SQL Server 2008 Full-Text Search: Internals and Enhancements

SQL Server Technical Article

Writer: Fernando Azpeitia Lopez, Microsoft Corp.

Published: July 2008

Applies to: SQL Server 2008

Summary

Database systems must go beyond the traditional realm of relational data by covering an increasing amount and variety of unstructured and semistructured information, be it speech, documents, XML, bioinformatics, chemical, or multimedia. Search is a key technology capable of working with vast amounts of data: it is scalable, low-latency, and very user-friendly. It is just what is needed to make a database the best place to store all types of data.

SQL Server 2008 introduces a new Full-Text Engine that is integrated into the relational database. Full-text search is now as fully integrated as any other database service. The main goal for full-text search in SQL Server 2008 was to introduce a new integrated Search platform. In addition to the improvements and gains that come with this integration, this platform provides a strong base for delivering, in the near future, innovative new features and capabilities that combine the strengths of searching and querying.

This white paper covers in detail the new integrated full-text search architecture, new features, main changes from full-text search in SQL Server 2000 and SQL Server 2005, and limitations and best practices for deploying applications that use SQL Server 2008 full-text search.

Table of Contents

Introduction

This white paper contains a complete high-level description of the Microsoft® SQL Server® 2008 full-text search feature architecture, new functionality, compatibility, upgradeability, limitations, and best practices.

The new integrated full-text search architecture in SQL Server 2008 does not alter either the existing functionality or the command syntax of the full-text search functionality in SQL Server 2000 and SQL Server 2005. High-level features, including syntax, are not changed in SQL Server 2008. Low-level syntax and reference details about existing full-text search DDLs and DMLs are not covered here. SQL Server Books Online is an excellent reference to explore these.

This white paper discusses differences between SQL Server releases and considerations for deploying your application on top of the new integrated full-text search capabilities in SQL Server 2008.

Background

SQL Server has had full-text search capabilities since version 7.0. Later versions consistently improved the functionality, performance, scalability, and administration aspects of the feature set. SQL Server 2005 in particular achieved large gains in all of these areas.

Following is a brief description of what we accomplished with SQL Server 2005 in the area of full-text search**:**

· Full-text search allows fast and flexible indexing for keyword-based querying 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 the data, operating on words and phrases based on rules of a particular language.

· The performance benefit of using full-text search is best realized when querying against a large amount of unstructured text data. A LIKE query (such as '%fernando%') against millions of rows of text data can take minutes to return, whereas a full-text query (for 'fernando') can take only seconds or less against the same data, depending on the number of rows that are returned.

· Full-text indexes can 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 Web sites, product catalogs, news items, and other data), document management systems, and custom applications that must provide text search capabilities over data stored in a SQL Server database.

SQL Server 2005 full-text search scales from small mobile or personal deployments with relatively few and simple queries, 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 indexing and querying (by adding word breakers or stemmers from third-party vendors), 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 Microsoft Developer Network (MSDN) topics IFilter, IWordBreaker, and IStemmer.

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.

An underlying problem in SQL Server full-text query processing infrastructure is that it was based on MSSearch (part of the Microsoft Office group) technology, and therefore not integrated with the SQL Server query processor and storage engine. The lack of integration limits the performance of queries (such as queries that mix full-text and relational predicates), our ability to integrate with high availability and scalability functionality (such as database mirroring and partitioning), as well as our ability to build the new features that our customers have requested.

Following is the SQL Server 2005 full-text architecture:

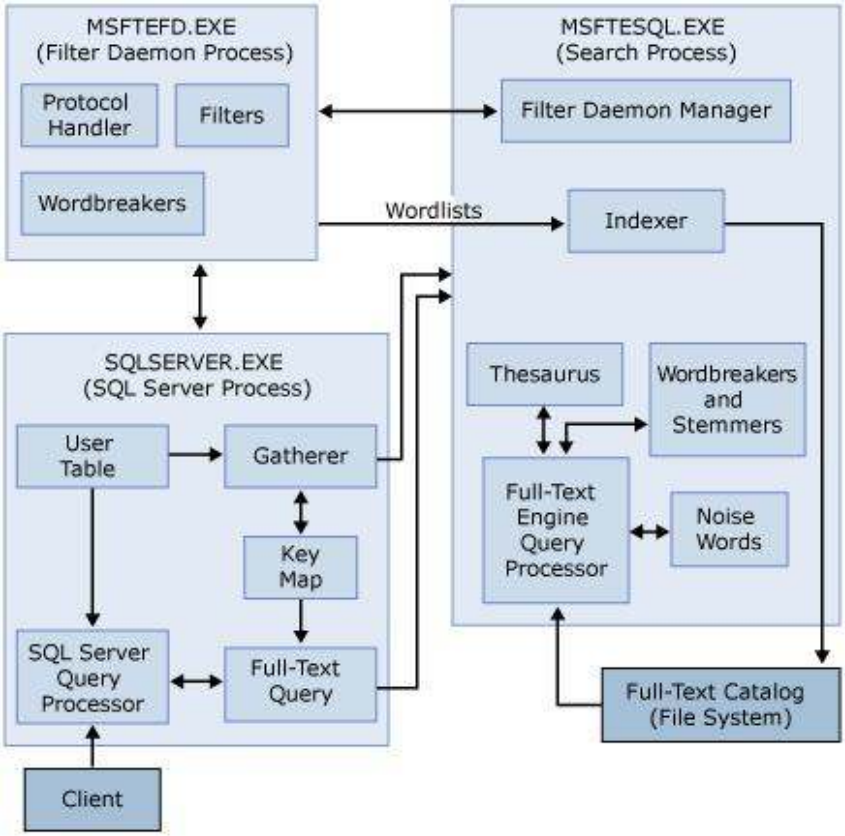

Figure 1

Note that the Full-Text Engine is outside of the SQL Server process, and therefore not fully integrated with

SQL Server capabilities such as scalability, security, storage engine, query processor, and so on.

Integrated full-text search in SQL Server 2008 addresses these limitations by introducing a full-text query processing component that is integrated with the server query and storage engine infrastructure. For the first time, the full-text search component is fully integrated and owned entirely by SQL Server, allowing us to start implementing specific important features that were not possible because of supportability challenges between different teams.

SQL Server 2008 full-text search architecture is based on the following high-level goals:

· **Integrated storage and management** – Full-text search is integrated directly with SQL Server inherent storage and management features. Full-text indexes are now stored inside database filegroups, rather than in the file system. The **data_space_id** column stores the filegroup identifier of the filegroup that contains the full-text index. To see in which filegroup a full-text index is stored, query the **data_space_id** column returned by **sys.fulltext_indexes**. Manageability operations for your database such as backups, automatically include the database full-text indexes. Stop word configuration has been migrated into new database objects (called stoplists) that also facilitate manageability tasks as well as improves integrity between different servers and environments.

· **Integrated query processing** – The new full-text search query processor is part of the Database Engine and fully integrated with the existing SQL Server query processor. The query optimizer is now aware of full-text query predicates and how to execute them as efficiently as possible.

· **Ease of administration and troubleshooting** – Typically, search engines contain many particularities and hidden treasures that make tracking problems and even understanding the results extremely difficult if not impossible. Full-text search now provides more analyzability to search structures such as a full-text indexes, the output of any given word breaker, noise word configuration, and so on.

SQL Server 2008 Full-Text Search Architecture

Figure 2 shows a high-level view of the architecture of SQL Server 2008 full-text search. Note that the Full-Text Engine and full-text indexes are now inside SQL Server. Only third-party code runs outside the SQL Server process; this is for security reasons.

This code is hosted by fdhost.exe.

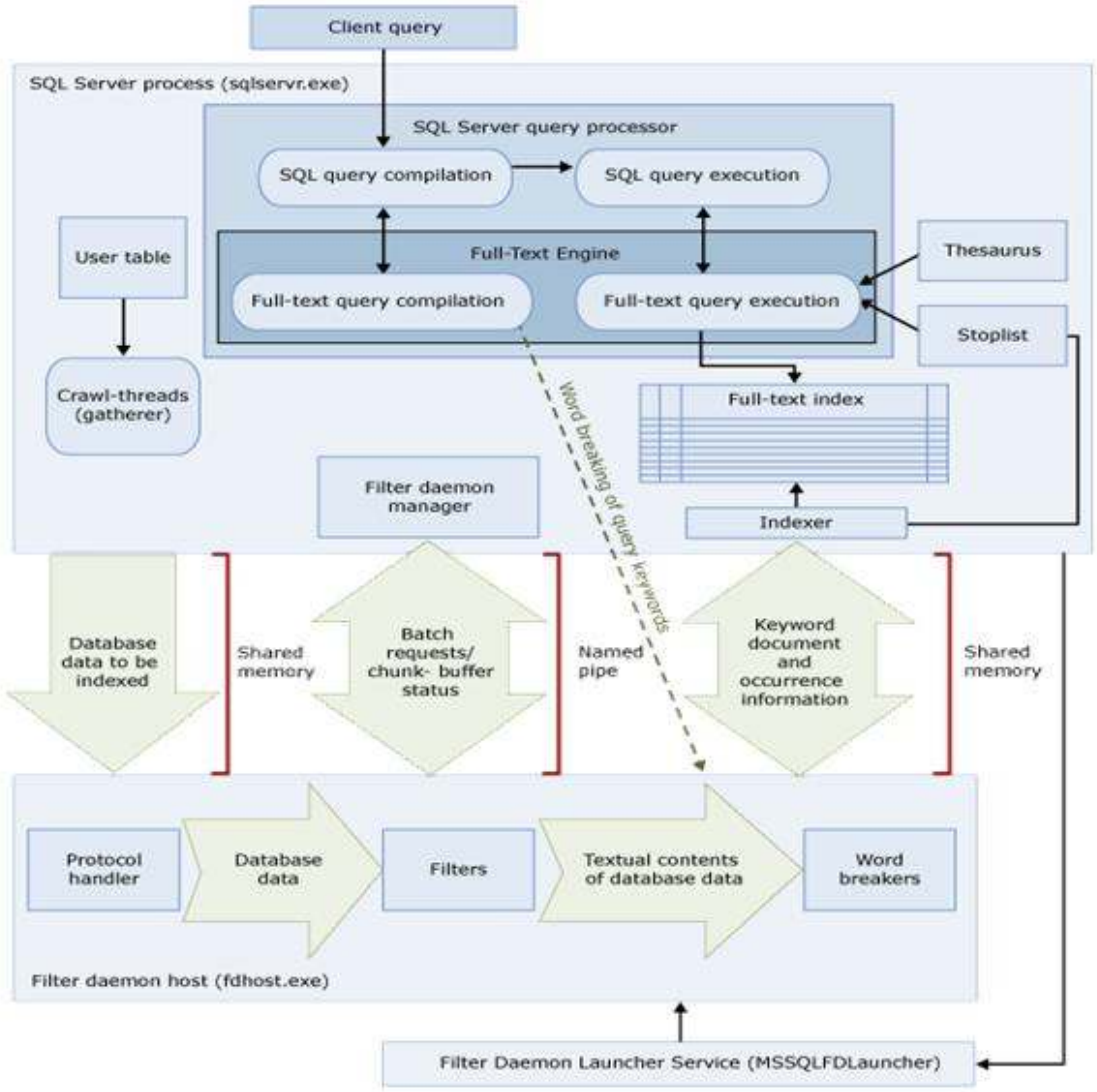

Figure 2

Note that the fdhost.exe process hosts third-party components that are not directly owned by SQL Server. Due to the nature of these components, we run them sandboxed in this process to protect the sqlserver.exe process. Because of this architecture, if any of these components were to fail or introduce a security threat, SQL Server security would not be compromised.

In previous versions of SQL Server, the Full-Text Engine ran under a different service that was separate from the SQL Server service. In addition, a separate process was responsible for loading third-party components. This process was named MSFTESQL.

Main Components of Full-Text Search in SQL Server 2008

Following are high-level descriptions of the main components related to the Full-Text Engine solution in SQL Server 2008.

Most of these components and their behavior are common to previous full-text implementations; because of the full integration of the SQL Server 2008 full-text search feature into the Full-Text Engine, most have been re-engineered internally to accomplish the integration.

Full-Text Engine	The Full-Text Engine manages the filter daemon host component, performs administrative operations, and executes full-text queries. For the first time in SQL Server history, this component runs exclusively inside the Database Engine.

Filter Daemon host (new in SQL Server 2008)	The full-text filter daemon host safely loads and drives third-party extensible components used for indexing and querying, such as word breakers, stemmers, and filters, without compromising the integrity of the Full-Text Engine. This process runs externally to the SQL Server service to protect the Full-Text Engine and SQL Server from possible unsecure third-party components.
Filter Daemon Launcher (MSSQLFDLauncher) **(new in SQL Server 2008)**	This service is responsible for creating filter daemon host processes when needed by the Full-Text Engine. For more information, see Understanding the Filter Daemon Process later in this paper.
Full-text index	A full-text index stores information about significant words and their location within a given column. This information is used to quickly compute full-text queries that search for rows with particular words or combinations of words. In SQL Server 2008, full-text indexes are integrated with the database engine, instead of existing in the file system as in previous versions of SQL Server. For more information, see "CREATE FULLTEXT INDEX (Transact-SQL)" in SQL Server Books Online.
Full-text catalog	For SQL Server 2008 databases, a full-text catalog is a logical concept that refers to a group of full-text indexes. The full-text catalog is a virtual object that does not belong to any filegroup.
Word Breaker	For a given language, a word breaker tokenizes text based on the lexical rules of the language. See "Word Breakers and Stemmers" in SQL Server Books Online.
Token	A token is a word or a character string identified by a word breaker.
Stemmer	For a given language, a stemmer generates inflectional forms of a particular word based on the rules of that language. See "Word Breakers and Stemmers" in SQL Server Books Online.
Thesaurus XML files	As in previous Full-Text Engine releases, SQL Server 2008 provides a thesaurus per each language. Thesaurus files are still XML files in the FTData folder of your SQL Server installation. For details, see Thesaurus Improvements later in this paper or see SQL Server Books Online.
Filter	Given a specified file type such as .doc, filters extract text from a file stored in a **varbinary(max)** or IMAGE column. For more information, see "Filters" in SQL Server Books Online.
Population (also known as a crawl)	Population is the process of creating and maintaining a full-text index. See "Full-Text Index Population Types" in SQL Server Books Online.
Stopword (new in SQL Server 2008)	A stopword is a word that is not relevant to your search and is filtered out from indexing and query processes. For example, words such as "a", "and", "is", and "the" are stopwords for the English locale. These words are ignored to prevent the full-text index from becoming bloated. See "Stopwords and Stoplists" in SQL Server Books Online.
Stoplist (new in SQL Server 2008)	A list of stopwords that, when associated with a full-text index, filters which words are indexed as well as which words are filtered from the queries on that full-text index. See "Stopwords and Stoplists" in SQL Server Books Online.

Upgrading your Full-Text Application to SQL Server 2008

While designing the new Full-Text Engine, we carefully kept all existing functionality from previous releases in order to facilitate upgrading applications to SQL Server 2008. Due the deep re-engineering in the full-text area, there are inevitable breaking changes that our customers will need to adapt to. However, in the vast majority of scenarios, users can upgrade their full-text application into SQL Server 2008 with zero work needed. Once upgraded, applications and scripts that use full-text capabilities will need no change in order to work because we kept all the functionalities provided in former versions of SQL Server full-text search.

This section focuses on considerations for upgrading your application into SQL Server 2008 full-text search. Full-text indexes are now fully integrated into the Database Engine instead of in the file system. This provides huge gains, but also means that you must keep a few things in mind when upgrading your former full-text

catalogs with their indexes to SQL Server 2008.

Upgrading Existing Full-Text Catalogs from Previous Releases

SQL Server 2008 introduces a new Full-Text Engine that is completely integrated with the SQL Server engine. An important consequence is that full-text indexes now exist in the SQL Server architecture, and not in the file system. This means you must migrate existing SQL Server 2000 or 2005 full-text catalogs (and their indexes) into this new full-text index format integrated into the engine.

Full-text catalogs coming from previous releases are of no use in SQL Server 2008. A process is needed before existing data that is indexed by using full-text can be available in the integrated full-text search component of SQL Server 2008.

Following are the ways you can upgrade your full-text catalogs into SQL Server 2008:

- **Perform an in-place upgrade**

In an in-place upgrade, the existing SQL Server instance is replaced by the new SQL Server 2008 one. Existing full-text catalogs must also be upgraded to the new architecture.

- **Detach and attach the database**

Detach a database containing full-text catalogs, and then attach it to a given instance of SQL Server 2008. The full-text catalogs must also be upgraded to the new architecture. For more detailed information on how to detach, and then attach a database containing full-text catalogs from SQL Server 2000 or 2005 to SQL Server 2008, see SQL Server Books Online.

- **Backup and restore the database**

Restore a database that has been backed up in a former version of SQL Server. If this database contains full-text catalogs, these must also be upgraded to the new architecture when you restore.

Note that in SQL Server 2008, there is no concept of backing up or restoring a full-text catalog between 2008 instances, as this is just a logic concept. You can, however, back up or restore individual full-text indexes in a given database filegroup. You just need to specify the particular files within a given filegroup that contain the full-text index. At full-text index creation time, the user can specify in which filegroup the full-text index should be created. For more information, see SQL Server Books Online.

- **Copy the database by using the Copy Database Wizard**

Copy an existing database with full-text catalogs into a SQL Server 2008 instance. The existing full-text catalogs must be upgraded as well.

SQL Server 2008 introduces the new **upgrade_option** server property that specifies the course of action to take when upgrading full-text catalogs. Following are the ways you can upgrade your full-text catalogs:

- **Use full population (rebuild)**

Rebuild the SQL Server 2000 or SQL Server 2005 full-text catalogs. By default, this option triggers a full population of the full-text catalogs that are involved. A full population of a very large full-text catalog can take significant time and resources to finish. During this process, a perfect recall of a query that runs against any of the full-text indexes that are being repopulated as part of the rebuild process cannot be guaranteed. The query will return the existing results indexed up to that point during the indexing/rebuild process.

- **Import the indexed data (default option)**

Import the indexed data from full-text catalogs to insert it into the new full-text indexes in the SQL Server 2008 architecture. It should take only a matter of hours (depending on your hardware) to copy a very large full-text search catalog into the new full-text search indexes because there are no filtering or word breaking processes to perform.

The data is directly imported from the previous index structure to the new index structure. If a full-text catalog is not available, the associated full-text indexes are rebuilt. This process can be up to approximately ten times faster than the rebuild process.

Until this process is completed, the results of any full-text query run against these indexes cannot be guaranteed.

Note that this process can be very CPU intensive. If you must pause the import process to resume it later, use the **pause_indexing** property in **sp_fulltext_service**. To monitor the import status, see the **import_status** property of the FULLTEXTCATALOGPROPERTY scalar function.

There is no **Import** option when you upgrade an instance of SQL Server 2000 to SQL Server 2008. Under this scenario, all existing full-text catalogs are rebuilt or reset during the upgrade, depending on the upgrade option.

- **Reset the full-text catalogs**

Resetting your full-text catalogs deletes the existing full-text catalogs belonging to the database that is being upgraded, restored, or attached to SQL Server 2008. After the upgrade, all full-text indexes are disabled for change tracking and crawls are not started automatically.

During this type of upgrade, your search application is partially offline. This option performs a fast upgrade and leaves search offline after the upgrade is complete. You have the option to later rebuild the catalogs.

Upgrade Options: Semantic Consistency

If importing the indexed data is a much faster upgrade process, why would you use the full population (rebuilding) method to upgrade full-text catalogs? The reason is related to several new *word breakers* and *stemmers* introduced in SQL Server 2008 full-text search. (These are covered in more detail later in this white paper.)

For some scenarios or languages used, it might be appropriate to repopulate (rebuild) the entire full-text catalog using the new word breakers. Using 'Import' as a different method of migrating the catalogs might cause semantic inconsistency at query time due to the different word breaking behavior of the word breakers that were used to index the imported full-text catalog, and the new word breakers in SQL Server 2008 that parse the current queries.

If you issue a full-text query that looks for a phrase that is broken differently by the word breaker in previous versions of full-text search and the new SQL Server 2008 word breakers, a document or row containing the phrase might not be retrieved. This is because the word breaking during indexing in the former SQL Server version was performed by a different logic than in SQL Server 2008 full-text search at query time.

The solution is to repopulate (rebuild) the full-text catalogs with the new word breakers so that index and query time behavior are identical.

Important Note: This semantic inconsistency does not apply to all word breakers for languages (LCIDs) shipped with SQL Server 2008. The following word breakers, shipped in SQL Server 2008 are identical to the ones shipped in SQL Server 2005:

English	Turkish
English UK	Simplified Chinese
Thai	Traditional Chinese
Korean	Chinese (Hong Kong)
Danish	Chinese (Macau)
Polish	Chinese (Singapore)

Importing a SQL Server 2005 full-text catalog that uses only these languages cannot cause semantic inconsistency. If this is the case, set the **Upgrade** option to **Import** for better performance at upgrade time.

Following is what happens during the upgrade, depending on which upgrade option you select:

· **Rebuild**

SQL Server 2008 repopulates the full-text catalogs by using the new word breakers, which will be used from now on at query time. Following are the advantages and disadvantages of this option:

· There is no semantic consistency issue because the new word breakers are set as the default ones from now on for index and query time.

· Users will be using fully supported word breakers, which have more advanced linguistic behavior than those in previous releases.

· The full-text application cannot be running during the full population. If the full-text catalog is huge, the upgrade can take many hours depending on the hardware. Note that indexing performance has been improved in SQL Server 2008, which somewhat reduces this first population time.

· **Import**

SQL Server 2008 imports the existing full-text catalogs into the new format. Then the new word breakers are used to index any new full-text indexes, as well as for future queries. Following are the advantages and disadvantages of this option:

· The full-text application will be down for a considerably shorter time as compared to the full population (rebuild) method. Even with very large full-text catalogs, this method should take no more than a few hours for most large deployments.

· Users will not be using fully supported word breakers in some cases. Only security issues are addressed by the former word breakers. Also, the former word breakers have inferior linguistic quality compared to the new ones.

· The semantic inconsistency previously described might occur because different word breakers are used at indexing and query times.

The new word breakers are provided by the Microsoft Natural Language Group. Other important Microsoft products (Windows® and the Microsoft Office System) have migrated as well to this new generation of word breakers.

Upgrading Thesaurus Content and Noise Word Lists

Upgrading Customized Noise Files to SQL Server 2008

At upgrade time the noise word files in the former FTData folder are kept in the following folder:

SQL_Server_install_path\Microsoft SQL Server\MSSQL.1\MSSQL\FTDATA\FTNoiseThesaurusBak

Note that SQL Server does not automatically migrate the noise file content into a new or existing STOPLIST object in SQL Server 2008. Currently, it is the user's responsibility to create a new stoplist (or update an existing one) with the altered content of the former noise files per a given language.

Following are the possible upgrade scenarios as they relate to noise file manageability:

· If the default noise file content for any used language has never been altered, there is no need to keep these noise files after upgrading because the default SYSTEM STOPLIST already contains the exact same noise words.

· If the content of the existing noise files for one or more languages that are used in any of the existing full-text indexes has been altered, additional steps are required to create an identical user experience after the upgrade. By default, the upgraded full-text indexes are associated with the default SYSTEM STOPLIST.

Create a new stoplist (copy the SYSTEM STOPLIST if necessary) that contains the <noise_word, LCID> pairs specified in the noise files for previous SQL Server versions. A simple procedure that parses the former noise file content and adds it to or removes it from this new stoplist will suffice. After this is done, use DDLs to associate the new stoplist with the desired full-text indexes. Note that a full population for the full-text index is recommended any time the user alters the stoplist the full-text index is associated with.

Upgrading Customized Thesaurus Files to SQL Server 2008

After upgrading, thesaurus files are in the following folder:

SQL_Server_install_path\Microsoft SQL Server\MSSQL.1\MSSQL\FTDATA\FTNoiseThesaurusBak

SQL Server does not replace the newly installed default thesaurus files with any previously customized thesaurus files. Currently, it is the user's responsibility to update the default thesaurus files with any changes. If you have rights to alter an existing thesaurus file for a given language, simply paste the content of the previously altered thesaurus file in the FTNoiseThesaurusBak folder into the new thesaurus in the FTData directory.

SQL Server 2008 introduces new tools that enable you to alter a thesaurus file and make the Full-Text Engine pick up the changes without the need to restart the Full-Text Engine and without causing application downtime. More details are available in Thesaurus Improvements later in this white paper.

Understanding the Filter Daemon Process

To protect the SQL Server process from third-party components, the full-text search component loads and executes external components (word breakers, stemmers, and filters) under a different process from SQL Server. This process must have low privilege settings in order to provide a secure running environment. In previous releases, the MSFTEFD processes accomplished a similar task. SQL Server 2008 introduces the *filter daemon host* (fdhost.exe) process, responsible for sandboxing these components.

The MSSQLFDLauncher service, accessed through Configuration Manager, is responsible for starting and stopping the filter daemon process. The only responsibility of this service is to launch fdhost.exe processes when needed, assigning them the correct account/password information and privileges.

Due to security improvements in Windows Vista® and Windows Vista Server (Longhorn), when you install SQL Server 2008 and select to install full-text search, the MSSQLFDLauncher service automatically runs under a local service account, propagating the settings of this account to the fdhost.exe processes created by this service. The fdhost.exe processes can then serve the Full-Text Engine needs at word breaking, filtering, and stemming times.

When installing SQL Server 2008 with full-text search on Windows Server® 2003, Setup prompts for a name and password for the low-privilege account to be created to host this fdhost.exe process. The MSSQLFDLauncher

service uses these credentials to start the fdhost.exe process for full-text needs when necessary.

Do not use the same account for this service that you use to run other services. Typically, these accounts have many privileges that the fdhost.exe process does not need. Granting too many privileges can have security implications, for instance if the fdhost.exe process hosts unsecure third-party components**.** We strongly recommend providing a low-privilege account for the MSSQLFDLauncher service when installing SQL Server 2008 in Windows Server 2003.

Note that Setup through the command line also supports as parameters the name and password of the account used by the MSSQLFDLauncher to create the fdhost.exe process. The functionality is equivalent to Setup through the UI.

Updating the Filter Daemon (FDHOST) Credentials

As with other services, you can use the Configuration Manager tool to manage the MSSQLFDLauncher service to control fdhost.exe credentials (account and password). However, restarting this service does not shut down and create a new fdhost.exe using the updated credentials. You need to execute the following in SQL Server:

exec sp_fulltext_service 'restart_all_fdhosts'

This stored procedure shuts down any existing fdhost.exe processes running on the given instance. Immediately after, it requests that the MSSQLFDLauncher service create a new fdhost.exe for that instance.

Following are the basic steps for updating the account/password on which the fdhost.exe process will run:

1. In Configuration Manager, update the MSSQLFDLauncher credentials. Make sure the account and password provided exist and are correct.

2. Execute sp_fulltext_service 'restart_all_fdhosts' to force the shutdown of the existing fdhost.exe process. This triggers the creation of a new fdhost.exe process on that instance using the recently provided credentials assigned to the MSSQLFDLauncher service. Note that this operation cancels any full-text running activity and restarts it when possible after the new fdhost.exe process is running.

3. For important details on how to configure and troubleshoot issues related to FDHOST and MSSQLFDLauncher, see "Set the FDHOST Launcher (MSSQLFDLauncher) Service Account for Full-Text Search (SQL Server Configuration Manager)" in SQL Server 2008 Books Online.

SQL Server 2008 Full-Text Search Improvements

Improvements to full-text search fall into two categories: brand new functionality in SQL Server 2008 and improvements to the existing functionality in previous versions.

New Functionality

The main goal in the full-text search area for SQL Server 2008 was to deliver our new integrated Full-Text Engine. The vast majority of our work focused on providing SQL Server with full-text search functional parity under the new integrated full-text search architecture.

To accomplish this integration, certain functionality and changes were needed besides entirely re-engineering the component. The section describes the main new functionality added to SQL Server 2008 for full-text search.

Introducing Stoplists

Within text, there are typically many words that are almost never relevant to a search. Some of these words are "the", "a", and "an."

For example, in the following text:

"The SQL Server architecture scales better than any other database platform in the current market"

We can identify the relevant **words** and the non relevant ones:

*"The **SQL Server architecture scales better** than any other **database platform** in the current **market**"*

The list of terms for a particular language that are not relevant is called a *noise word list*. Noise word lists specify which tokens to ignore for a particular language.

At indexing and query time, external components called *word breakers*, are responsible for breaking each token (word) from the text and, using noise word lists, returning only the relevant ones, ignoring the others. In Shiloh and Yukon, this list was maintainable by full-text search customers so that they could add or suppress different terms.

In SQL Server 2005 full-text search, noise words were configured by means of individual files stored in the file system. Noise word configuration was by language, at the instance level, and the only method of accessing and

changing these configurations was through standard text editing tools.

In the new integrated full-text search, it makes sense to move the noise word lists into the database. This new type of database object is called a *stoplist*.

The advantages of this approach includes easier management (via Transact-SQL), improved change tracking (notifications), the use of special named objects as opposed to arbitrary per-language resources, the ability to query, consistency, and transportability (the lists are stored in the database and are transported at attach/detach time, and so on). The implications of this architectural shift are outlined below.

Usability: Easy access to view, add, modify, and delete items from lists.

Programmability: Items are programmatically accessible through Transact-SQL.

Manageability: Lists are no longer arbitrarily bound to a particular language. Named lists can be created to line up with business and application logic requirements that may vary widely in a single instance.

Transportability: Lists are stored in the database and can be moved with the database.

Migration: Lists from previous product releases, as well as competitor's products can be easily migrated, automatically, into integrated full-text search.

Security: Lists are stored and managed within the database, and have the same security rules applied to them as other database objects (such as tables).

Integration: The integrated full-text search architecture in SQL Server enables developers to build sophisticated applications.

This design adds a new dependent object type (STOPLIST) and an internal table that is viewable by the user. These objects are bound to full-text indexes.

From a functional point of view, the user can do the following:

· **Create a stoplist**

You can define a new stoplist, which can be the system default stoplist per instance or a user-specific one. You can specify whether this is a new stoplist or a copy of an existing one, including the system default stoplist.

· **Alter a stoplist**

You can insert or delete *stopwords* (noise words to be ignored by the word breakers) in any full-text stoplist other than the system one. You can insert a noise word for a given language, delete a word in a given language, delete a given word for all languages present in the stoplist, or delete any word belonging to a given language (for all practical purposes removing the language from the stoplist).

· **Delete a stoplist**

You can delete an existing stoplist unless it is associated with an existing full-text index.

· **Attach and detach a stoplist to a full-text index**

Create full-text index DDL is augmented with the stoplist option. This way, a given full-text index can be associated with any stoplist.

· Alter full-text index DDL is augmented with the stoplist option. The association between a given index and its stoplist can be updated at any time by disabling the stoplist for a particular full-text index and associating a different stoplist with the index. This automatically issues a full population of the index, unless specified otherwise in the DDL.

Types of Stoplist Objects

There are two types of stoplists:

· **User-created stoplist**

This is a stoplist created by any user who has rights to do so. You can create an empty stoplist or copy an existing one (if you have reference permissions on the target stoplist). The new stoplist can be associated to any full-text index the user has rights to alter.

· **Default system stoplist**

This is the populated stoplist available out of the box. This stoplist contains exactly the same content, for each language, that is in the noise files included out of the box in SQL Server 2005 full-text search noise files. By default, all imported full-text catalogs as well as new full-text indexes created are associated with this single stoplist. This stoplist is not alterable. To customize noise words, create a new stoplist, copy from the system one, and then remove or add stopwords for any language in the new stoplist. After modifying the new stoplist,

associate it with the full-text indexes that are required. Note that after modifying the stoplist association of a full-text index or after modifying a stoplist that is already associated to a given full-text index, it is recommended that you perform a full population of the full-text catalog that contains that full-text index. This way, the new stoplist content filtering kicks in and perfect recall is guaranteed.

For a complete stoplist command list and syntax, see Stopwords and Stoplists [http://msdn.microsoft.com/en-us/library/ms142551(SQL.100).aspx] in SQL Server 2008 Books Online.

Thesaurus Improvements

This section focuses on the main differences between the SQL Server 2005 thesaurus capabilities and the new ones in the SQL Server 2008 thesaurus

To become familiar with the thesaurus, see SQL Server Books Online, which provides examples and use cases of how to configure the thesaurus for your given language, as well as its syntax and several manageability aspects.

The SQL Server thesaurus component has not changed in SQL Server 2008 as much as the noise file configuration did. We realize the importance of migrating these files into internal database objects. Due to time constraints, we could not finish this integration, thus, SQL Server 2008 still has the thesaurus configuration based in XML files existing in the FTData folder of your SQL Server installation directory tree:

SQL_Server_install_path\Microsoft SQL Server\MSSQL.10.MSSQLSERVER\MSSQL\FTDATA\

However, certain improvements have been accomplished in this important component.

Maintaining Thesaurus Content

In SQL Server 2008, we lazily cache the content of each thesaurus file in internal tables under the **tempdb** database. At query time, we look up these internal tables for efficient expansion or replacement operations. If a thesaurus file has not been loaded, it is automatically loaded at query time.

In SQL Server 2005, to change thesaurus content, you had to edit the XML file, save it, and then restart the MSFTESQL service in order to restart the Full-Text Engine and pick up the thesaurus changes. This is no longer necessary in SQL Server 2008. After you update the XML file of a given thesaurus for a particular language, you can use the **sp_fulltext_load_thesaurus_file** stored procedure 73a309c3-6d22-42dc-a6fe-8a63747aa2e4 to load the content of the thesaurus file into **tempdb**. Note that you must specify the local identifier (LCID) that corresponds to the language of the thesaurus file that you updated. For example, for the English thesaurus file, Tsenu.xml, the corresponding LCID is 1033.

sys.sp_fulltext_load_thesaurus_file ***1033***

This public stored procedure parses the XML file and inserts rows in the internal tables by using XQuery. Note that thesaurus files will be loaded lazily as well. This means that we load the thesaurus file the first time a query uses it. This method saves resources. If the thesaurus file is so large that the loading time is unacceptable for the query response time, you can load it by using the stored procedure.

Note that to update the thesaurus XML file and call the stored procedure, you must be in the Administrator role on the server where SQL Server is running.

Updating the language components

The following stored procedure exists in SQL Server 2005 and can be used to update/refresh the registry information for language-dependent files such as thesaurus files.

sys.sp_fulltext_service ‘update_languages’

After you update the registry with information related to a language component (such as adding a new word breaker for a given LCID, or changing the location of the thesaurus XML file per a given LCID), invoke this stored procedure to refresh SQL Server as well. **sys.sp_fulltext_load_thesaurus_file** LCID loads the thesaurus file for the new LCID added or for an existing one for which the thesaurus location file was changed.

Note that you do not need to execute this stored procedure for regular thesaurus content changes.

Thesaurus Algorithm Characteristics

Besides the existence of expansion and replacement rules that apply to single terms in phrases, some cases need clarification. Following are some details of these subtleties.

· Only the original phrase/term is used in thesaurus matching. The original term is decided by the word breaker. For example, the query is “tic-tac-toe”, which is broken as (“tic tac toe” OR “tic tac toes” OR “tictactoe”) and “tic tac toe” is the original phrase as emitted by the word breaker. In this case, if there is a replacement (or expansion rule) R(“tictactoe” -> “X and O”), this rule is not used.

· The thesaurus is not recursive. Only the chosen rule is applied, ignoring any other rule that might apply to the result of the first one.

For example, if we search for 'Intranet', we can see from the following code that it is replaced with 'Internet'. There is an expansion rule for 'Internet' ,'IE5'. However, 'IE5' is not part of the final search terms—only 'Internet' is.

```
<expansion>

<sub>Internet</sub>

<sub>IE5</sub>

</expansion>


<replacement>

<pat>Intranet </pat>

<sub>Internet</sub>

</replacement>
```

· Thesaurus matching occurs after inflectional expansion. This differs from SQL Server 2005 thesaurus behavior. Hence, if the query is Q("seek") and there is a thesaurus replacement entry R(seek->went), then:

· In SQL Server 2005, the query is first converted to Q("went"), which would become Q("went" OR "go") assuming 'go' is an inflectional expansion of 'went'.

· In SQL Server 2008, the query is converted to Q("seek") -> Q("seek" OR "sought") -> Q("went"). Note that it does not search for "sought" because the original term suffered a replacement, thereby making the inflectional forms of the original term "seek" invalid for search. In the case of an expansion, SQL Server 2008 full-text search searches:

1. The original term or terms

2. Its expansions

3. The inflectional forms of the original term or terms

· SQL Server 2008 has a single global thesaurus that is applied for every thesaurus invocation, regardless of which language the query is working in.

If the rules for the global thesaurus and the local thesaurus for a particular language overlap, the local thesaurus rule is given preference.

For example, if the local thesaurus has a rule R(A -> B) and there is a global thesaurus rule (A ->C) and the original query is Q(A P D), the new query is Q(**B** P D).

· When overlap between rules exists, the **longest matching** rule wins over any others.

If you issue the following full-text query:

```
SELECT Description

FROM Production.ProductDescription

WHERE CONTAINS(Description,'FORMSOF (THESAURUS,Tech,"Internet Explorer is great") ');
```

The full-text search thesaurus finds two rules that match part of the query phrase:

```
<expansion>

<sub>Internet Explorer</sub>

<sub>IE</sub>
```

_{IE5}

</expansion>

<replacement>

<pat>**Internet**</pat>

_{intranet}

</replacement>

Because the first rule that satisfies part of the query is longer, it is picked up. Therefore, you see Product descriptions that contain the following phrases:

-Internet Explorer is great

-IE is great

-IE5 is great

· If two overlapping matches are the same length, the one that occurs earlier in the query is chosen.

· Thesaurus expansions and replacements are always case-insensitive.

· If there is a mismatch between the accent sensitivity of the full-text index and the thesaurus:

· If the thesaurus is accent-sensitive and index is not, accented terms in the query will not find a match in the thesaurus. Also, if a query is expanded to include accented terms from the thesaurus, these terms will not find a match in the index.

· If the thesaurus is not accent-sensitive and the index is, accented terms in the query will not find a match in the thesaurus. Also, if a query is expanded to include terms from the thesaurus, only the un-accented versions of those terms are found in the index.

Restrictions and Recommendations

The following restrictions and recommendations apply to editing a thesaurus file:

· When editing thesaurus files by using text editor tools, the files must be saved in Unicode format and Byte Order Marks must be specified.

· It is recommended that entries in the thesaurus file not contain special characters. This is because word breakers have subtle behaviors in relation to special characters; in combination with thesaurus matching, these can have behavioral implications for the full-text query.

· It is recommended that <sub> entries in a thesaurus file do not contain stopwords. This is because we expand the query to include the <sub> entries from a thesaurus file and there is no point in doing this if the <sub> entry contains stopwords since stopwords are not present in the full-text index.

· Thesaurus entries cannot be empty or word break to an empty string.

· Thesaurus entries can be no longer than 512 Unicode characters.

· A thesaurus cannot contain duplicate entries among the <sub> entries of expansion sets and <pat> elements of replacement sets. Since this leads to ambiguity in expanding a full text queries, a warning message in the error log mentioning that only the first occurrence of the phrase in the thesaurus file will be retained and the rest discarded, is logged.

New Tools for Troubleshooting SQL Server 2008 Full-Text Search

Many customer requirements exist around the capability of understanding and troubleshooting specific scenarios while configuring or using full-text search. SQL Server 2008 introduces tools helping to accomplish this important goal.

Access to the Full-Text Index Content

One of the most common requests coming from our customers is to provide a way to extract the raw data stored in the full-text index. So far, this full-text index has been inaccessible to our customers and used only to satisfy

full-text queries against it.

As applications using full-text search become more and more complex (complexities include storing different types of large amounts of data, or allowing the loading of external components such as filters and word breakers where the behavior is not always transparent), a set of tools for debugging and displaying the information stored in the full-text index becomes very important.

Two dynamic management functions, **sys.dm_fts_index_keywords** and **sys.dm_fts_index_keywords_by_document**, enable you to view content in full-text indexes in SQL Server 2008.

Following are the main goals of this functionality:

· To expose the relevant content of any full-text index in SQL Server. This function receives the database and table that the full-text index being requested covers.

· These functions are not designed for end users, but only for investigation and debugging tasks usually performed by the administrator of the SQL Server installation. The compatibility of these feature as-is in future releases is not guaranteed. The behavior of these functions and/or the fields or columns exposed in full-text indexes may differ from release to release.

· Because the terms stored in the full-text index are in their normalized form (not human readable), these functions perform internally the needed operations to expose the denormalized version of the term stored in a full-text index. This way the user can easily recognize a particular term in both of its forms.

To describe this functionality, several use cases follow. For the use cases, we have the following table, named Documents, which contains two rows with the following data:

Id	ftcol
0	*Company*
1	*Company visit*

This table has an associated full-text index on the ftCol column.

Displaying the full-text index content at the keyword level

sys.dm_fts_index_keywords

You want to see the content of the full-text index associated to the table named Documents, which is in the Demo database.

You want to see the information in the full-text index at the keyword level, regardless of which rows are contained in the index.

Issue the following statement:

```
select * from sys.dm_fts_index_keywords(db_id('Demo'), object_id('Documents'))
```

This query selects all the data returned by the function **sys.dm_fts_index_keywords**:

· keyword

· display_term

· column_id

· document_count

The output is the following:

Keyword	display_term	column_id	document_count
0x0063006F006D00700061006E0079	company	2	2
0x00760069007300690074	visit	2	1
0xFF	END OF FILE	2	2

Note that **OxFF** represents the special character for the end of the file or dataset.

Displaying the full-text index content at the document level

sys.dm_fts_index_keywords_by_document

You want to see the content of the full-text index associated to the table Documents in the Demo database.

You also want to see the information in the full-text index at the document/row level. Issue the following statement:

```
select * from sys.dm_fts_index_keywords_by_document (db _id('Demo'), object_id('Documents'))
```

This query selects all the data returned from the function **sys.dm_fts_index_keyword_by_ document**:

- keyword
- display_term
- column_id
- document_id (see the following note)
- occurrence_count

The output is the following:

keyword	display_term	column_id	document_id	occurrence_count
0x0063006F006D00700061006E0079	company	2	0	1
0x0063006F006D00700061006E0079	company	2	1	1
0x00760069007300690074	visit	2	1	1
0xFF	END OF FILE	2	0	1
0xFF	END OF FILE	2	1	1

Note: When your full-text key is an integer type (which is our recommendation), **document_id** maps directly to this value in the base table. However, if the full-text key is a non-integer type, **document_id** will not represent the full-text key in the base table. To identify the row in the base table returned by this dynamic management view (DMV), you must join it with the results returned by **sys.sp_fulltext_keymappings**. You must store this stored procedure output in a temporary table before you can join the **document_id** column with the **DocId** column returned by the stored procedure. For more information on this stored procedure, see the "sys.sp_fulltext_keymappings" topic in SQL Server Books Online.

SQL Server 2008 has two different functions for accessing full-text index content because of different requirements and for performance reasons.

To display the data stored in the full-text index at the document/row level, we must decompress the index fragments containing the low level information. If the full-text index is huge, this operation can be costly.

Following is the level of detail that each function presents:

- **sys.dm_fts_index_keywords**: (unique) term level of the data
- **sys.dm_fts_index_keywords_by_document**: term per document level of the data

Following are typical user scenarios for accessing full-text index content and which of these functions is suitable:

- **sys.dm_fts_index_keywords** (content at the keyword level)
- "I want to know if a keyword is part of the full-text index"
- "I want to know how many docs/rows contain a given keyword"
- "I want to know how many doc/rows are indexed in the full-text index" (document_Count of 0xFF)
- "I want to know the most common keywords in the full-text index to declare them as noise words" (docCount of each keyword versus total docCount)

- **sys.dm_fts_index_keywords_by_document** (content at the keyword/doc level):
- "I want to know how many keywords the full-text index contains"
- "I want to know if a keyword is part of a given doc/row"
- "I want to know how many times a keyword appears in the whole full-text index" (sum(occurrence_Count) where keyword=……)
- "I want to know how many times a keyword appears in a given doc/row"
- "I want to know how many keywords a given doc/row contains"
- "I want to retrieve all the keywords belonging to a given doc/row"
- Scenarios not covered in this release:
- "I want to know the offset (word or byte) of a given keyword in a given doc/row"
- "I want to know the distance (in words) between two keywords per a given doc/row"

For a complete reference on the syntax and data types of these dynamic management functions, see the "sys.dm_fts_index_keywords_by_document" and "sys.dm_fts_index_keywords" topics in SQL Server Books Online 2008.

The Full-Text Parser

Many full-text search users struggle to understand what the final output stored in the full-text index is. This is because the raw data is not stored as-is in the full-text index. Rather, a set of components parse the data and output the final set of keywords to be indexed. The most relevant of these components are the word breakers. These components are language specific and have the ability to extract the correct terms contained in a give text in a given language.

Customers often do not understand how word breakers work for certain input and word patterns—a tool that facilitates this investigation is very useful.

This tool in SQL Server 2008 also enables you to transform the input, including into the transformation pipe, for other full-text search components such as stoplists and the thesaurus. Users can choose which components to include in the process and see the final output out of the original text/query.

sys.dm_fts_parser is a dynamic management function that returns the final tokenization resulting after applying a given word breaker, thesaurus, and stoplist combination to a given query string input. The output is identical to the output that would result if the specified given query string were issued against the Full-Text Engine.

The main goals of this tool are:

- To output the final set of keywords that are the result of applying a word breaker.
- To support any query that is consistent with CONTAINS syntax. This means that logical operators, thesaurus, and other specific full-text options are allowed as part of the input query.
- To enable the user to specify a stoplist to use against the function input.
- To enable the user to specify whether the input should be treated with accent sensitivity on or off.

These functions are not designed for end users, but only for investigation and debugging purposes performed by the administrator of the SQL Server installation. SQL Server does not guarantee the compatibility of these feature as-is in future releases. The behavior of these functions and/or the fields or columns in the full-text index that are exposed may differ from release to release.

Because the terms stored in the full-text index are in their normalized form (not human readable), these functions perform internally the operations needed to expose the denormalized version of the term stored in a full-text index so that you can easily recognize a term in both its forms.

Displaying the output of a given word breaker considering stoplist filtering

You want to see the output after using a specific word breaker. You input the following:

' "*The Microsoft business analysis" OR "MS revenue"* '

You choose English (LCID=1033) as the parsing language and use a stoplist named 'MyStopList' (with ID =77). Assume that the word 'The' is listed in this stoplist for English. When the stoplist id=0, the system stoplist is used.

This is the Transact-SQL statement:

select * from sys.dm_fts_parser (' "The Microsoft business analysis" OR " MS revenue" ', 1033, 77,0)

This query selects all the data returned from the function **sys.dm_fts_parser**:

- Keyword
- group_Id
- phrase_Id
- occurrence
- special_term
- display_term
- expansion_type
- source_term

The output is the following:

keyword	group_id	phrase_id	occurrence	special_term	displ
0x007400680065	1	0	1	Noise Word	The
0x006D006900630072006F0073006F00660074	1	0	2	Exact Match	Micros
0x0062007500730069006E006500730073	1	0	3	Exact Match	busin

0x0061006E0061006C0079007300690073	1	0	4	Exact Match	analy
0x006D006900630072006F0073006F00660074	2	0	1	Exact Match	Micro
0x0072006500760065006E00750065	2	0	2	Exact Match	reven

Following is a description of each column returned by **sys.dm_fts_parser**:

· **expansion_type**

· This column contains information about the nature of the expansion that a given term experienced. The possible values are the following:

§ 0 =Single word case

§ 2 =Inflectional expansion

§ 4 =Thesaurus expansion/replacement

For example:

FORMSOF (FREETEXT, run)
(assuming that run has 'jog' as expansion in the thesaurus file for the language being used)

This generates the following output:

§ *run* with expansion_type=0

§ *runs* with expansion_type=2

§ *running* with expansion_type=2

§ *ran* with expansion_type=2

§ *jog* with expansion_type=4

· **special_term**

· This column contains information about the characteristics of the term being issued by the word breaker. The possible values are the following:

§ Exact match

§ Noise word

§ End of sentence

§ End of paragraph

§ End of chapter

· **display_term**

· This column contains the displayable form of the keyword. This facilitates the task of identifying which are the parsed terms from the input. The displayed term might not be perfectly identical to the original due to denormalization limitations. However, it should be precise enough so that you can understand its value.

· **occurrence**

· This column contains the integer describing the order of the given term in the parsing result. For example:

"SQL Server SKU" outputs in English:

· *SQL* with occurrence=1

· *Server* with occurrence=2

· *SKU* with occurrence=3

· **source_term**

· This column contains the term or phrase from which a given term was generated/parsed. For example:

'Server AND "Data Base"' outputs in English:

§ *Server* with *source_term = Server*

§ *Data* with *source_term = Data Base*

§ *Base* with *source_term = Data Base*

· **group_id**

· This column contains an integer that is useful for differentiating the logical group from which a given term was generated. For example:

Server AND DB OR FORMSOF (THESAURUS, DB)"' outputs in English:

· *Server* with *group_id=1*

§ *DB* with *group_id=2*

§ *DB* with *group_id=3*

· **phrase_id**

· Sometimes, with compound words (such as 'multi-million'), alternative forms are issued by the word breakers. These alternatives forms (phrases) must sometimes be differentiated. This column contains an integer useful for differentiating in such cases. For example, in English, 'multi-Million' returns:

· *Multi* with *phrase_id=1*

· *Million* with *phrase_id=1*

· *MultiMillion* with *phrase_id=2*

This function can be very powerful for debugging purposes. Following are some benefits of this feature:

· **Understand how a given word breaker treats a given input**

Our customers sometimes wonder why a given query does not return the expected results. Usually this has to do with the way the word breaker parses or breaks the data. In most cases, its behavior is correct. By using **sys.dm_fts_parser**, you can check the exact result that the word breaker will pass to the full-text index. In addition, you can see which terms are stopwords (noise words), and therefore not searched in the full-text index. Whether a term is a stopword for a given language is determined by the stoplist ID declared in the function (if any).

Note as well the accent sensitivity flag, which enables you to see how the word breaker will parse the input according to its accent sensitivity information.

· **Understand how the stemmer works**

The stemmer is very flexible; it allows any kind of input that follows the CONTAINS syntax. For example, you can issue the following query:

FORMSOF (INFLECTIONAL, <*query_term*>)

The results will show how the word breaker and the stemmer parsed the original term plus its stemming forms. You know exactly which terms are being passed against the full-text index. Example: run à runs, ran, running

· **Understand how the thesaurus expands or replaces either part of or the entire input**

In the following command:

FORMSOF (THESAURUS, <*query_term*>)

the results show how the word breaker and thesaurus interacted. You can see the expansions or replacements that took place from the thesaurus and understand which final query is being issued against the full-text index.

Note that if you issue the following:

FORMSOF (FREETEXT, <*query_term*>)

the inflectional and thesaurus capabilities will take place automatically due the FREETEXT invocation. (The same thing happens when you use FREETEXT through a Transact-SQL predicate.)

There are many more ways to use **sys.dm_fts_parser** to help you understand and troubleshoot full-text search query issues.

To force the Unicode representation of the query input, add a capital N in front of the query term, for example: (N '在庫'). This is useful when you want to parse Unicode languages when using a computer with a non-Unicode code page (English for instance).

For a complete reference on the syntax and data types of these dynamic management functions, see the "sys.dm_fts_parser" topic in SQL Server 2008 Books Online.

A New Word Breaker Family

We are glad to announce that SQL Server 2008 integrated full-text search includes a complete new family of word breakers and stemmers. These word breakers and stemmers are significantly better than those in former SQL Server releases. The Microsoft Natural Language Group (NLG) now owns the research, implementation, and support of these linguistic components.

Word breakers are language-aware components responsible for parsing textual data and breaking it into single words (terms). These terms are then indexed by the Full-Text Engine. Stemmers are language-aware components that determine the inflectional forms of some terms for a given language. For instance, the English stemmer can generate the terms 'ran' and 'running' from the term 'run.'

Following are the main enhancements that this new family of word breakers and stemmers provide.

Robustness

Testing proves that the new word breakers are robust in high-pressure query environments.

Security

All the new word breakers are enabled by default in SQL Server 2008. The security improvements in these word breakers are well known and proven by extensive testing.

Quality

NLG redesigned some of the existing word breakers to solve issues that could arise in former word breakers (such as German and Japanese). Testing proves the better semantic quality of these word breakers. This increases the recall accuracy for full-text search users.

Coverage

SQL Server 2008 includes out of the box and enabled by default NLG word breakers for the following languages.

Language
French
German
Japanese
Spanish
Bengali
Bulgarian
Catalan
Croatian
Gujarati

Hindi

Icelandic

Indonesian

Canada

Latvian

Lithuanian

Malay

Malayalam

Marathi

Neutral

Punjabi

Romanian

Serbian Cyrillic

Serbian Latin

Slovak

Slovenian

Tamil

Telugu

Ukrainian

Urdu

Vietnamese

Arabic

Norwegian

Portuguese Brazilian

Russian

Dutch

Portuguese

Hebrew

Italian

Swedish

In addition to these new word breakers, SQL Server 2008 will ship with the following, which are identical to those

shipped in SQL Server 2005. This means that when you upgrade, if you use only the following languages, you do not need to repopulate your full-text catalogs to guarantee perfect recall. The word breaker behavior for these languages in both versions is identical.

English

English UK

Simplified Chinese

Traditional Chinese

Chinese (Hong Kong)

Chinese (Macau)

Chinese (Singapore)

Thai

Korean

Note: Besides these word breakers, other languages are available after the user enables them in SQL Server 2008. These are disabled by default because they are owned by third parties who have not yet provided the level of testing, security, and robustness that is required for them to be enabled by default.

Danish

Polish

Turkish

Performance Improvements

This section focuses on indexing and querying, two important aspects of performance.

Indexing

The indexing performance observed in SQL Server 2008 full-text search is equal to or slightly superior to that of SQL Server 2005. We observed that 64-bit architectures experience a performance gain in comparison with 32-bit. Most of the performance gains introduced by the new full-text architecture are for query performance, as described later in this white paper.

Note that SQL Server 2008 full-text search is more aggressive in memory usage and may use more I/O; 64-bit architectures are better than 32-bit in doing crawl tasks in general.

Sometimes crawl can be too aggressive in committing memory and this could cause sqlserver/system to have fewer resources than needed. It is highly recommended that you do the following:

· Limit the memory used by the crawl (full population) by setting **max server memory** to an appropriate value to prevent the crawl task from taking all available memory.

· Increase the page file size if applicable.

Indexing performance is generally superior to SQL Server 2005. However, in certain situations there may be some performance degradation as follows:

· Crawl on large documents (~>8KB). In non 64-bit platforms, this might be slower in some scenarios.

· **varchar** type data. This is because SQL Server 2005 does a better job with the **varchar** type than with a

text type, whereas in the new full-text architecture there is no difference in theory between **varchar** data and **text** data.

· Noninteger keys, especially GUID columns used as a full-text key

· Too many unique keywords. It has been shown in random document crawl modes that performance can be slower than SQL Server 2005 (when **uniquekeycount** > 80 million).

For details and troubleshooting recommendations, see Indexing Performance later in this white paper.

Querying

Due the complete integration between the new Full-Text Engine and the SQL Server query processor, we are in a position to improve overall query performance in most scenarios. This integration brings several advantages that previous full-text search solutions could not leverage. To understand the performance gains of SQL Server 2008 full-text search, it is important to review the main performance problem existing in previous full-text search implementations: the mixed query scenario.

Mixed Query Scenarios

A *mixed query* is a query that presents predicates from both the relational and full-text worlds:

· Relational: WHERE *x* (=,BETWEEN,<,>, etc…) *y*

· Full-Text: CONTAINS (description, 'Carbs')

The following query retrieves matches from a database that stores information about candidates who mention 'SQL Server' in their resume and belong to a specific hiring division.

```
SELECT candidate_ID FROM Candidates

WHERE CONTAINS (Resume,"SQL Server")

AND candidate_division=534A
```

Assume that in this example, the table named Candidates contains a large number of candidates (rows) containing the phrase 'SQL Server', but only a few of this belong to the division 534A. You would expect that the query plan for this query would first find the candidates belonging to that specific division, and then use those results to find who has SQL Server experience. This would be far more effective than using the reverse order.

Full-text search in SQL Server 2000 and 2005 cannot operate this way because it is not integrated with the SQL Server query processor. When this type of query ran, the query processor had no statistics from full-text search to determine the optimal approach, nor did it have a way to push predicates/IDs to the external Full-Text Engine. Thus, previous releases of full-text search had no other solution than to send the full-text predicate to the Full-Text Engine (outside), and then wait for the results to come in order to filter them by a more selective predicate. This is very inefficient.

This problem is solved in SQL Server 2008 with the full integration of the SQL Server query processor and the Full-Text Engine. The query runs efficiently, retrieving the candidates belonging to a specific division, and then pushing these against the full-text index to find which of these contain 'SQL Server'. This assumes, of course, that the query optimizer has the correct statistics to detect this selectivity fact.

Following are the possible scenarios:

	FT ~ relational	**FT >> relational**	**FT << relational**
Selectivity			
Cardinality estimation			
Small Cardinality	Nested loop join	Nested loop join	Nested loop join
Large Cardinality	Merge join or hash join	Merge join or hash join	Merge join or hash join (new in SQL Server 2008) Nested loop +predicate Push-down to full-text index

· FT ~ relational

CONTAINS (movieDescription,"Action") AND country=USA

· FT >> relational

CONTAINS (movieDescription,"World War II") AND country=USA

· FT << relational

CONTAINS (movieDescription,"Action") AND director="Coppola"

· FT <<<< relational

CONTAINS (movieDescription,"Action") AND Category=2345A

For example:

SELECT candidate_ID FROM Candidates

WHERE CONTAINS(resume,'"SQL Server"')

AND candidate_division=534A

The Candidates table has 1 million candidates having 'SQL Server' experience but only 50,000 of them are associated with the divisionID=534A. Assume that SQL Server 2005 takes 20 seconds to return the 1 million rows that match the CONTAINS predicate from the full-text index. (This is an imaginary number, by no means representative of real performance.)

· *X*: % of rows returned by the full-text index matching the relational condition= **5%**

In general, we have seen in our testing that:

integrated full-text search query time ~= (SQL 2005 query time) x (X/100)

where *X* is the number of rows returned by the full-text index matching the relational condition.

In our example, the integrated full-text search query time would be:

20 seconds x (5/100) = **1 second**

(Assuming that the query plan calculates the relational predicate first, and then pushes the small subset of matching rows to the full-text index.)

Integrated full-text search query response time is approximately 100/*X* times faster than SQL Server 2005 full-text index time in these ideal query scenarios, where *X* is the percentage of rows returned by the full-text index matching the relational condition. Note that in scenarios where the selectivity is not so different, you will not see this significant gain because full-text search in previous versions already chose the correct operation path: the full-text predicate side first, no matter what.

Compilation time of full-text queries

In SQL Server 2000 and 2005, full-text predicates were treated as remote scan operators, causing a given compilation time and query plan creation. A similar query plan for full-text queries was always created; it may or may not be ideal, but it was simple.

In SQL Server 2008 where the Full-Text Engine is integrated with QP/QO, the full-text predicates translates to TVFs, sometimes causing different operators at compilation time and therefore, different query plans in some cases (we now have statistics on full-text predicates). This allows SQL Server to create better query plans for your full-text queries (mixed-query types), but these plans can be more complex, increasing compilation to the same as that in SQL Server 2005 for the same query. Sometimes creating a better query plan does increase the compilation time although it should improve execution time afterwards. Please note this in case some full-text queries will have significantly higher compilation times than in previous SQL Server releases.

Important Note: In SQL Server 2000 and 2005, a valid workaround for some scenarios (where the relational cardinality was far lower than the full-text one) was to push the content of the relational columns into the full-text index, so the final query need only work at the full-text index level using full-text ANDs. This workaround is not valid in SQL Server 2008 and you should not use it. Do *not* push/duplicate relational column content into full-text columns. Instead, you can now combine relational predicates and full-text predicates naturally, without the need to duplicate/push relational data into the full-text index to speed up this type of query. Otherwise your application will not take advantage of the iFTS/QP integration and statistics at query time, making the full-text query more complex than necessary.

For example. the following is correct in SQL Server 2008:

CONTAINS (movieDescription,”Action”) AND Category=***2345A***

The following is incorrect in SQL Server 2008:

CONTAINS (movieDescription,”Action AND Category%%***2345A”)***

For details, see Querying in the Performance Improvements section.

Query Parallelism

In previous releases of SQL Server, full-text queries could not produce parallel plans. With integrated full-text search, they now can. The best-case scenario for this gain is when the number of rows to be returned by the Full-Text index side is large and there is no selective relational predicate. In this case, full-text search can generate parallel hash joins, which improve execution time by a factor of almost X times (where *X* = number of processors) compared to previous releases of SQL Server full-text search.

In the case of a parallel hash join, the full-text table-valued function (TVF) is executed serially but the hash build and look up are parallelized. This means that if the total time to run a query in SQL Server 2005 full-text search is:

Time for TVF execution (A) + Time for join (B)

In SQL Server 2008 full-text search architecture, it is:

Time for TVF execution (A) + Time for join (B)***/X***

Because join costs are typically much higher than TVF execution costs, the improvement is close to *X*.

Integer Full-Text Keys

In previous full-text search releases, an internal table named docidmap mapped the internal full-text index IDs to the base table full-text key. For performance reasons, we use integers to map each indexed row to the full-text index, even when the row or document ID is another type such as string, text, computed column, and so on.

This meant that every full-text query needed an extra internal JOIN against this internal table in order to map the retrieved rows from the full-text index against the real rows in the base use table.

In SQL Server 2008 full-text search, we improved this design and no longer need the internal docidmap table in cases where the base table full-text key is already an integer type (**bigint** and **smallint** are considered to be integer types in this case). The docidmap table is not created, avoiding an extra join at query time. This results in a significant performance gain in many scenarios.

We strongly recommend using an integer type as the full-text key of the base table to be indexed so that you can experience this gain.

Note that **numeric()** types are not mapped as integer internally, causing the automatic creation of the docidmap table.

Breaking Changes

Following is a list of breaking changes in relation to SQL Server 2005 introduced by SQL Server 2008 full-text search architecture changes.

Feature	Scenario	SQL Server 2005	SQL Server 2008
CONTAINSTABLE with user-defined types (UDTs)	The full-text key is a SQL Server user-defined type, for example: MyType = char(1)	Returns key of the type assigned to the user-defined type. In the example, this would be char(1).	Returns key of the user-defined type. In the example, this would be MyType.
top_n_by_rank parameter (of the CONTAINSTABLE and FREETEXTTABLE Transact-SQL statements)	top_n_by_rank queries using zero as the parameter	Fails with an error message stating that you must use a value greater than zero.	Succeeds, returning zero rows.
CONTAINSTABLE and ItemCount	Deletes rows from base table before it pushes changes to MSSearch.	CONTAINSTABLE returns ghost record. ItemCount is not changed.	CONTAINSTABLE does not return any ghost records.
ItemCount	Table contains null documents or type columns.	In addition to indexed documents, documents that are null or that have null types are counted in the ItemCount value.	Only indexed documents are counted in the ItemCount value.
Catalog ItemCount	BLOB column with a NULL extension.	The row is counted in ItemCount of catalog	Is not counted in ItemCount of catalog.
UniqueKeyCount	Querying a unique key count from a catalog, for example, two tables (table1 and table2) each with three words: word1, word2, and word3.	UniqueKeyCount = 9. The following summarizes how this value is attained: table1 = 3 EOF for full-text index of table1 = 1	For each table, UniqueKeyCount is the number of distinct keywords + 1 (0xFF). This does not treat same words in multiple docs as a new unique key. For a catalog,

		table2 = 3 EOF for full-text index of table2 = 1 full-text catalog = 1	UniqueKeyCount is the sum of UniqueKeyCount of each of the tables under the catalog. Identical words from different tables are treated as unique keys. In this case the unique key count is 8.
sp_fulltext_pendingchanges when updating key column	Update the full-text key column on one row of a two-row table, run **sp_fulltext _pendingchanges**	Both rows appear.	Only one row appears.
Inline functions	Inline functions with a full-text operator	Returns an error message.	Return the relevant rows.
sp_fulltext_database	Enable or disable full-text search by using **sp_fulltext_database**.	No results are returned for full-text queries. If full-text is disabled for the database, full-text operations are not allowed.	Returns results to full-text queries; full-text operations allowed, even if full-text is disabled for the database.
Locale-specific stop words	Queries in locale-specific variants of a parent language, such as Belgian French and Canadian French.	Queries in locale-specific variants are processed by the components (word breakers, stemmers, and stop words) of their parent language. For example, the French (France) components are used to parse French (Belgium).	You must add stop words explicitly for each locale identifier (LCID). For example, you would need to specify an LCID for Belgium, Canada, and France.
Thesaurus stemming process	Using thesaurus and Inflectional forms (stemming).	A thesaurus word is automatically stemmed after its expansion.	If you want the stemmed form in the expansion, explicitly add the stemmed form.
Full-text catalog path and filegroup	Working with full-text catalogs.	Each full-text catalog has a physical path and belongs to a filegroup. It is treated as a database file.	A full-text catalog is a virtual object and does not belong to any filegroup. A full-text catalog is a logical concept that refers to a group of full-text indexes. **Note:** SQL Server 2005 Transact-SQL DDL statements

			that specify full-text catalogs work correctly.
sys.fulltext_catalogs*cf1489ff-4819-41fa-a62a-4ed797a16207*	Using the **path**, **data_space_id**, and **file_id** columns of this catalog view.	These columns return a specific value.	These columns return NULL because the full-text catalog is no longer located in the file system.
sys.sysfulltextcatalogs	Using the **path** column of this deprecated system table.	Returns the file system path of the full-text catalog.	Returns NULL because the full-text catalog is no longer located in the file system.
sp_help_fulltext_catalogs1b94f280-e095-423f-88bc-988c9349d44c **sp_help_fulltext_catalogs_cursor**	Using the **path** column of these deprecated stored procedures.	Returns the file system path of the full-text catalog.	Returns NULL because the full-text catalog is no longer located in the file system.
sp_help_fulltext_catalog_components	Using **sp_help_fulltext_catalog_components**	Returns a list of all components (filters, word breakers, and protocol handlers) used for all full-text catalogs in the current database.	Returns empty rows.
DATABASEPROPERTY and DATABASEPROPERTYEX	Using the IsFullTextEnabled property.	The IsFullTextEnabled setting indicates whether full-text search is enabled in a given database.	The value of this property has no effect. User databases are always enabled for full-text search.

Implementing Full-Text Search: Best Practices

It is very important that you follow best practices so that you can tune your application to satisfy your needs. This section covers the main best practices we would like our customers to have in mind. These apply to SQL Server 2008.

Manageability and Security

Following are the main best practices related to manageability (such as backup and restore) and security.

· SQL Server 2008 full-text search allows for loading only trusted (signed) components. We recommend keeping the default setting—load only trusted code—to keep your server safe. Change the **load_os_resources** and **verify_signature** properties if you need to load third-party filters and word breakers

· Perform a Full or Differential backup before and after making major schema changes or starting a population of your full-text indexes and catalogs.

· If you have manageability or performance problems, it can be a good practice to create a new filegroup in a different fast volume (disk) from the filegroup containing the database files. Allocate the full-text index files on this new separate filegroup to allow parallel access to full-text indexes and the base tables. This can also help manageability operations such as backup and restore, as now you can select which filegroups to include in the operation, enabling you to manage the base table and/or full-text indexes independently.

· Monitor crawl (population) logs. Crawl logs are located in the \LOG directory. They have been improved in SQL Server 2008 to provide lots of information for cases where the source of a problem might be an index.

Indexing and Full Population Performance

Full-text indexing performance is influenced by hardware resources, such as memory, disk speed, CPU speed, and machine architecture. The main cause for reduced full-text indexing performance is hardware resource limits.

· If CPU usage by the filter daemon host process (fdhost.exe) or the SQL Server process is close to 100 percent, the CPU is the bottleneck.

· If the average disk-waiting queue length is more than two times the number of disk heads, there is a bottleneck on the disk. The primary workaround is to create full-text catalogs that are separate from the SQL Server database files and logs. Put the logs, database files, and full-text catalogs on separate disks. Buying faster disks and using RAID can also help improve indexing performance.

· If there is a shortage of physical memory (3-GB limit), memory might be the bottleneck. Physical memory limitations are possible on all systems, and on 32-bit systems, virtual memory pressure can slow down full-text indexing.

Note In SQL Server 2008 and later versions, the Full-Text Engine can use AWE memory because the Full-Text Engine is part of the SQL Server process.

If the system has no hardware bottlenecks, the indexing performance of full-text search depends mostly on the following:

· How long it takes SQL Server to create full-text batches.

· How quickly the filter daemon can consume those batches.

Note Unlike full population, incremental, manual, and auto change tracking population are not designed to maximize hardware resources to achieve faster speed. Therefore, these tuning suggestions may not enhance performance for these other population methods.

Recommended Steps Prior to a Full Population

1. To utilize all processors/cores to the maximum, set **sp_configure** *'max full-text crawl ranges'* to the number of CPUs on the system

2. Make sure that the base table has a clustered index. Use an integer data type for the first column of the clustered index. Avoid using GUIDs in the first column of the clustered index. A multi-range population on a clustered index can produce the highest population speed. We recommend that the column serving as the full-text key be an integer data type.

3. Update the statistics of the base table by using the UPDATE STATISTICS statement. More importantly, update the statistics on the clustered index or the full-text key for a full population. This helps a multi-range population generate good partitions on the table.

4. Build a secondary index on a **timestamp** column if you want to improve the performance of incremental population.

Troubleshooting Full Population Performance Issues

It is recommended that you troubleshoot in the following order if the performance of full populations is not satisfactory. First, check physical memory usage, and then check for low CPU consumption.

Physical memory usage

If the amount of available physical memory during the population is zero, it is possible that the SQL Server buffer pool is consuming most of the physical memory on the system. This could cause one of the following two problems, which you can diagnose by looking at the full-text crawl logs:

· The page file size is not large enough. This could cause fdhost or the SQL Server process (sqlservr.exe) to run out of memory. To troubleshoot, look at the full-text crawl logs. If filter daemon restarts often or if error code 8007008 is encountered, fdhost or sqlservr.exe is running out of memory. These failures can be eliminated by increasing the page file size on the system.

· If there are no memory-related failures in the crawl logs, it is likely that performance is not optimal because of excessive paging. To solve this, set the **max server memory** of the SQL Server buffer pool appropriately.

For x86 platform (AWE disabled)

F = Number of crawl ranges * 50 MB

M = min(T, 2000) - F -500

For x86 platform (AWE enabled)

F = Number of crawl ranges * 50 MB

M = T – F - 500

For x64 or ia64 platform

F = Number of crawl ranges * 10 * ism_size

M = T – F - 500

F is an estimate of memory needed by fdhost.exe (in MB).

T is the total physical memory available on the system (in MB).

M is the optimal max server memory setting in **sp_configure**.

Platform	Estimating fdhost.exe memory needs in MB—'F'	Formula for calculating **max server memory**—'M' (for essential information, see footnotes [1] and [2])
x86 with AWE disabled	F = Number of crawl ranges * 50MB	M = min(T, 2000) – F – 500
x86 with AWE enabled	F = Number of crawl ranges * 50MB	M = T – F – 500
x64 or IA64	F = Number of crawl ranges * 10 * 8 [3]	M = T – F – 500

Compute *ism_size* by executing **sp_fulltext_service** *'ism_size'*. The default value is ism_size=8 for x64 or ia64.

For example, if the computer is an amd64 computer with 8 GB of RAM and four dual core processors, F=8*10*8=640MB assuming the ism size is 8 MB and number of crawl ranges is e. Then:

M = 8192-640-500=7052MB

Low CPU consumption

If CPU consumption is low (under 30 percent), the environment is not optimal for full population. Several factors can cause this behavior:

· The page wait time may be high. To find out if a page wait time is high, execute the following Transact-SQL statement:

Execute SELECT TOP 10 * FROM sys.dm_os_wait_stats ORDER BY wait_time_ms DESC;

The following table describes the wait types of interest:

Wait Type	**Description**	**Possible Resolution**
PAGEIO_LATCH_SH (_EX or _UP)	This could indicate an I/O bottleneck; typically the average disk-queue length is high.	Moving the full-text index to a different filegroup on a different disk could help reduce the I/O bottleneck.

PAGELATCH_EX (or _UP)	This could indicate contention among threads that are trying to write to the same file.	Adding files to the filegroup on which the full-text index resides could help alleviate contention.

· A good practice is to turn on trace flag 7603 to investigate problems related to low CPU usage. If there are many messages of the following form in the error log:

L"IFTS: CFTFullCrawl::ExecuteCrawl fragments limit reached... re-queueing\n

it means that batches are being produced at a much faster rate than being consumed. This could again mean a slow I/O system and would typically be associated with an average disk queue write length that is high. Moving the full-text index to a different file group on faster disk could help reduce the I/O bottleneck.

· There may be inefficiencies in scanning the base table. During full population, we scan the base table to produce batches. This could be inefficient in the following scenarios:

· If the base table has a high percentage of out-of-row columns being full-text indexed, it is possible that scanning the base table to produce batches is the bottleneck. In this case, moving the smaller data in-row using **varchar(max)** or **nvarchar(max)** might help.

· If the base table is very fragmented, scanning can be inefficient. Reorganizing or rebuilding the clustered index would reduce the fragmentation.

Use **sys.dm_db_partition_stats** and **sys.dm_db_index_physical_stats** to compute out-of-row data and index fragmentation.

Query Performance

Full-text indexing performance is influenced by hardware resources, such as memory, disk speed, CPU speed, and computer architecture. The performance of full-text queries is highly dependent on many factors such as your hardware, size of the full-text catalog, size of the result set, query complexity, language used, thesaurus content, and so on.

This section covers troubleshooting and tuning your query load performance.

Increasing Full-Text Query Performance

The recommendations will help increase full-text query performance:

· Defragment the index of the base table by using ALTER INDEX REORGANIZE.

· Reorganize the full-text catalog by using ALTER FULLTEXT CATALOG REORGANIZE. Make sure that you do this before performance testing because running this statement causes a master merge of the full-text indexes in that catalog.

· Restrict your choice of full-text key columns to a small column. Although a 900-byte column is supported, we recommend using a smaller key column in a full-text index. **int** and **bigint** provide the best performance.

· Combine multiple CONTAINS predicates into one CONTAINS predicate. In SQL Server you can specify a list of columns in the CONTAINS query.

· If you only require full-text key or rank information, use CONTAINSTABLE or FREETEXTTABLE instead of CONTAINS or FREETEXT, respectively.

· To limit results and increase performance, use the *top_n_by_rank* option with FREETEXTTABLE and CONTAINSTABLE. Use this option when you are not interested in all possible hits.

· Check the full-text query plan to make sure that the appropriate join plan is chosen. Use a join hint or query hint if you have to. If a parameter is used in the full-text query, the first-time value of the parameter determines the query plan. You can use the OPTIMIZE FOR query hint to force the query to compile with the value you want. This helps achieve a deterministic query plan and better performance.

· Use an integer type for the full-text key when possible. If this is not possible, try using as small a column as possible. When the full-text key is an integer type, there is no need to maintain an additional internal table (docidmap table) to map full-text index IDs (integers) with base table full-text keys (nonintegers). This saves an additional internal JOIN in full-text queries. We highly recommend therefore using a column of type integer as the full-text key. Note that when you upgrade, importing the full-text catalogs does not remove the temporary docidmap table from the previous SQL Server implementation. If you imported the full-text catalogs, you might want to schedule a rebuild of these to avoid this additional step at query time. The docidmap table is not needed if you have an integer full-text key.

· Monitor full-text index fragmentation and reorganize if necessary. A single logical full-text index might be composed of several small physical chunks. This is a typical case when many updates are performed against the

base table that you have full-text indexed. It is recommended that you manually trigger this merge as doing so can improve the query performance and cardinality/ranking estimations. In addition, memory usage when fragmentation is high is larger as well.

You can check the number of fragments in the index by checking the contents of **sys.fulltext_index_fragments** where the table ID matches the object ID of the table. If you have only one fragment with a status of 4 (closed), you have a fully merged index; otherwise, you have multiple fragments and may benefit from a merge.

Check the **MergeStatus** property from the FULLTEXTCATALOGPROPERTY scalar function to monitor the merge status.

ALTER FULLTEXT CATALOG *catalog_name* REORGANIZE

Note that this can be an expensive operation and should not be performed constantly.

· Do not push/duplicate relational columns content into the full-text columns. This *mixed*-query workaround is valid for SQL Server 2000 and 2005 in some scenarios but is not ideal in SQL Server 2008. Instead, combine relational predicates and full-text predicates naturally; there is no need to duplicate/push relational data into the full-text index to speed up this type of queries. For more details, see the Note in "Performance Improvements."

· Logical operators specified in CONTAINSTABLE (AND, OR) can be implemented either as SQL joins or inside the full-text execution STVF. Typically, queries with only one type of logical operator are implemented purely by full-text execution, whereas queries that mix logical operators also possess SQL joins. Implementation of a logical operator inside the full-text execution STVF uses some special index properties that make it much faster than SQL joins. For this reason, we recommend that, where possible, you frame queries by using only a single type of logical operator. This also improves the compilation cost of your queries.

For example, sometimes users issue queries like the following:

….WHERE CONTAINS(*,'car OR vehicle OR automobile') AND CONTAINS (*,'Mercedes OR BMW OR Toyota OR…) etc…'

It is simpler to use the thesaurus to add synonyms of 'car' and car brands than it is to exploit the query massively to add all the combinations. The correct query is:

….WHERE CONTAINS(*,'FORMSOF(THESAURUS,car) AND 'FORMSOF(THESAURUS,'Mercedes'))

This is assuming, of course, that the correct thesaurus has been modified to include the synonyms.

· For applications that contain selective-relation predications, queries that use selective relational predicates and unselective full-text predicates might perform best when they are written to use the query optimizer. This allows the query optimizer to decide whether it can exploit predicate or range push down to produce an effective query plan. This approach is simpler and often more efficient than indexing relational data as full-text data.

Autochange tracking

When the autochange tracking option is turned on (set to ON by default at full-text index creation time) and the rate of updates, inserts, and deletes (DMLs) is very high, this can cause query performance regression in some cases.

When autochange tracking is turned on, every single change to the base table is propagated to an internal table that is then constantly parsed by the Full-Text Engine to keep the full-text index up to date. When the rate of DMLs (insert, updates, or deletes) is very high (~>150/seconds), this might cause significant blocking that can cause full-text query performance to suffer. If you detect performance degradation due a high level of changes in the base table, we highly recommend the following:

· Switch to manual change tracking when possible. Scheduling manual change tracking from time to time (as it can be tolerated by your application and needs) frees up resources and internal blocking so queries can run faster. If your latency requirements are not strict, we recommend not doing this too often.

· Periodically issue a full-text master merge (REORGANIZE). Changes to the base table cause the creation of new full-text index fragments. When the number of fragments is very high, query performance and ranking results can suffer. Therefore, we recommend that you schedule a REORGANIZE of the full-text catalog containing your full-text index.

· Note that this operation is costly, so you should schedule master merges infrequently and only when observed that query performance is suffering.

· If you must use autochange tracking while your DML and query loads are high, activate trace flag 7646, which will alleviate the internal blocking between your DML statements and queries.

Because this trace flag alleviates blocking, it is possible that a given document containing the term being queried

might not be retrieved at query time until the changes related to that document are committed into the Full-Text Index. This is a very small time window but it is possible that this could occur when this trace flag is activated. Subsequent queries will retrieve this document after the change has been propagated.

Query plan issues associated with full-text queries

A typical full-text query plan looks like that in the following figure.

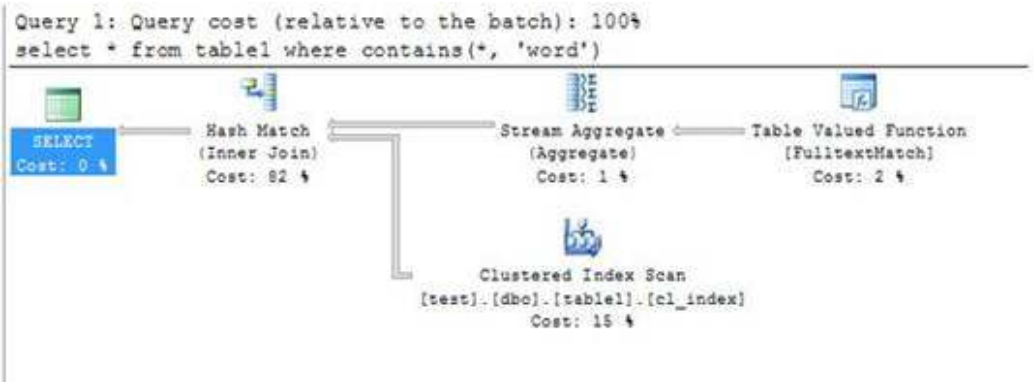

Figure 3

· The most important thing is that the correct join type is picked for full-text query. Cardinality estimation on the FulltextMatch STVF is very important for the right plan. So the first thing to check is the FulltextMatch cardinality estimation. This is the estimated number of hits in the index for the full-text search string. For example, in the query in Figure 3 this should be close to the number of documents containing the term 'word'. In most cases it should be very accurate but if the estimate was off by a long way, you could generate bad plans. The estimation for single terms is normally very good, but estimating multiple terms such as phrases or AND queries is more complex since it is not possible to know what the intersection of terms in the index will be based on the frequency of the terms in the index. If the cardinality estimation is good, a bad plan probably is caused by the query optimizer cost model. The only way to fix the plan issue is to use a query hint to force a certain kind of join or OPTIMIZE FOR.

When there is relational and full-text mixed query, if the relational predicate is very selective, it is optimal to have a predicate push-down plan. Continuing with the same query example, add an additional relational predicate and the following query:

select * from table1 where contains(*, 'word') and column1 > 4900.

If only 100 rows qualify for the column1 > 4900 predicate, it is more efficient to have the following query plan with a nest-loop join and a TVF predicate push down. So if the relational predicate is selective, make sure the predicate push-down plan is chosen. To verify that the predicate push down occurred, see if the full-text table-valued function has one estimated return row.

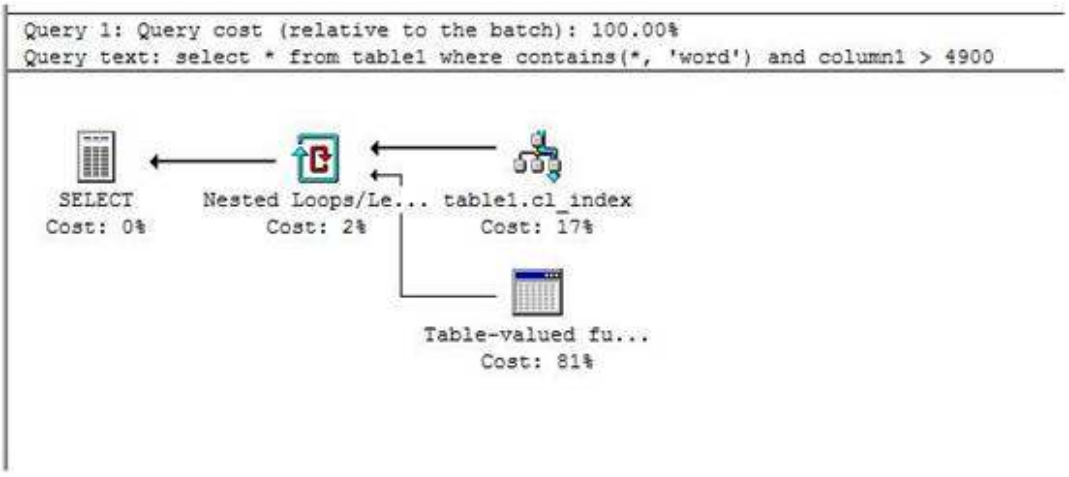

Figure 4

· Prefix queries that return a large result set can affect performance. Most full-text query strings result in either single terms such as CONTAINS(*,'run'), or a few terms such as CONTAINS(*,'FORMSOF(INFLECTIONAL,'run'), but in some cases the query string can result in many more matches in the index. For example, the query CONTAINS(*,'"a*"') matches any word in the index beginning with "a". This might match hundreds or thousands of keywords. This query would use a lot of memory and consume a large share of system resources. To protect against this, a fallback mechanism automatically switches to a slower processing mode when a threshold of 20 MB of required memory for a query is reached. If you see a sudden drop in prefix query performance, you may have reached this threshold.

There are several approaches you can take to resolve this issue:

· Check for index fragmentation (multiple fragments cause larger numbers of rowsets to be opened and more memory to be consumed). This is another reason why it is a good idea to keep your index fragments under control.

· Validate whether you really need to run prefix queries. Applications often add *a** at the end of the query string trying to help the overall recall but often it is not beneficial and hurts performance.

If you must run the query and have plenty of memory, use trace flag 7662 to disable this behavior.

· In stored procedures or any procedure that is precompiled, the queries inside the stored procedure are precompiled and a query plan fixed. This has the following consequences in the new full-text search architecture:

· The best possible query plan for the given search term that is passed might not be used. In SQL Server 2008 full-text search we have the ability to alter the plan that is generated based on a cardinality estimation of the search term used. If the query plan is fixed (as it is in a parameterized query inside a stored procedure), this step does not take place. Therefore, the compiled plan always serves this query, even if this plan is not ideal for a given search term. For example, if a given term is very common in the full-text index, it might be better for the optimizer to operate another part of the query first and later push the row IDs to the full-text index. If this query plan is not the default, it will not be used for the query in the stored procedure.

· Query recompilations might occur because, depending on the search term, the current STVF used in the compiled plan is not valid anymore. Most often, this applies to ranking calculations in CONTAINSTABLE. For example, if you search for 'run AND walk', this translates to a specific ranking pattern that might cause the STVF serving the full-text part to be recompiled if the plan (STVF) was created by using a ranking based on a single-term search such as 'run'. The same happens if you search for 'run OR walk', or ISABOUT (weighted terms…). When this occurs, the current STVF and query plan must be recompiled to serve the new ranking needs.

Application Development Aspects

You must make some high-level decisions before you deploy your full-text application. Use the following recommendations to deploy your application as efficiently as possible in SQL Server full-text search.

· Understand the differences between the CONTAINS and FREETEXT families of functions. Balance between precise control over query semantics versus ease of use. The CONTAINS ranking is slightly faster to calculate.

· We strongly recommend that you use the *top_n_by_rank* parameter in CONTAINSTABLE and FREETEXTTABLE to reduce result set sizes. It works well for most scenarios, above all when your application applies paging of the results. Note that you may need to re-issue the query when there are additional filtering predicates.

· Consider whether to scale up or scale out. Scaling up generally works well when you are using up to hundreds millions of documents. The need to scale out can vary greatly depending on document size, query workload, typical query result set size, and so on. Beyond that, we recommend that you design your application for scale out with horizontal partitioning. Also consider scaling out if query workload is very high (such as when the number of concurrent users proves to be too many for a single server).

The most important factors when determining whether to scale out or scale up are:

· **Hardware**. A powerful multi-core computer with a large amount of memory and fast I/O makes a difference in performance. 64-bit architectures also offer important advantages.

· **Types of queries.** Selective queries can work very well on a single server, no matter the size of the full-text index. When each query returns many rows, query throughput might start to suffer due to the individual cost of queries.

· **Number of queries versus the number of DMLs**. A very high rate of queries and updates/inserts per second might require scale measures. The type of queries that you use is an important consideration as well.

· **Type of documents and LCIDs.** Document type and LCID type can affect indexing time performance, and also relates to the final size of the full-text catalog after textual data extraction. iFilters and iWordBreakers operate in specific ways and the final size of the full-text index depends entirely on these.

· Choose the correct change tracking method, as well as DML/query loads. For more details, see Query Performance.

· Use the full-text catalog size as a starting point for estimating memory requirements for the fdhost.exe cache at query time. It should be able to cache at least one third of the catalog.

· A 64-bit architecture is ideal because of the extra memory it provides.

· In some cases, the base table is too large to fit in the buffer pool. To reduce I/O on the base table:

· Turn random I/O on the base table into sequential I/O.

· Insert full-text results (such as when you use CONTAINSTABLE) into a temp table clustered in FTKEY. Then join them with the base table, or replace references to CONTAINSTABLE with a subquery of the form of:

(SELECT TOP (1000) * FROM CONTAINSTABLE(*cols, your query, top_n_by_rank*) ORDER BY FTKEY)

· In both cases, *top_n_by_rank* is your best friend.

· If you use multilanguage support, take the following into consideration.

There are several things to consider when you choose the column language when creating a full-text index. These relate to how your text is tokenized and then indexed by the Full-Text Engine in SQL Server. A word breaker tokenizes the text being indexed on word boundaries. These word boundaries, in the English language, are typically white space or some form of punctuation. In other languages, such as German, words or characters may be combined. Specify a column-level language that represents the language that you expect to be stored in rows of that column.

If you are unsure, a general best bet is to use the word breaker for the most complex language within the same family of the language that will be stored in the column. For example, if you expect to store English, Spanish, and German content in a column, you would use the German word breaker because German is the most complex of these languages. These are all Western languages with common word breaking patterns. The English word breaker might not process German content as well because of the compound words in the German content. For non-Western languages this rule does not necessarily work.

If languages differ dramatically (such as Spanish and Japanese), you might store them in separate columns and apply the correct language to each one. At query time, if you do not know the language, you might need to issue the original query against both columns (using the correct language) to make sure you find the specific row/document.

If none of the above are solutions for your case, try the following:

· When the indexed content is of binary type (such as a Microsoft Word document), the iFilter responsible for processing the text content before sending it to the word breaker might honor specific language tags in the binary file. When this is the case, at indexing time the iFilter invokes the correct word breaker for a specific document or section of a document specified in a particular language. All you need to do in this case is to verify after indexing that the multilanguage content was indexed correctly. Filters for Word, HTML, and XML documents honor language specification attributes in document content:

Word – language settings

HTML - <meta name="MS.locale"…>

XML – xml:lang attribute

· When your content is plain text, you can convert it to the XML data type and add specific language tags to indicate the language corresponding to that specific document or document section. Note that for this to work, before you index you must know the language that will be used.

· Use the query-time language specification available since SQL Server 2005. That way you can reissue the same query for each language that might be present in the columns searched.

In summary, always try using the most complex language expected to be present in your column, but always from the same family and alphabet (do not specify Japanese for content that might be in Spanish). If languages differ dramatically, you might want to store them in separate columns and apply the right language to each one. You will need to query both columns later with the right language to make sure you find back the specific row/document. When none of the above is possible, you might want to convert your text data to XML type and add language tags that indicate which word breaker to use for each row or row section. Remember that some binary types already contain such tags (.doc, .xls, etc.).

Supporting Aspects

Following are some of the mechanisms available to troubleshoot and support your full-text application. Most of these are unchanged from SQL Server 2005.

User error messages

For all user-issued DDLs, DMLs, and queries, full-text related errors are always returned. Full-text related errors are in the range of 76xx, 99xx, and 300xx.

Error log file

There are several background and/or asynchronous tasks involved in full-text indexing. Errors encountered by these tasks are written to the sqlserver error log file. These include indexing errors, fdhost termination, failure to

initialize the filter daemon process, and so on. Some of these messages are also sent to the event log.

Full-text crawl log

Full-text crawl related information, such as crawl starts, end, and abort events, and any errors hit during crawl are logged in the full-text crawl log file. The crawl log file is named as sql<*database id*>+<*catalog id*>.log. This file is in the LOG directory for the SQL Server instance, just as in SQL Server 2005.

Following is an example of the information that would be logged for a corrupted or mis-typed document in a table at key value 5:

2007-09-26 16:48:10.03 spid20s Informational: Full-text Auto population initialized for table or indexed view '[repro].[dbo].[t1]' (table or indexed view ID '549576996', database ID '9'). Population sub-tasks: 1.

2007-09-26 16:48:12.05 spid20s Error '0x8004170c: The document format is not recognized by the filter.' occurred during full-text index population for table or indexed view '[repro].[dbo].[t1]' (table or indexed view ID '549576996', database ID '9'), full-text key value '5'. Attempt will be made to reindex it

Event log

Serious errors or events that administrators might need to act on go to the Windows event log. The failure might be a read-only file group, terminated fdhost process, and so on. This uses the same mechanism as in SQL Server 2005.

Crash dumps

The Full-Text Engine takes advantage of sqldumper integration so that severe exceptions inside the server produce dumps. We also integrated fdhost.exe with sqldumper so that any severe exceptions in filtering, word breaking, and so on generate crash dumps in the same way the server does. The dump files for fdhost issues are named sqldmpr*xxx*.mdmp. Note this is different from the dump file names generated for sqlserver.exe dumps, which are sqldmp*xxx*.mdmp.

Because all dumps from either the server or fdhost are integrated with sqldumper, they can also be sent via Watson if you select that option when configuring the installation.

Troubleshooting hangs

Because of the new integrated full-text search architecture, SQL Server no longer suffers the same hang problems as in SQL Server 2005 full-text search when calls got stuck in the MSSearch engine. That problem could be drastic if it occurred in critical database startup code. However, integrated full-text search has different scenarios when the full-text indexing process is stuck and is not making any progress. The problem could be due to missing events, a background task did not properly handle an error properly, and similar events. To diagnose this type of problem, check for blocking background tasks (FT GATHERER tasks) by using **sys.sysprocesses**, **sys.dm_os_wait_tasks**, and so on. To recover, pause and resume the population.

At query time, if the word breaking infrastructure is stuck, user queries will time out and receive the correct error message. Following queries will keep suffering timeouts if the communication channel between sqlserver and fdhost gets stuck.

Query execution uses the full-text filter daemon process (fdhost.exe) for word breaking. The filter daemon process hosts word breaker and stemmer DLLs that are considered to be not trusted and unreliable by SQL Server.

Note: This is a design perspective and not indicative of the reliability or trustworthiness of any one specific word breaker. Full-text queries cannot execute if word breaking services from the filter daemon are not available.

This kind of problem can occur if:

· The word breaker corresponding to the query language is configured incorrectly. (For example, there might be bad registry settings.)

· The word breaker malfunctions for a specific query string.

· The word breaker returns too much data for a specific query string. We treat this as a potential buffer overrun attack and shut down the filter daemon process.

· The filter daemon process is not configured correctly. The most common reasons are password expiry or a domain policy that prevents the filter daemon account from logging on.

· You have a very high query workload on the server although this is the most unlikely root cause.

To recover from query time failure, restarting fdhost.exe is a solution. Use the stored procedure (**Sp_fulltext_service 'restart_all_fdhosts**) to recycle fdhost.exe.

Troubleshooting an unexpected outcome (The query runs but results or execution are not as expected.)

Query results that are incorrect could happen for many reasons. The most common causes are due to a word breaking behavior change, mismatched full-text query language, noise word processing, or an indexing failure. Tools are available to diagnose the problem. Following are some common diagnostic steps:

· Use **sys.dm_fts_parser()** to see how the query string gets broken for the query language. The query engine does an exactly match on the normalized form against a full-text index. For people familiar with debugging Yukon full-text issues, this is analogous to lrtest.exe but it is much more useful because it is all done from within a sql session without the need to run executables on a customer computer. For details, see the specification for this feature. You can use this tool to see exactly how the query string is broken and if any of the terms in the resulting query are noise words or other special terms. It is extremely useful for debugging query issues. This is intended for debugging only.

· Use the new supportability functions to access the contents of the full-text index. These are analogous to the tool CIDUMP.exe that was used in SQL Server 2005 to dump the index but are much more useful as they are inside the sqlservr.exe process. The two functions are **sys.dm_fts_index_keywords()** and **sys.dm_fts_index_keywords_by_document()**.

If the query term does not show up in the index for the document, there are several reason why this might occur:

· **The document was not indexed**. In this case, you will see errors for the document in the crawl logs, which should point you to the source of the problem. The most common reasons for a document to not be indexed are:

· Filter not installed for the document type

· Corrupt document

· Filter or word breaker bug caused an error in the document that crashed or hung the fdhost process

· **The document was indexed but the term you are searching for was not found.** This might be because the term is a noise word; this would be in the output of **sys.dm_fulltext_parser()**. Or, the term was not output by the filter that was used to filter the document. This could be by design or a filter bug. To validate filter behavior, use lrtest.exe as in SQL Server 2005.

Conclusion

In previous SQL Server releases, the full-text search feature made its first steps into the Search world while maintaining a double ownership between the SQL Server and MSSearch teams. We believe that SQL Server 2008 is our main and best milestone from for full-text search in SQL Server.

Full-text search in SQL Server 2008 is the outcome of a massive effort: re-engineering full-text search so that it is fully integrated with SQL Server. This accomplishment puts us the position to offer what our customers have asked us to provide in the area of full-text search.

We are very proud of what full-text search in SQL Server 2008 has come to be and we are greatly excited to start implementing our next set of enhancements on top of this new, fully owned, and SQL Server fully integrated full-text search architecture.

For more information:

SQL Server Web site: http://www.microsoft.com/sqlserver/2008/en/us/default.aspx [http://www.microsoft.com/sqlserver/2008/en/us/default.aspx]

SQL Server TechCenter: http://technet.microsoft.com/en-us/sqlserver/default.aspx [http://technet.microsoft.com/en-us/sqlserver/default.aspx]

SQL Server DevCenter: http://msdn2.microsoft.com/en-us/sqlserver/default.aspx [http://msdn2.microsoft.com/en-us/sqlserver/default.aspx]

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

· Are you rating it high due to having good examples, excellent screenshots, clear writing, or another reason?

· Are you rating it low due to poor examples, fuzzy screenshots, unclear writing?

This feedback will help us improve the quality of white papers we release. Send feedback.

Notes

[1] 500 MB is an estimate of the memory required by other processes in the system. If the system is doing additional work, increase this value accordingly.

[2] If multiple full populations are in progress, calculate the fdhost.exe memory requirements of each separately as F1, F2, and so forth. Then calculate M as T **–** sigma**(**Fi**)**.

[3] 8 represents the typical size of inbound shared memory (ISM).

Tags:

Community Content