 2,000,000,000.

The **Occ** column is actually a list of lists—for each **DocId** value, there is a list of occurrence values that correspond to relative offsets of the particular keyword within that **DocId**. Occurrence values are useful in determining phrase or proximity matches (for example, phrases have numerically adjacent occurrence values), as well as in computing relevance score (for example, the number of occurrences in a **DocId** as well as in a particular full-text index may be used in scoring).

As indicated before, the **DocId** and **Occ** values are compressed, primarily to save space and I/O costs at query time. During the indexing process, index fragments of a certain size are generated and periodically merged together into a larger master index; otherwise, the cost of inserting into this specially keyed compressed structure, potentially thousands of times per row, would be very expensive.

Now that you have a basic idea of the full-text index structure, let's look at how SQL Server 2005 full-text search can take massive amounts of text data stored in the database and build a full-text index on it.

Indexing Process

The indexing process consists of two conceptual pieces: gathering data and constructing the full-text index structure. The architecture of the full-text gathering mechanism was improved in SQL Server 2005 in order to make the full-text indexing process more efficient, leading to significant performance improvements.

When a full-text population (also known as a "crawl") is initiated, the Database Engine pushes large batches of data into memory and tells the full-text engine to begin indexing. (The values from a column or columns of a table are full-text indexed.) Using a protocol handler component, the full-text engine pulls the data from memory through its indexing pipeline to be further processed, resulting in a full-text index.

In past releases, the gathering process was akin to a Web crawl, based on a row-by-row pulling mechanism as opposed to the batching semantics described above. SQL Server Yukon's batching mechanism, by contrast, makes more aggressive and efficient use of resources, and in turn yielding a massive full-text indexing performance improvement.

> **Note** When indexing data stored in a BLOB column, a special component that implements the **IFilter** interface is used to extract text based on the specified file format for that data (e.g., Microsoft Word). In some cases, the filter components require the BLOB data to be written out to disk (as opposed to pushed into memory), so the indexing process can be slightly different for that type of data.

As part of processing, the gathered text data is passed through a word breaker to separate the text into individual tokens (keywords). The language to be used for tokenization is specified on a per-column level, or may be identified within BLOB or XML data by the filter component.

Additional processing may be performed to remove "noise" words (that is, words with little value as search criteria), and to normalize tokens before they are stored in the full-text index or an index fragment.

Figure 1 shows an overview of the components of the full-text engine in SQL Server 2005.

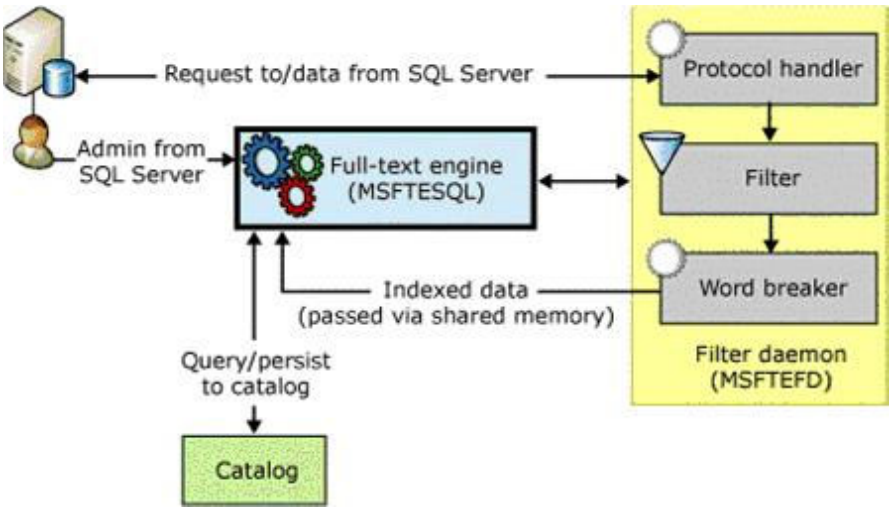

Figure 1. Full-text engine components in SQL Server 2005

When a population has completed, a final 'master' merge process is triggered. By merging the index fragments together, query performance is improved (because only the master index needs to be queried rather than a number of index fragments), and better scoring statistics may be used for relevance ranking. Because large amounts of data must be written and read when index fragments are merged, the master merge can be I/O intensive, but it will not block incoming queries.

Query

Querying a full-text index is extremely fast and flexible. Because of the specialized index structure described above, it is very quick and easy to locate matches for particular search criteria.

Below is an example of a full-text query in SQL Server 2005:

Copy Code

```
SELECT ProductModelId, ProductName
FROM ProductModel
WHERE CONTAINS(CatalogDescription, ' " aluminum alloy " ')
```

This query will project out **ProductModelId** and **ProductName** from the **ProductModel** table for each row that contains the phrase "aluminum alloy" anywhere in its **CatalogDescription** field.

The full-text portion of the query is parsed and sent to the full-text engine to be evaluated against the full-text index for the **ProductModel** table. The query terms in the search condition (in this case, "aluminum alloy") are tokenized by the word breaker for the column- or query-specified language, any noise words are removed, and a list of matching **DocId** values is returned from the full-text engine. Within the Database Engine, the list of **DocId** values is looked up against an internal structure that maps **DocId** values to full-text key values, and the resulting full-text key values are joined to the **ProductModel** table in order to project out the **ProductModelId** and **ProductName** values for the matching rows.

> **Note** In past releases, the mapping of DocIds to full-text key values was performed in the full-text engine. In SQL Server 2005, this process has migrated into the database engine where more efficient and consistent caching strategies may be utilized. This migration (plus deeper enhancements made to the query engine) should also speed up full-text queries over previous releases. The full-text query performance improvements range from modest to orders of magnitude better for some queries.

The above query example represents a simple full-text query. Full-text queries have a robust yet simple syntax that allows for the evaluation of extremely powerful query expressions. This paper will not go into much more detail on the specific elements of full-text search expressions; for more information, see SQL

Server 2005 Books Online.

Ranking

Full-text search in SQL Server can generate a score (or rank value) about the relevance of the data being returned; this per-row rank value can be an ordering semantic, so that data that is more relevant is presented ahead of data that is considered less relevant.

Rank values for full-text queries may be returned from CONTAINSTABLE and FREETEXTTABLE full-text query clauses. CONTAINSTABLE and FREETEXTTABLE behave like table-valued functions that return two columns: a key column (which can be used to join to the full-text indexed base table), and a rank column (which can be used to order results by rank). The CONTAINS and FREETEXT clauses do not expose a score value, because they are predicates in the Transact-SQL language.

Three types of ranking are used in full-text queries including CONTAINSTABLE, CONTAINSTABLE using ISABOUT (weighting), and FREETEXTTABLE. All are based on the distribution of words in the query and the indexed data, but each works differently. Further, none of the ranking methods is absolute; for performance reasons, many values are rounded and normalized. Ranks of query results are only useful in relation to the ranks of other results from the same query. They are not comparable to the ranks of results from other queries, either from the same catalog or from other catalogs.

The science of ranking is far from mature, and as the field evolves, Microsoft may change how ranking works. Therefore, applications built on MSSearch must not rely on any particular ranking implementation, or they may break when a new version of MSSearch is released. In fact, ranking in MSSearch of 2005 differs in several ways from ranking in products such as Sharepoint Portal Server, Indexing Service, SQL Server 2000, and others.

Ranking over Boolean clauses is handled by taking the minimum rank from nodes under an AND clause and the maximum rank from nodes under an OR clause. Because ranks aren't directly comparable, this is at best an incorrect approximation of the ideal. For this reason, queries like

```
FREETEXTTABLE(col1,  'foo')
```

and

```
CONTAINSTABLE(col1, 'bar')
```

are certainly not recommended. Given the combination heuristics above, there would be no telling for each result which clause the rank came from—the FREETEXTTABLE or the CONTAINSTABLE clause.

Statistics for Ranking

When an index is built, statistics are collected for use in ranking. To minimize the size of the index and computational complexity, the statistics are often rounded.

When a catalog is being built, the algorithm creates small indexes as data is indexed, then merges the indexes into a large index. This process is repeated many times. A final merging of indexes into one large master index is called a "master merge". Some statistics are taken from individual indexes that contain query results, and some from the master index; others are computed only when a master merge takes place. As a result, the ranking statistics can vary greatly in accuracy and timeliness. This also explains why the same query can return different rank results over time as indexes are merged. Further, as full-text indexed data is added, modified, and deleted, those changes also will impact statistics and rank computation.

The list below includes some commonly used statistics that are important in calculating rank:

- **Property:** An attribute of a document. This corresponds to a column in SQL Server.
- **Document:** The entity that is returned in queries. In SQL Server this corresponds to a row. A document can have multiple properties, just as a row can have multiple full-text indexed columns.
- **Index:** A single inverted index of one or more documents. This may be entirely in memory or on disk. Many query statistics are relative to the individual index where the match occurred.
- **Catalog:** A collection of indexes treated as one entity for queries. Catalogs are the unit of organization visible to the SQL Server administrator.
- **Word:** The unit of matching in the full-text engine. Streams of text from documents are tokenized into words by language-specific word breakers.

- **Occurrence:** The word offset in a document property as determined by the word breaker. The first word is at occurrence 1, the next at 2, and so on. In order to avoid false positives in phrase and proximity queries, end-of-sentence skips 8 occurrences. End-of-paragraph skips 128 occurrences.
- **Key:** Combination of a property and a word.
- **HitCount:** The number of times the key occurs in the result.
- **Log2:** The highest-order bit set in a 4-byte value. Log base 2 was chosen over any other type of logarithmic computation for performance reasons. It is much faster than computing log base 10 or log base *e*.

Copy Code

```
unsigned Log2 ( unsigned long s )
{
    for ( unsigned iLog2 = 0; s != 0; iLog2++ )
        s >>= 1;
    return iLog2;
}
```

- **IndexDocumentCount:** Total number of documents in the index.
- **KeyDocumentCount:** Number of documents in the index containing the key.
- **MaxOccurrence:** The largest occurrence stored in an index for a given property in a document.
- **MaxQueryRank:** The maximum rank returned by the engine (1000).

Ranking of CONTAINSTABLE

Copy Code

```
StatisticalWeight = Log2( ( 2 + IndexDocumentCount ) / KeyDocumentCount )
Rank = min( MaxQueryRank, HitCount * 16 * StatisticalWeight / MaxOccurrence )
```

Phrase matches are ranked just like individual keys except that KeyDocumentCount (the number of documents containing the phrase) is assumed to be 1. This can be wrong in many cases, and leads to phrases having relatively higher weights than individual keys.

Ranking of ISABOUT

ISABOUT is a vector-space query in traditional information retrieval terminology. The default ranking algorithm used is Jaccard, a widely known formula. The ranking is computed for each term in the query and then combined as described below.

Copy Code

```
ContainsRank = same formula used for CONTAINSTABLE ranking of a single term (above).
Weight = the weight specified in the query for each term.  MaxQueryRank is the
default weight.
WeightedSum = Σ[key=1 to n] ContainsRankKey * WeightKey
Rank =  ( MaxQueryRank * WeightedSum ) / ( ( Σ[key=1 to n] ContainsRankKey2 )
      + ( Σ[key=1 to n] WeightKey2 ) - ( WeightedSum ) )
```

The sums are computed using unsigned 4-byte integers. For this reason, no more than 4294 keys can be in any given vector query because the integer may overflow (this condition is checked and such queries are failed). The other math is done in 8-byte integers.

Ranking of FREETEXT

Freetext ranking is based on the OKAPI BM25 ranking formula. Each term in the query is ranked, and the

values are summed. Freetext queries will add words to the query via inflectional generation (stemmed forms of the original query terms); these words are treated as separate terms with no special weighting or relationship with the words from which they were generated. Synonyms generated from the Thesaurus feature are treated as separate, equally weighted terms.

Copy Code

```
Rank = Σ[Terms in Query] w ( ( ( k1 + 1 ) tf ) / ( K + tf ) ) * ( ( k3 + 1 ) qtf / ( k3 + qtf ) ) )
```

Where:

- `w`
  is the Robertson-Sparck Jones weight.
- Originally,
  `w`
  is defined as:

Copy Code

```
w = log10 ( ( ( r + 0.5 ) * ( N - n - R + r + 0.5 ) ) / ( ( R - r + 0.5 ) * ( n - r + 0.5 ) ) )
```

This was simplified to:

Copy Code

```
w = log10 ( ( ( r + 0.5 ) * ( N - R + r + 0.5 ) ) / ( ( R - r + 0.5 ) * ( n - r + 0.5 ) ) )
```

- `R`
  is the number of documents marked relevant by a user. This is not implemented in SQL Server 2005 full-text search, and thus is ignored.
- `r`
  is the number of documents marked relevant by a user containing the term. This is not implemented.
- `N`
  is the number of documents with values for the property in the query.
- `n`
  is the number of documents containing the term.
- `K`
  is
  `( k1 * ( ( 1 - b ) + ( b * dl / avdl ) ) )`
  .
- `dl`
  is the document length, in word occurrences.
- `avdl`
  is the average document length of the property over which the query spans, in word occurrences.

- `k1`, `b`, and `k3` are the constants 1.2, 0.75, and 8.0, respectively.
- `tf` is the frequency of the term in a specific document.
- `qtf` is the frequency of the term in the query.

Rank Computation Issues

The process of computing rank, as described above, depends on a number of factors:

- Different word breakers tokenize text differently. For example, the string "dog-house" will be broken into "dog" "house" by one word breaker, and into "dog-house" by another. This means that matching and ranking will vary based on the language specified, because not only are the words different, but also the document length is different. The document length difference can affect ranking for queries not involving any of the words in the string.
- Statistics such as **IndexDocumentCount** can vary widely. For example, if a catalog has 2 billion rows in the master index, then one new document is indexed into an in-memory index, and ranks for that document based on the number of documents in the in-memory index will be skewed compared with ranks for documents from the master index. For this reason, it is recommended that the indexes be merged into a master index (using the ALTER FULLTEXT CATALOG ... REORGANIZE DDL statement) once all indexing is complete.
- Word breakers and filters together detect sentences and paragraphs. Occurrences skip by 8 at end of sentence and by 128 at end of paragraph. So **MaxOccurrence** can vary widely depending on how many sentences and paragraphs are detected in a property value.
- **MaxOccurrence** values are normalized into 1 of 32 ranges. This means, for example, that a document 50 words long is treated the same as a document 100 words long. Below is the table used for normalization. Because the document lengths are in the range between adjacent table values 32 and 128, they are effectively treated as having the same length, 128 (32 < docLength <= 128).

  Copy Code

```
{ 16, 32, 128, 256, 512, 725, 1024, 1450, 2048, 2896, 4096, 5792, 8192, 11585,
16384, 23170, 28000, 32768, 39554, 46340, 55938, 65536, 92681, 131072, 185363,
262144, 370727, 524288, 741455, 1048576, 2097152, 4194304 };
```

- In cases where TOP_N_BY_RANK is used with the new precomputed rank option (discussed later), some additional rounding may be done while computing rank, so there may be some difference in those values based on whether or not the precomputed rank option is enabled.

Now that you have a good understanding of full-text search, let's move on to the new features in SQL Server 2005.

New Features for the Database Administrator

Side-by-Side Full-Text Engine Install

The Full-Text Search Architecture section mentioned side-by-side full-text engine install, but did not spell out its meaning for database administrators.

First of all, side-by-side install of the full-text engine simplifies the process of managing and updating your servers. In the past, updating operating system service packs, or other instances or server products on the system, could affect SQL Server full-text search through the shared Microsoft Search (MSSearch) service. Furthermore, installing products like SharePoint Portal Server could introduce a newer version of MSSearch, which could change the behavior of full-text search. In contrast, isolating the full-text engine to the instance level prevents such unwanted side effects.

Side-by-side install means that the following components now exist at the per-instance level in SQL Server 2005:

- **MSFTESQL**. The full-text engine service manages the filter daemon component, performs administrative operations, and executes full-text queries. It appears as MSFTESQL$<*instance_name*> for a named instance.
- **MSFTEFD**. The full-text filter daemon safely loads and drives third-party extensible components used for index and query, such as word breakers, stemmers, and filters, without compromising the integrity of the full-text engine service itself.
- **Word breakers, stemmers, and filters**. Each instance now uses its own set of word breakers, stemmers, and filters, rather than relying on the operating system version of these components. These components are also easier to register and configure at a per-instance level. For more information, see SQL Server 2005 Books Online.

Security Enhancements

Improved security is a benefit of the side-by-side install capabilities of full-text search in SQL Server 2005. Past releases required the shared MSSearch service to run as LocalSystem; however, many organizations' security policies require services to run in a low-privileged security context. In SQL Server 2005, the full-text engine is synchronized to use the same account as the Database Engine (as opposed to LocalSystem), making it is easier to deploy full-text search functionality in a locked-down environment, yet the side-by-side approach ensures there is no risk of the side effects that can be introduced in a shared-service setting.

SQL Server 2005 full-text search can also prevent unsigned binaries from loading into the full-text engine process. Full-text search exposes an extensible architecture for filters, stemmers, and word breakers, so you can install third-party components as well. To prevent the execution of dangerous or untrusted code, full-text search exposes a configurable setting that by default prohibits loading components that are not Authenticode-signed by a trusted source.

Indexing Performance and Scale Enhancements

Perhaps the most dramatic improvement in SQL Server 2005 full-text search over previous releases has been made in the area of full-text indexing performance. Re-architecture of the full-text gathering mechanism (discussed briefly in "Indexing" above), as well as improvements in index merge strategy, have improved indexing performance by more than an order of magnitude in internal tests. For example, on the same hardware, with the same data set, building a full-text index on 20 million rows of character-based text data took roughly 14 days in SQL Server 2000, while in SQL Server 2005, the same index required less than 10 hours.

An additional benefit of the indexing re-architecture in SQL Server 2005 is improved resource utilization. Building a full-text index on a server with a large amount of available memory achieves optimal performance, as that memory will be used for pushing larger batches of data to be indexed.

Full-text indexes in SQL Server 2005 also can scale up to large quantities of data. Previous releases scaled to tens- or low-hundreds of thousands of rows of data; SQL Server 2005 full-text catalogs, by contrast, have been tested with and can support up to 2,000,000,000 rows of data (based on the 4-byte internal DocId). Further, the indexing process also scales up to that amount of data on a larger number of CPUs. Scalability of the text engine at indexing time over multiple CPUs has also improved significantly over previous releases (the full-text engine scales well up to roughly 16 CPUs on a 32-bit platform).

Full-Text Data Definition Language

Not only is it faster than ever to create full-text indexes in SQL Server 2005, the process has also become

easier. SQL Server 2005 introduces Data Definition Language (DDL) statements for creating, altering, and dropping full-text indexes and catalogs. (Formerly, these indexes and catalogs were created using system stored procedures, which are deprecated but continue to work.)

An example of basic full-text DDL statements is provided below:

Copy Code

```
USE AdventureWorks
GO
-- creates full-text catalog - note accent sensitivity may now be
-- specified note also that this catalog is created as default so any
-- index I create that does not specify catalog will use this catalog
CREATE FULLTEXT CATALOG AwCat WITH ACCENT_SENSITIVITY=OFF AS DEFAULT
GO

-- creates index in AwCat (default) on ProductModel table
-- note we are using ProductModelId (unique single-column non-nullable)
-- as our FT key we also set change tracking to be manual so we can
-- periodically sync changes
CREATE FULLTEXT INDEX ON ProductModel(CatalogDescription)
KEY INDEX PK_ProductModel_ProductModelId WITH CHANGE_TRACKING MANUAL
GO

-- start a manual change tracking update on ProductModel
-- see Books Online for more information on full-text population options
ALTER FULLTEXT INDEX ON ProductModel START MANUAL POPULATION
GO

-- trigger a master merge on AwCat - can help with query performance
-- and improves ranking. This is done automatically after full and
-- manual populations, but can help periodically in auto change
-- tracking with a good amount of updates
ALTER FULLTEXT CATALOG AwCat REORGANIZE
GO
```

Using full-text DDL in SQL Server 2005 provides access to an easier-to-use syntax, as well as the many new features of full-text search.

Backup, Restore, and Recovery

One of the most significant full-text integration features of the SQL Server 2005 release has been the capability to back up, restore, and recover full-text catalogs using the same facilities as regular database and transaction log files. Full-text catalogs are managed by the full-text engine, and are stored in the file system. In previous releases, full-text catalogs needed to be backed up separately from SQL Server data files, which could cause difficulty in ensuring a smooth disaster recovery.

In contrast, SQL Server 2005 full-text catalogs are included in database, log, file, and filegroup backups. When a backup is taken, any indexing activity is temporarily suppressed to ensure consistent state between the full-text catalog and database, at which point the full-text catalog is backed up. For regular log backups, the full-text change-tracking log is included as part of the backup, and only data that has changed between the last full or differential backup needs to be re-indexed. (To speed recovery, take a full or differential backup after making schema changes or significant data loading to a full-text catalog.)

Using the **AdventureWorks** database example above, which includes the AwCat full-text catalog, you could execute the following command to back up the database, including its full-text catalog:

Copy Code

```
BACKUP DATABASE AdventureWorks TO DISK='C:\backups\aw.dmp'
GO
```

After backing up the database, you could simulate a disaster by executing:

Copy Code

```
USE master
GO
DROP DATABASE AdventureWorks
GO
```

Likewise, to restore the database, including its full-text catalog, you can execute the command:

Copy Code

```
RESTORE DATABASE AdventureWorks FROM DISK='C:\backups\aw.dmp'
GO
```

Therefore, the following query should work just as it did before the disaster:

Copy Code

```
USE AdventureWorks
SELECT ProductName
FROM ProductModel
WHERE CONTAINS(CatalogDescription, ' " aluminum alloy " ')
```

Transportability via sp_detach_db, sp_attach_db

SQL Server 2005 full-text search provides the ability to easily detach and move full-text catalogs in the same way that SQL Server database files may be detached, moved, and re-attached. Full-text catalogs are included with **sp_detach_db** and **sp_attach_db**. After detaching a database, you may move the full-text catalog and/or database data files, and then re-attach the database. Full-text catalog metadata is updated to reflect the change of location. This capability simplifies building, testing, moving, and deploying databases across multiple servers.

Rich Status Reporting

To provide the best possible manageability and supportability experience for full-text users, SQL Server 2005 full-text search provides easy ways for database administrators to diagnose and repair errors.

The crawl logging mechanism of SQL Server 2005 full-text maintains verbose, plain-text logs under the \LOG directory for each full-text catalog; these logs provide informational messages about full-text populations, as well as useful error messages and, if necessary, suggested resolutions to those problems.

Further, improved granularity of status reporting for processes such as full-text populations allows for easier tracking of various processes that are under way on the server. Additionally, more granular properties for status reporting were added to a number of built-in functions like OBJECTPROPERTY, and all full-text metadata is exposed through the new catalog views (such as **sys.fulltext_catalogs**, **sys.fulltext_indexes**, and **sys.fulltext_index_columns**).

Profiler Support for Full-Text Queries

Database administration and profiling are now easier with the addition of the full-text query event to the SQL Server 2005 Profiler tool. Adding the full-text query event helps isolate performance bottlenecks by providing detailed information on full-text query execution. Using the full-text query Profiler trace, you can see how expensive the full-text query is overall relative to other portions of your Transact-SQL query, and determine whether index lookup or some other part of the query could be optimized.

New Features for the Developer

Thesaurus Functionality

The new thesaurus functionality of SQL Server 2005 full-text search allows developers to reliably refine incoming full-text queries programmatically by expanding or replacing the incoming search term(s) with more appropriate or useful search terms. The full-text thesaurus eliminates the need for custom scripts to parse and expand or replace incoming query strings; the thesaurus is also easy to use through standard full-text query syntax. An example CONTAINS query that uses the thesaurus functionality follows:

Copy Code

```
SELECT ProductId, ProductName
FROM ProductModel
WHERE CONTAINS(CatalogDescription, ' FORMSOF(THESAURUS, metal) ')
```

Further, with FREETEXT queries, the thesaurus is invoked automatically as part of FREETEXT query processing, for example:

Copy Code

```
SELECT ProductId, ProductName
FROM ProductModel
WHERE FREETEXT(CatalogDescription, ' metal ')
```

The FREETEXT query invokes the thesaurus similarly to what was done in the CONTAINS query before it. The full-text thesaurus feature allows expansions and replacements to be configured for each language.

Configurable Accent Sensitivity

In previous releases, full-text catalogs and queries were accent sensitive by default. In SQL Server 2005, accent sensitivity is now a catalog-level setting that may be configured using the new full-text catalog DDL (mentioned above). For example, an accent-insensitive full-text catalog that is queried for "cafe" will match rows that have the keywords "café" and "cafe" stored in them. On the other hand, an accent-sensitive full-text catalog that is queried for the word "café" will only match rows that contain the word "café".

This accent sensitivity may be configured at CREATE FULLTEXT CATALOG time:

Copy Code

```
CREATE FULLTEXT CATALOG AwCat WITH ACCENT_SENSITIVITY=OFF
GO


Or by using ALTER FULLTEXT CATALOG:
```

Copy Code

```
ALTER FULLTEXT CATALOG AwCat REBUILD WITH ACCENT_SENSITIVITY=ON
GO
```

In some cases, applications would filter for certain query terms, and would expand a word like "café" to ' "café" OR "cafe" ' to find the both accented and unaccented forms of the word. With the configurable accent sensitivity in SQL Server 2005, this application-level processing is no longer necessary.

Query-Time Transformation of Noise Words

Application developers have long desired to be able to change the noise-word handling behavior of full-text search in contexts where users can input arbitrary full-text query expressions. The main area of pain has been CONTAINS-style queries, where a search expression such as ' "search" NEAR "the" ' ("the" is a noise word) would fail with the error, "Your query contained only noise words." The desired behavior in these cases

is to ignore the noise word, and process the query to simply look for the word "search".

In SQL Server 2005, a new **sp_configure** setting is introduced to allow developers and database administrators to relax some parts of full-text noise-word handling. By executing the following Transact-SQL statement, you can take advantage of the new transformation of noise words in CONTAINS queries:

Copy Code

```
EXEC sp_configure 'show advanced options', 1
GO
RECONFIGURE
GO
EXEC sp_configure 'transform noise words', 1
GO
RECONFIGURE
GO
```

A full description of the transform noise words feature is provided in SQL Server 2005 Books Online; however, with SQL Server 2005 full-text search, developers will no longer need to maintain a copy of the noise list in their applications and programmatically pre-parse query strings to remove noise words from user input.

Multi-Column Query

Full-text queries traditionally have been executed against a single column (by using the name as column specifier) or all columns of a particular table (by using an asterisk). Unfortunately, this one-or-all approach required developers to use multiple full-text query clauses for queries against only some, but not all full-text indexed columns of a table, which resulted in a separate query to the full-text engine for each clause. In SQL Server 2005, a column list can now be specified instead, allowing developers to query one, some, or all full-text indexed columns of a table using a single query.

Let's look at an example:

Copy Code

```
USE MyDb
GO

CREATE TABLE myFt_Table(pk INT NOT NULL, txtCol1 VARCHAR(1024), txtCol2 VARCHAR
(1024), txtCol3 VARCHAR(1024) )
GO

-- populate table with values

CREATE UNIQUE CLUSTERED INDEX idx1 ON myFt_Table(pk)
GO

CREATE FULLTEXT CATALOG myFtCat WITH ACCENT_SENSITIVITY=OFF AS DEFAULT
GO

CREATE FULLTEXT INDEX ON myFt_Table(txtCol1, txtCol2, txtCol3)
KEY INDEX idx1
GO
```

Above we've created a simple four-column table in some database. The pk column becomes the unique non-nullable single-column full-text key column, and we have three **varchar** columns that we have decided to full-text index. Now that the table and full-text index have been created and populated, let's look at some queries:

Copy Code

```
SELECT pk, txtCol1
```

```
FROM myFt_Table
WHERE CONTAINS(txtCol1, ' "foo" AND "bar" ')

SELECT *
FROM myFt_Table
WHERE CONTAINS(*, ' "foo" AND "bar" ')

SELECT pk, txtCol2, txtCol3
FROM myFt_Table
WHERE CONTAINS( ( txtCol2, txtCol3 ) , ' "foo" AND "bar" ')
```

Notice that in the third query above, the multiple columns are specified as a parenthesized, comma-separated list. That query is equivalent to the following query:

Copy Code

```
SELECT pk, txtCol2, txtCol3
FROM myFt_Table
WHERE CONTAINS(txtCol2, ' "foo" AND "bar" ')
OR CONTAINS(txtCol3, ' "foo" AND "bar" ')
```

Note that the above query uses multiple clauses to express the search condition (we are not searching against txtCol1). As mentioned previously, using multiple clauses requires the full-text engine to evaluate multiple full-text queries; in contrast, providing the column list as part of a single full-text query clause requires it to evaluate only a single query expression. Using this technique where possible will improve query performance and efficiency.

Query-Level Language Specification

SQL Server 2005 full-text search honors document-specified language settings in BLOB and XML documents at indexing time. For example, consider the following XML document:

Copy Code

```
<myXmlDoc>
<docTitle>
<docENUTitle xml:lang="en-us">
Yukon full-text search
</docENUTitle>
<docDEUTitle xml:lang="de">
Yukon full-text search (german equivalent)
</docDEUTitle>
</docTitle>
…
</myXmlDoc>
```

The **docENUTitle** element is specified to use the US English language as its language, whereas the **docDEUTitle** specifies German. At indexing time, the content of **docENUTitle** is tokenized by the US English word breaker, and the content of **docDEUTitle** is parsed by the German word breaker.

So what to do at query time? SQL Server 2005 full-text query syntax for the CONTAINS, CONTAINSTABLE, FREETEXT, and FREETEXTTABLE clauses all support a new LANGUAGE <*lcid*> parameter (specified as a numeric value or as a string) which may be used to override the column-default full-text language with the clause-level language. This clause-level language dictates which word breaker, stemmer, thesaurus, and noise word list to use for all terms of the full-text query. For example, if my column's default full-text language was set to 0 (Neutral), I could execute the following queries:

Copy Code

```
-- use neutral language (column default)
SELECT *
```

```
FROM ft_Table
WHERE CONTAINS(xmlFtCol, ' "Full-Text Search" ')

-- use English language
SELECT *
FROM ft_Table
WHERE CONTAINS(xmlFtCol, ' "Full-Text Search" ', LANGUAGE 'English')

-- use German language
SELECT *
FROM ft_Table
WHERE CONTAINS(xmlFtCol, ' "Full-Text Search" ', LANGUAGE 'German')
```

Because the neutral, English, and German word breakers tokenize text differently, it is desirable to use the correct word-breaking semantics for a query in English or German. Since keywords are indexed from the XML data using the English and German word breakers where specified, the language specification in the above queries becomes quite useful.

The LANGUAGE <*lcid*> parameter's value may be specified as a Transact-SQL variable if it is used in a full-text query clause in a stored procedure. For example:

Copy Code

```
CREATE PROCEDURE SearchDocuments
   @srchString nvarchar(1024),
   @language int,
   @topN int,
   @searchType smallint
AS
IF @searchType = 0
   SELECT [KEY], [RANK]
FROM FREETEXTTABLE(Document, Document, @srchString,
LANGUAGE @language, @topN)
ORDER BY [RANK] DESC
ELSE
   SELECT [KEY], [RANK]
FROM CONTAINSTABLE(Document, Document, @srchString,
LANGUAGE @language, @topN)
ORDER BY [RANK] DESC
GO
```

The above example searches against a Document table and passes in, as a parameter to the stored procedure, the search string, query-level language value (**int**), TOP_N_BY_RANK value, and the conditions under which to use CONTAINSTABLE or FREETEXTTABLE.

The above stored procedure makes the process of hooking this search up to a user interface much easier. The search string can be pulled from a text input field, query-level language can become a drop-down list of supported languages (this list may now be selected from the system catalog view **sys.fulltext_languages**), and the search type and TOP_N_ BY_RANK values can be easily set from drop-down boxes or radio buttons in the application.

Precomputed Rank

Developers using SQL Server 2005 full-text search can take advantage of a new optimization for FREETEXTTABLE queries that use the TOP_N_BY_RANK parameter (an example is shown above). As described earlier, there is a significant difference in the type of rank computation used for FREETEXTTABLE queries and the rank computation used for CONTAINSTABLE queries. The precomputed rank optimization in SQL Server 2005 allows FREETEXTTABLE queries to use rank values in a full-text catalog as opposed to having to calculate those values on the fly. As a result, these queries should increase to near CONTAINSTABLE speed the FREETEXTTABLE queries that use the TOP_N_BY_RANK parameter.

Precomputed rank is an sp_configure option that may be enabled as follows:

Copy Code

```
EXEC sp_configure 'show advanced options', 1
GO
RECONFIGURE
GO
EXEC sp_configure 'precompute rank', 1
GO
RECONFIGURE
GO
```

Enhanced Query Performance and Ranking Improvements

As mentioned in previous sections, the performance and scale of full-text search in SQL Server 2005 have been improved. Database developers and users of their full-text enabled applications should see an improvement over previous releases on the lines of 30-50 percent for straight full-text queries. The development team has focused on query performance improvements in the full-text engine itself, as well as the migration into the database engine of DocId-to-key-mapping facilities. Users should find that SQL Server 2005 delivers good improvement in query performance over previous releases.

Relevance scores returned from FREETEXTTABLE queries are higher quality than in previous releases. Refinements to scoring formulas used by full-text search, as well as improvements to the implementation of the Okapi BM-25 scoring algorithm, improve scoring over large and small data sets, as well as over multiple columns in a single table.

Conclusion

The SQL Server 2005 release signifies a significant step forward for the full-text search feature. Users of full-text search in SQL Server 2005 will find an enterprise-class, best-of-breed text-search facility that is easy to use and manage.

Additional Resources

For general information about SQL Server, see the Microsoft SQL Server website [http://www.microsoft.com/sql] .

White Papers

The following white papers provide additional information on key aspects of full-text search:

- Building Search Applications for the Web Using Microsoft SQL Server 2000 Full-Text Search [http://technet.microsoft.com/en-us/library/aa902674(printer).aspx]
- Full-Text Search Deployment [http://support.microsoft.com/default.aspx?scid=/support/sql/content/2000papers/fts_white paper.asp]

Sample Application

Included with SQL Server 2005 samples is an application, ItemFinder, that exercises new features of full-text search and demonstrates a number of full-text search best practices. The sample application includes full C# and Visual Basic .NET sources as well as Transact-SQL scripts.

Newsgroups

Developers and administrators can share questions and insights with other full-text search users on these Usenet newsgroups:

- microsoft.public.sqlserver.fulltext
- microsoft.beta.yukon.relationalserver.fulltext

Copyright