**Pro Full-Text Search in SQL Server 2008**

**Copyright © 2009 by Michael Coles and Hilary Cotter**

ISBN-13 (pbk): 978-1-4302-1594-3

ISBN-13 (electronic): 978-1-4302-1595-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jonathan Gennick
Technical Reviewer: Steve Jones
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Denise Santoro Lincoln
Copy Editor: Benjamin Berg
Associate Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositor/Artist: Octal Publishing, Inc.
Proofreader: Patrick Vincent
Indexer: Broccoli Information Management
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at http://www.apress.com/info/bulksales.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at http://www.apress.com.

CHAPTER 1

# SQL Server Full-Text Search

*. . . but I still haven't found what I'm looking for.*

—Bono Vox, U2

Full-text search encompasses techniques for searching text-based data and documents. This is an increasingly important function of modern databases. SQL Server has had full-text search capability built into it since SQL Server 7.0. SQL Server 2008 integrated full-text search (iFTS) represents a significant improvement in full-text search functionality, a new level of full-text search integration into the database engine over prior releases. In this chapter, we'll discuss full-text search theory and then give a high-level overview of SQL Server 2008 iFTS functionality and architecture.

## Welcome to Full-Text Search

Full-text search is designed to allow you to perform linguistic (language-based) searches against text and documents stored in your databases. With options such as word and phrase-based searches, language features, the ability to index documents in their native formats (for example, Office documents and PDFs stored in the database can be indexed), inflectional and thesaurus generational terms, ranking, and elimination of noise words, full-text search provides a powerful set of tools for searching your data. Full-text search functionality is an increasingly important function in modern databases. There are many reasons for this increase in popularity, including the following:

- Databases are increasingly being used as document repositories. In SQL Server 2000 and prior, storage and manipulation of large object (LOB) data (textual data and documents larger than 8,000 bytes) was difficult to say the least, leading to many interesting (and often complicated) alternatives for storing and manipulating LOB data outside the database while storing metadata within the database. With the release of SQL Server 2005, storage and manipulation of LOB text and documents was improved significantly. SQL Server 2008 provides additional performance enhancements for LOB data, making storage of all types of documents in the database much more palatable. We'll discuss these improvements in later chapters in this book.

- Many databases are *public facing*. In the not too distant past, computers were only used by a handful of technical professionals: computer scientists, engineers, and academics. Today, almost everyone owns a computer, and businesses, always conscious of the bottom dollar, have taken advantage of this fact to save money by providing self-service options to customers. As an example, instead of going to a brick-and-mortar store to make a purchase, you can shop online; instead of calling customer service, you check your orders online; instead of calling your broker to place a stock trade, you can research it and then make the trade online. Search functionality in public-facing databases is a key technology that makes online self-service work.

- Storage is cheap. Even as hard drive prices have dropped, the storage requirements of the average user have ballooned. It's not uncommon to find a half terabyte (or more) of storage on the average user's personal computer. According to the Enterprise Strategy Group Inc., worldwide total private storage capacity will reach 27,000 petabytes (27 billion gigabytes) of storage by 2010. Documents are born digitally, live digitally, and die digitally, many times never having a paper existence, or at most a short transient hard-copy life.

- New document types are constantly introduced, and there are increasing requirements to store documents in their native format. XML and formats based on or derived from XML have changed the way we store documents. XML-based documents include XHTML and Office Open XML (OOXML) documents. Businesses are increasingly abandoning paper in the normal course of transactions. Businesses send electronic documents such as purchase orders, invoices, contracts, and ship notices back and forth. Regulatory and legal requirements often necessitate storing exact copies of the business documents when no hard copies exist. For example, a pharmaceutical company assembles medications for drug trials. This involves sending purchase orders, change orders, requisition orders, and other business documents back and forth. The format for many of these documents is XML, and the documents are frequently stored in their native formats in the database. While all of this documentation has to be stored and archived, users need the ability to search for specific documents pertaining to certain transactions, vendors, and so on, quickly and easily. Full-text search provides this capability.

- Researching and analyzing documents and textual data requires data to be stored in a database with full-text search capabilities. Business analysts have two main issues to deal with during the course of research and analysis for business projects:

  - Incomplete or dirty data can cripple business analysis projects, resulting in inaccurate analyses and less than optimal decision making.

  - Too much data can result in information overload, causing "analysis paralysis," slowing business projects to a crawl.

- Full-text search helps by allowing analysts to perform contextual searches that allow relevant data to reveal itself to business users. Full-text search also serves as a solid foundation for more advanced analysis techniques, such as extending classic data mining to text mining.

- Developers want a single standardized interface for searching documents and textual data stored in their databases. Prior to the advent of full-text search in the database, it was not uncommon for developers to come up with a wide variety of inventive and sometimes kludgy methods of searching documents and textual data. These custom-built search routines achieved varying degrees of success. SQL Server full-text search was designed to meet developer demand for a standard toolset to search documents and textual data stored in any SQL Server database.

SQL Server iFTS represents the next generation of SQL Server-based full-text search. The iFTS functionality in SQL Server provides significant advantages over other alternatives, such as the `LIKE` predicate with wild cards or custom-built solutions. The tasks you can perform with iFTS include the following:

- You can perform linguistic searches of textual data and documents. A *linguistic search* is a word- or phrase-based search that accounts for various language-specific settings, such as the source language of the data being searched, inflectional word forms like verb conjugations, and diacritic mark handling, among others. Unlike the `LIKE` predicate, when used with wild cards, full-text search is optimized to take full advantage of an efficient specialized indexing structure to obtain results.

- You can automate removal of extraneous and unimportant words (stopwords) from your search criteria. Words that don't lend themselves well to search and don't add value to search results, such as *and*, *an*, and *the*, are automatically stripped from full-text indexes and ignored during full-text searches. The system predefines lists of stopwords (stoplists) in dozens of languages for you. Doing this on your own would require a significant amount of custom coding and knowledge of foreign languages.

- You can apply weight values to your search terms to indicate that some words or phrases should be treated as more important than others in the same full-text search query. This allows you to normalize your results or change the ranking values of your results to indicate that those matching certain terms are more relevant than others.

- You can rank full-text search results to allow your users to choose those documents that are most relevant to their search criteria. Again, it's not necessarily a trivial task to create custom code that ranks search results obtained through custom search algorithms.

- You can index and search an extremely wide array of document types with iFTS. SQL Server full-text search understands how to tokenize and extract text and properties from dozens of different document types, including word-processing documents, spreadsheets, ZIP files, image files, electronic documents, and more. SQL Server iFTS also provides an extensible model that allows you to create custom components to handle any document type in any language you choose. As examples, there are third-party components readily available for additional file formats such as AutoCAD drawings, PDF files, PostScript files, and more.

It's a good bet that a large amount of the data stored by your organization is unstructured—word processing documents, spreadsheets, presentations, electronic documents, and so on. Over the years, many companies have created lucrative business models based on managing unstructured content, including storing, searching, and retrieving this type of content. Some rely on SQL Server's native full-text search capabilities to help provide the back-end functionality for their products. The good news is that you can use this same functionality in your own applications.

The advantage of allowing efficient searches of unstructured content is that your users can create documents and content using the tools they know and love—Word, Acrobat, Excel—and you can manage and share the content they generate from a centralized repository on an enterprise-class database management system (DBMS).

# History of SQL Server FTS

Full-text search has been a part of SQL Server since version 7.0. The initial design of SQL Server full-text search provided for reuse of Microsoft Indexing Service components. Indexing Service is Microsoft's core product for indexing and searching files and documents in the file system. The idea was that FTS could easily reuse systemwide components such as word breakers, stemmers, and filters. This legacy can be seen in FTS's dependence on components that implement Indexing Service's programming interfaces. For instance, in SQL Server, document-specific filters are tied to filename extensions.

Though powerful for its day, the initial implementations of FTS in SQL Server 7.0 and 2000 proved to have certain limitations, including the following:

- The DBMS itself made storing, manipulating, searching, and retrieving large object data particularly difficult.
- The fact that only systemwide shared components could be used for FTS indexing caused issues with component version control. This made side-by-side implementations with different component versions difficult.
- Because FTS was implemented as a completely separate service from the SQL Server query engine, efficiency and scalability were definite issues. As a matter of fact, SQL Server 7.0 FTS was at one point considered as an option for the eBay search engine; however, it was determined that it wasn't scalable enough for the job at that time.
- The fact that SQL Server had to store indexes, noise word lists, and other data outside of the database itself made even the most mundane administration tasks (such as backups and restores) tricky at best.
- Finally, prior versions of FTS provided no transparency into the process. Troubleshooting essentially involved a sometimes complicated guess-and-fail approach.

The new version of SQL Server integrated FTS provides much greater integration with the SQL query engine. SQL Server 2008 large object data storage, manipulation, and retrieval has been greatly simplified with the new large object `max` data types (`varchar(max)`, `varbinary(max)`). Although you can still use systemwide FTS components, iFTS allows you to use instance-specific installations of FTS components to more easily create side-by-side implementations. FTS efficiency and scalability has been greatly improved by implementing the FTS query engine directly within the SQL Server service instead of as a separate service. Administration has been improved by storing most FTS data within the database instead of in the file system. Noise word lists (now stopword lists) and the full-text catalogs and indexes themselves are now stored directly in the database, easing the burden placed on administrators. In addition, the newest release of FTS provides several dynamic management views and functions to provide insight into the FTS process. This makes troubleshooting issues a much simpler exercise.

## MORE ON TEXT-BASED SEARCHING

Text-based searching is not exclusively the domain of SQL Server iFTS. There are many common applications and systems that implement text-based searching algorithms to retrieve relevant documents and data. Consider MS Outlook—users commonly store documents in their Outlook Personal Storage Table (PST) files or in their MS Exchange folders. Frequently, Outlook users will email documents to themselves, adding relevant phrases to the email (*mushroom duxelles recipe* or *notes from accounting meeting*, for example) to make searching easier later. What we see here is users storing all sorts of data (email messages, images, MS Office documents, PDF files, and so on) somewhere on the network in a database, tagging it with information that will help them to find relevant documents later, and sometimes categorizing documents by putting them in subfolders. The key to this model is being able to find the data once it's been stored. Users may rely on MS Outlook Search, Windows Desktop Search, or a third-party search product (such as Google Desktop) to find relevant documents in the future.

Searching the Web requires the use of text-based search algorithms as well. Search engines such as Google go out and scrape tens of millions of web pages, indexing their textual content and attributes (like META tags) for efficient retrieval by users. These text-based search algorithms are often proprietary in nature and custom-built by the search provider, but the concepts are similar to those utilized by other full-text search products such as SQL Server iFTS.

Microsoft has being going back and forth for nearly two decades over the idea of hosting the entire file system in a SQL Server database or keeping it in the existing file system database structure (such as NTFS [New Technology File System]). Microsoft Exchange is an example of an application with its own file system (called *ESE*—pronounced "easy") that's able to store data in rectangular (table-like) structures and nonrectangular data (any file format which contains more properties than a simple file name, size, path, creation date, and so forth). In short, it can store anything that shows up when you view any documents using Windows Explorer. Microsoft has been trying to decide whether to port ESE to SQL Server. What's clear is that SQL Server is extensible enough to hold a file system such as NTFS or Exchange, and in the future might house these two file systems, allowing SQL FTS to index content for even more applications.

Microsoft has been working on other search technologies since the days of Windows NT 3.5. Many of their concepts essentially extend the Windows NT File System (NTFS) to include schemas. In a schema-based system, all document types stored in the file system would have an associated schema detailing the properties and metadata associated with the files. An MS Word document would have its own schema, while an Adobe PDF file would also have its own schema. Some of the technologies that Microsoft has worked on over the years promise to host the file system in a database. These technologies include OFS (Object File System), RFS (Relational File System, originally intended to ship with SQL 2000), and WinFS (Windows Future Storage, but also less frequently called Windows File System). All of these technologies hold great promise in the search space, but so far none have been delivered in Microsoft's flagship OS yet.

# Goals of Search

As we mentioned, the primary function of full-text search is to optimize linguistic searches of unstructured content. This section is designed to get you thinking about search in general. We'll present some of the common problems faced by search engineers (or as they're more formally known, *information retrieval scientists*), some of the theory behind search engines, and some of the search algorithms used by Microsoft. The goals of search engines are (in order of importance):

1. To return a list of documents, or a list of links to documents, that match a given search phrase. The results returned are commonly referred to as a list of *hits* or *search results*.
2. To control the inputs and provide users with feedback as to the accuracy of their search. Normally this feedback takes the form of a ratio of the total number of hits out of the number of documents indexed. Another more subtle measure is how long the search engine churns away before returning a response. As Michael Berry points out in his book *Understanding Search Engines- Mathematical Models and Text Retrieval* (SIAM, ISBN 0-89871-437-0), an instantaneous response of "No documents matched your query" leaves the user wondering if the search engine did any searching at all.
3. To allow the users to refine the search, possibly to search within the results retrieved from the first search.
4. To present the users with a search interface that's intuitive and easy to navigate.
5. To provide users a measure of confidence to indicate that their search was both exhaustive and complete.
6. To provide snippets of document text from the search results (or document abstracts), allowing users to quickly determine whether the documents in the search results are relevant to their needs.

The overall goal of search is to maximize user experience in all domains. You must give your users accurate results as quickly as possible. This can be accomplished by not only giving users what they're looking for, but delivering it quickly and accurately, and by providing options to make searches as flexible as possible.

On one hand, you don't want to overwhelm them with search results, forcing them to wade through tens of thousands of results to find the handful of relevant documents they really need. On the other hand, you do want to present them with a flexible search interface so they can control their searching without sacrificing user experience.

There are many factors that affect your search solution: hardware, layout and design, search engine, bandwidth, competitors, and so on. You can control most of these to some extent, and with luck you can minimize their impact. But what about your users? How do you cater to them?

Search architects planning a search solution must consider their interface (or search page) and their users. No matter how sophisticated or powerful your search server, there may be environmental factors that can limit the success of your search solution. Fortunately, most of these factors are within your control. The following problems can make your users unhappy:

- Sometimes your users don't know what they're looking for and are making best guesses, hoping to get the right answers. In other words, unsophisticated searchers rely on a hit-or-miss approach, blind luck, or serendipity. You can help your users by offering training in corporate environments, providing online help, and instituting other methods of educating them. Good search engineers will institute some form of logging to determine what their users are searching for, create their own "best bets" pages, and tag content with keywords to help users find relevant content efficiently. User search requirements and results from the log can be further analyzed by research and development to improve search results, or those results can be directed to management as a guide in focusing development dollars on hot areas of interest.

- Sometimes users make spelling mistakes in their search phrases. There are several ingenious solutions for dealing with this. Google and the Amazon.com search engine run a spell check and make suggestions for other search terms when the number of hits is relatively low. In the case of Amazon.com, the search engine can recommend best-selling products that you might be interested in that are relevant to your search.

- Sometimes users are presented with results in an overwhelming format. This can quickly lead frustrated users to simply give up on continuing to search with your application. A cluttered interface (such as a poorly designed web page) can overwhelm even the most advanced user. A well-designed search page can overcome this. Take a tip from the most popular search engine in the world—Google provides a minimalist main page with lots of white space.

- Sometimes the user finds it too difficult to navigate a search interface and gives up. Again, a well designed web site with intuitive navigation helps alleviate this.

- Sometimes the user is searching for a topic and using incorrect terminology. This can be addressed on SQL Server, to some degree, through the use of inflectional forms and thesaurus searches.

In this chapter, we're going to consider the search site `Google.com`. We'll contrast Google against some of Microsoft's search sites, and against `Microsoft.com`. We'll be surveying search solutions from across the spectrum of possible configurations.

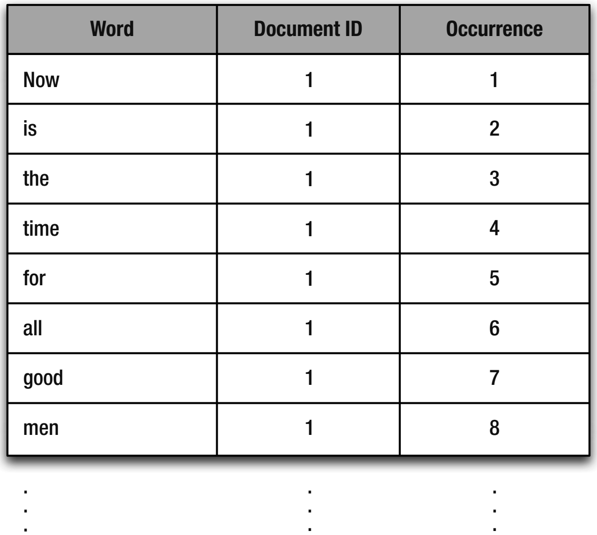
## GOOGLE

Google, started as a research project at Stanford University in California, is currently the world's most popular search engine. For years, `http://google.stanford.edu` used to redirect to `http://www.google.com`; it now redirects to their Google mini search appliance (`http://www.stanford.edu/services/websearch/Google/`). Google is powered by tens of thousands of Linux machines—termed *bricks*—that index pages, perform searches, and serve up cached pages. The Google ranking algorithm differs from most search algorithms in that it relies on inbound page links to rank pages and determine result relevance. For instance, if your web site is the world's ultimate resource for diabetes information, the odds are high that many other web sites would have links pointing to your site This in turn causes your site to be ranked higher when users search for diabetes-related topics. Sites that don't have as many links to them for the word *diabetes* would be ranked lower.

## Mechanics of Search

Modern search solutions such as iFTS rely on precompiled indexes of words that were previously extracted from searchable content. If you're storing word processing documents, for instance, the precompiled index will contain all of the words in the documents and references back to the source documents themselves. The index produced is somewhat similar to an index at the back of most books. Imagine having to search a book page by page for a topic you're interested in. Having all key words in an index returns hits substantially faster than looking through every document you're storing to find the user's search phrase.

SQL Server uses an inverted index structure to store full-text index data. The inverted index structure is built by breaking searchable content into word-length tokens (a process known as *tokenizing*) and storing each word with relevant metadata in the index. An inverted index for a document containing the phrase *Now is the time for all good men to come to the aid of the party* would be similar to Figure 1-1.

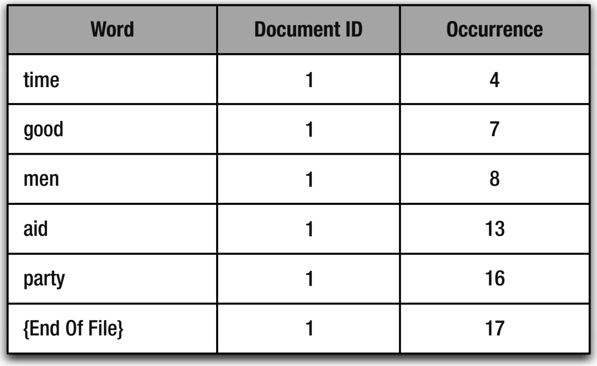


| Word | Document ID | Occurrence |
|---|---|---|
| Now | 1 | 1 |
| is | 1 | 2 |
| the | 1 | 3 |
| time | 1 | 4 |
| for | 1 | 5 |
| all | 1 | 6 |
| good | 1 | 7 |
| men | 1 | 8 |
| ⋮ | ⋮ | ⋮ |

**Figure 1-1.** *Inverted index of sample phrase (partial)*

The key fields in the inverted index include the word being indexed, a reference back to the source document where the word is found, and an occurrence indicator, which gives a relative position for each word. SQL Server actually eliminates commonly used stopwords such as *the*, *and*, and *of* from the index, making it substantially smaller. With system-defined stopwords removed, the inverted index for the previously given sample phrase looks more like Figure 1-2.

■ **Note** The sample inverted index fragments shown are simplified to include only key information. The actual inverted index structure SQL Server uses contains additional fields not shown.



| Word | Document ID | Occurrence |
|---|---|---|
| time | 1 | 4 |
| good | 1 | 7 |
| men | 1 | 8 |
| aid | 1 | 13 |
| party | 1 | 16 |
| {End Of File} | 1 | 17 |

**Figure 1-2.** *Inverted index with stopwords removed*

Whenever you perform a full-text search in SQL Server, the full-text query engine tokenizes your input string and consults the inverted index to locate relevant documents. We'll discuss indexing in detail in Chapter 2 and full-text search queries in Chapter 3.

# iFTS Architecture

The iFTS architecture consists of several full-text search components working in cooperation with the SQL Server query engine to perform efficient linguistic searches. We've highlighted some of the more important components involved in iFTS in the simplified diagram shown in Figure 1-3.

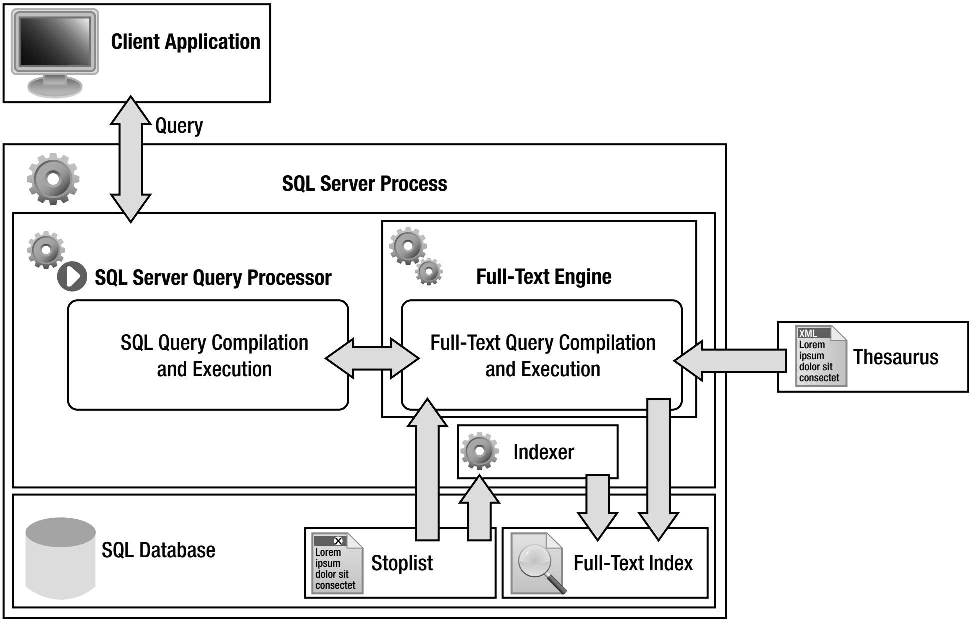

Client Application
Query
SQL Server Process
SQL Server Query Processor
SQL Query Compilation and Execution
Full-Text Engine
Full-Text Query Compilation and Execution
XML
Lorem ipsum dolor sit consectet
Thesaurus
Indexer
SQL Database
Lorem ipsum dolor sit consectet
Stoplist
Full-Text Index

**Figure 1-3.** *iFTS architecture (simplified)*

The components we've highlighted in Figure 1-3 include the following:

- *Client application*: The client application composes full-text queries and submits them to the SQL Server query processor. It's the responsibility of the client application to ensure that full-text queries conform to the proper syntax. We'll cover full-text query syntax in detail in Chapter 3.
- *SQL Server process*: The SQL Server process contains both the SQL Server query processor, which compiles and executes SQL queries, and the full-text engine, which compiles and executes full-text queries. This tight integration of the SQL Server and full-text query processors in SQL Server 2008 is a significant improvement over prior versions of SQL Server full-text search, allowing SQL Server to generate far more efficient query plans than was previously possible.
- *SQL Server query processor*: The SQL Server query processor consists of several subcomponents that are responsible for validating SQL queries, compiling queries, generating query plans, and executing queries in the database.
- *Full-text query processor*: When the SQL Server query processor receives a full-text query request, it passes the request along to the full-text query processor. It's the responsibility of the full-text query processor to parse and validate the query request, consult the full-text index to fulfill the request, and work with the SQL Server query processor to return the necessary results.
- *Indexer*: The indexer works in conjunction with other components to retrieve streams of textual data from documents, tokenize the content, and populate the full-text indexes. Some of the components with which the indexer works (not shown in the diagram) include the gatherer, protocol handler, filters, and word breakers. We'll discuss these components in greater detail in Chapter 10.
- *Full-text index*: The full-text index is an inverted index structure associated with a given table. The indexer populates the full-text index and the full-text query processor consults the index to fulfill search requests. Unlike prior versions of SQL Server the full-text index in SQL Server 2008 is stored in the database instead of the file system. We will discuss setup, configuration, and population of full-text indexes in detail in Chapter 2.
- *Stoplist*: The stoplist is simply a list of stopwords, or words that are considered useless for the purposes of full-text search. The indexer consults the stoplist during the indexing and querying process in order to eliminate stopwords from the index and search phrase. Unlike prior versions of SQL Server, which stored their equivalent of stoplists (noise word lists) in the file system, SQL Server 2008 stores stopword lists in the database. We'll talk about stoplists in greater detail in Chapter 7.
- *Thesaurus*: The thesaurus is an XML file (stored in the file system) that defines full-text query word replacements and expansions. Replacements and expansions allow you to expand a search to include additional words or completely replace certain words at query time. As an example, you could use the thesaurus to expand a query for the word *run* to also include the words *jog* and *sprint*, or you could replace the word *maroon* with the word *red*. Thesauruses are language-specific, and the query processor consults the thesaurus at query time to perform expansions and replacements. We'll detail the mechanics and usage of thesauruses in Chapter 8.

**Note** Though the XML thesaurus files are currently stored as files in the file system, the iFTS team is considering the best way to incorporate the thesaurus files directly into the database, in much the same way that the stoplists and full-text indexes have been integrated.

## Indexing Process

The full-text indexing process is based on the index population, or *crawl* process. The crawl can be initiated automatically, based on a schedule, or manually via T-SQL statements. When a crawl is started, an iFTS component known as the *protocol handler* connects to the data source (tables you're full-text indexing) and begins streaming data from the searchable content. The protocol handler provides the means for iFTS to communicate with the SQL storage engine. Another component, the *filter daemon host*, is a service that's external to the SQL Server service. This service controls and manages content-type-specific filters, which in turn invoke language-specific word breakers that tokenize the stream of content provided by the protocol handler.

The indexing process consults stoplists to eliminate stopwords from the tokenized content, normalizes the words (for case and accent sensitivity), and adds the indexable words to inverted index fragments. The last step of the indexing process is the *master merge*, which combines all of the index fragments into a single master full-text index. The indexing process in general and the master merge in particular can be resource- and I/O-intensive. Despite the intensity of the process, the indexing process doesn't block queries from occurring. Querying a full-text index during the indexing process, however, can result in partial and incomplete results being returned.

## Query Process

The full-text query process uses the same language-specific word breakers that the indexer uses in the indexing process; however, it uses several additional components to fulfill query requests. The query processor accepts a full-text query predicate, which it tokenizes using word breakers. During the tokenization process, the query processor creates *generational forms*, or alternate forms of words, as follows:

- It uses *stemmers*, components that return language-based alternative word forms, to generate inflectional word forms. These inflectional word forms include verb conjugations and plural noun forms for search terms that require them. Stemmers help to maximize precision and recall, which we'll discuss later in this chapter. For instance, the English verb *eat* is stemmed to return the verb forms *eating*, *eaten*, *ate*, and *eats* in addition to the root form *eat*.

- It invokes language-specific *thesauruses* to perform thesaurus replacements and expansions when required. The thesaurus files contain user-defined rules that allow you to replace search words with other words or expand searches to automatically include additional words. You might create a rule that replaces the word *maroon* with the word *red*, for instance; or you might create a rule that automatically expands a search for *maroon* to also include *red*, *brick*, *ruby*, and *scarlet*.

■ **Tip** Stemmer components are encapsulated in the word breaker DLL files, but are separate components (and implement a separate function) from the word breakers themselves. Different language rules are applied at index time by the word breakers than by the stemmers at query time. Many of the stemmers and word breakers have been completely rewritten for SQL 2008, which makes a full population necessary for many full-text indexes upgraded from SQL 2005. We'll discuss full-text index population in detail in Chapter 2.

After creating generational forms of words, the query processor provides input to the SQL Server query processor to help determine the most efficient query plan through which to retrieve the required results. The full-text query processor consults the full-text index to locate documents that qualify based on the search criteria, ranks the results, and works with the SQL Server query processor to return relevant results back to the user.

The new tighter integration between the full-text query processor and the SQL Server query processor (both are now hosted together within the SQL Server process) provides the ability to perform full-text searches that are more highly optimized than in previous versions of SQL Server. As an example, in SQL Server 2005 a full-text search predicate that returned one million matching documents had to return the full one-million-row result set to the SQL Server query processor. At that point, SQL Server could apply additional predicates to narrow down results as necessary. In SQL Server 2008, the search process has been optimized so that SQL Server can shortcut the process, limiting the total results that need to be returned by iFTS without all the overhead of passing around large result sets full of unnecessary data between separate full-text engine and SQL Server services.

## Search Quality

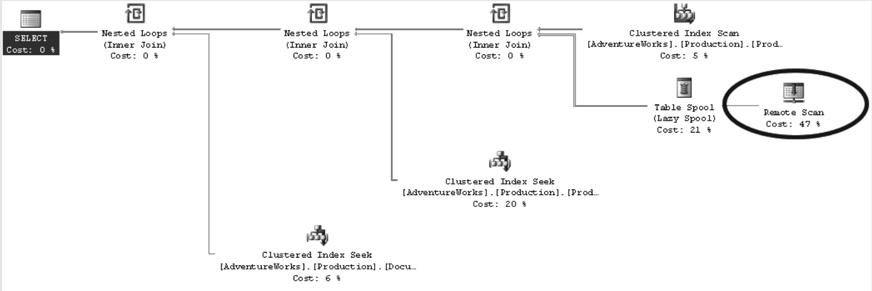
For most intranet sites and other internal search solutions, the search phrases that will hit your search servers will be a small fraction or subset of the total number of words in the English language (or any other language for that matter). If you started searching for medical terms or philosophical terms on the Microsoft web site, for instance, you wouldn't expect to get many hits (although we do get hits for *existentialist*, *Plato*, and *anarchist*, we aren't sure how much significance, if any, we can apply to this).

Microsoft's web site deals primarily with technical information—it can be considered a subset of the total content that's indexed by Google. Amazon indexes book titles, book descriptions, and other product descriptions. They would cover a much larger range of subjects than the Microsoft web site, but wouldn't get into the level of detail that the Microsoft site does, as Amazon primarily indexes the publisher's blurb on the book or other sales-related literature for their products.

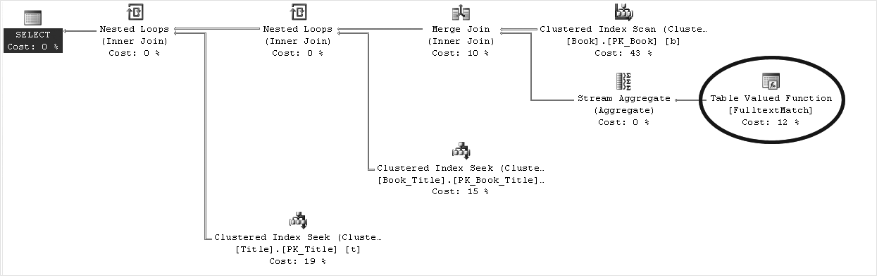
As you can see, Google probably contains many entries in its index for each word in the English language. In fact, for many words or phrases, Google has millions of entries; for example, the word *Internet* currently returns over 2.6 billion hits as of Fall 2008. Search engines with a relatively small volume of content to index, or that are specialized in nature, have fewer entries for each word and many more words having no entries.

### BENEFITS OF INTEGRATION

As we mentioned previously, the new level of integration that SQL Server iFTS offers means that the SQL query optimizer has access to new options to make your queries more efficient than ever. As an example, the following illustration highlights the SQL Server 2005 *Remote Scan* query operator that FTS uses to retrieve results from the full-text engine service. This operator is expensive, and the cost estimates are often inaccurate because of the reliance on a separate service. In the example query plan, the operator accounts for 47% of the total cost of the example query plan.



SQL Server 2008 iFTS provides the SQL query optimizer with a new and more efficient operator, the *Table Valued Function [FulltextMatch]* operator, shown in the following example query plan. This new query operator allows SQL Server to quickly retrieve results from the integrated full-text engine while providing a means for the SQL Server query engine to limit the amount of results returned by the full-text engine.



The new full-text search integration provides significant performance and scalability benefits over previous releases.

## Measuring Quality

The quality of search results can be measured using two primary metrics: *precision* and *recall.* Precision is the number of hits returned that are relevant versus the number of hits that are irrelevant. If you're having trouble with your car, for instance, and you do a search on *Cressida* on Google, you'll get many hits for the Shakespearian play *Troilus and Cressida* and one of the moons of Uranus, with later results further down the page referring to the Toyota product. Precision in this case is poor. Searching for *Toyota Cressida* gives you only hits related to the Toyota car, with very good or high precision. Precision can be defined mathematically using the formula shown in Figure 1-4, where *p* represents the precision, *n* is the number of relevant retrieved documents, and *d* is the total number of retrieved documents.

$$p = \frac{n}{d}$$

**Figure 1-4.** *Formula for calculating precision*

Recall is the number of hits that are returned that are relevant versus the number of relevant documents that aren't returned. That is, it's a measure of how much relevant information your searches are missing. Consider a search for the misspelled word *mortage* (a spelling mistake for *mortgage*). You'll get hits for several web sites for mortgage companies. Most web sites don't automatically do spell checking and return hits on corrected spelling mistakes or at least suggest spelling corrections. When you make spelling mistakes, you're missing a lot of relevant hits, or in the language of search, you're getting poor recall. Figure 1-5 is the mathematical definition of recall, where *r* represents recall, *n* is the number of relevant retrieved documents, and *v* is the total number of relevant documents.

$$r = \frac{n}{v}$$

**Figure 1-5.** *Formula for calculating recall*

Figure 1-6 is a visual demonstration of precision and recall as they apply to search. The large outer box in the figure represents the search space, or database, containing all of the searchable content. The black dots within the box represent individual searchable documents. The shaded area on the left side of the figure represents all of the documents relevant to the current search, while the nonshaded area to the right represents nonrelevant documents.

The complete results of the current search are represented by the documents contained in the dashed oval inside the box. The precision of this search, represented by the shaded area of the oval divided by the entire area of the oval, is low in this query. That is, out of all the documents retrieved, only about half are relevant to the user's needs.

The recall of this search is represented by the shaded area of the oval divided by the entire shaded area of the box. For this particular query, recall was low as well, since a very large number of relevant documents weren't returned to the user.

Precision and recall are normally used in tandem to measure search quality. They work well together and are often defined as having an inverse relationship—barring a complete overhaul of the search algorithm, you can generally raise one of these measures at the expense of lowering the other.

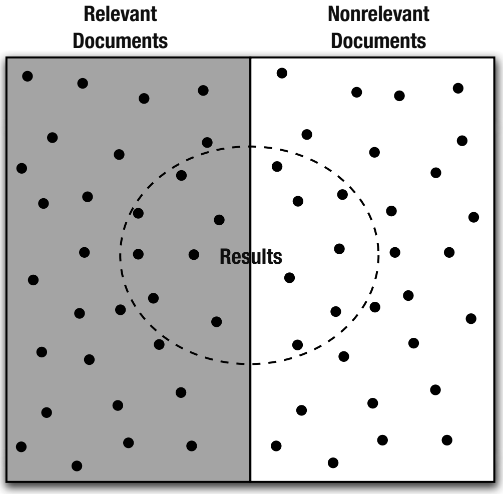


**Figure 1-6.** *Visual representation of precision and recall in search*

There are other calculations based on precision and recall that can be used to measure the quality of searches. The *weighted harmonic mean*, or *F-measure*, combines precision and recall into a single formula. Figure 1-7 shows the *F1* measure formula, in which precision and recall are evenly weighted. In this formula, *p* represents precision and *r* is the recall value.

$$F = \frac{2 \cdot (p \cdot r)}{p + r}$$

**Figure 1-7.** *Evenly weighted harmonic mean formula*

The formula can be weighted differently to favor recall or precision by using the weighted F-measure formula shown in Figure 1-8. In this formula, β represents the nonnegative weight that should be applied. A value of β greater than 1.0 favors precision, while a value of β less than 1.0 favors recall.

$$F_\beta = \frac{(1 + \beta^2) \cdot (p \cdot r)}{\beta^2 \cdot p + r}$$

**Figure 1-8.** *Weighted harmonic mean formula*

## Synonymy and Polysemy

Precision and recall are complicated by a number of factors. Two of the most significant factors affecting them are *synonymy* and *polysemy*.

*Synonymy*: different words that describe the same object or phenomenon. To borrow an example from Michel W. Berry and Murray Browne's book, *Understanding Search Engines*, a *heart attack* is normally referred to in the medical community as *myocardial infarction*. It is said that Inuit Alaskan natives have no words for war, but 10,000 words for snow (I suspect most of these words for snow are obscenities).

*Polysemy*: words and phrases that are spelled the same but have different meanings. *SOAP*, for instance, has a very different meaning to programmers than to the general populace at large. *Tiny Tim* has one meaning to the Woodstock generation and a completely different meaning to members of younger generations who've read or seen Dickens's *A Christmas Carol*. Another example: one of the authors met his wife while searching for his favorite rock band, Rush, on a web site. Her name came up in the search results and her bio mentioned that she loved Rush. Three years into the marriage, the author discovered that his wife's affection was not for the rock group Rush, but for a radio broadcaster of certain notoriety.

■**Note** For a more complete discussion of the concepts of synonymy and polysemy, please refer to *Understanding Search Engines-Mathematical Modeling and Text Retrieval* by Michael W. Berry and Murray Browne, (SIAM, ISBN 0-89871-437-0).

There are several strategies to deal with polysemy and synonymy. Among these are two brute force methods, namely:

- Employ people to manually categorize content. The Yahoo! search engine is an example. Yahoo! pays people to surf the Web all day and categorize what they find. Each person has a specialty and is responsible for categorizing content in that category.
- Tag content with keywords that will be searched on. For instance, in `support.microsoft.com`, you can restrict your search to a subset of the knowledge base documents. A search limited to the SQL Server Knowledge Base will be performed against content pertaining only to SQL Server Knowledge Base articles. These articles have been tagged as knowledge base articles to assist you in narrowing your search.

Currently, research is underway to incorporate automated categorization to deal with polysemy and synonymy in indexing and search algorithms, with particularly interesting work being done by Susan Dumais of Microsoft Research, Michael W. Berry, and others. Microsoft SharePoint, for example, ships with a component to categorize the documents it indexes.

## Summary

In this chapter, we introduced full-text search. We considered the advantages of using SQL Server full-text search to search your unstructured content, such as word processing documents, spreadsheets, and other documents.

We gave an overview of the goals and mechanics of full-text search in general, and discussed the SQL Server iFTS implementation architecture, including the indexing and querying processes. As you can see, there are a lot of components involved in the SQL Server iFTS implementation. What we explored in this chapter is a simplified and broad overview of iFTS architecture, which we'll explore further in subsequent chapters.

Finally, we considered search quality concepts and measurements. In this chapter, we introduced the terms and functions that define quality in terms of results.

In subsequent chapters, we'll explore all these concepts in greater detail as we describe the functional characteristics of the SQL Server iFTS implementation.