

DAXAT, INC.

Extensible Search Server 2005

Usage and Development Guide

EXTENSIBLE SEARCH SERVER 2005

Usage and Development Guide

© 2002-2005 Daxat, Inc.
35 Hidden Valley Rd
Groton, Massachusetts 01450
Phone 617.848.3908 • Fax 617.848.3914

LEGAL NOTICE INFORMATION

Daxat, the Daxat logo, and Daxat Extensible Search Server are either registered trademarks or trademarks of Daxat, Inc. Adobe, and Acrobat are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. Microsoft, Windows, Windows NT, Windows XP, Windows 98, Windows 2000 and Windows 2003 are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Google search engine is a trademark or registered trademark of Google Inc. in the United States and other countries. Other product and company names mentioned herein may be trademarks and/or service marks of their respective owners.

Disclaimer:

THIS DOCUMENTATION IS PROVIDED FOR REFERENCE PURPOSE ONLY. WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS DOCUMENTATION, THIS DOCUMENTATION IS PROVIDED "AS IS" WITHOUT ANY WARRANTY WHATSOEVER AND TO THE MAXIMUM EXTENT PERMITTED, DAXAT DISCLAIMS ALL IMPLIED WARRANTIES, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THE SAME. DAXAT SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, DIRECT, INDIRECT, CONSEQUENTIAL OR INCIDENTAL DAMAGES, ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS DOCUMENTATION OR ANY OTHER DOCUMENTATION. NOTWITHSTANDING ANYTHING TO THE CONTRARY, NOTHING CONTAINED IN THIS DOCUMENTATION OR ANY OTHER DOCUMENTATION IS INTENDED TO, NOR SHALL HAVE THE EFFECT OF, CREATING ANY WARRANTIES OR REPRESENTATIONS FROM DAXAT (OR ITS SUPPLIERS OR LICENSORS), OR ALTERING THE TERMS AND CONDITIONS OF THE APPLICABLE LICENSE AGREEMENT GOVERNING THE USE OF THE SOFTWARE.

© 2002-2005 Daxat, Inc. All rights reserved. Reproduction in whole or in any part or in any form or medium without express written permission is prohibited.

TABLE OF CONTENTS

Chapter 1: Introduction..... 1

ORGANIZATION OF THIS MANUAL	1
--------------------------------------	---

Overviews

Chapter 2: Installation..... 3

SERVER INSTALLATION	3
ABOUT SERVICE SECURITY	6
MANAGER INSTALLATION.....	7
CLIENT INSTALLATION	10

Chapter 3: ESS Manager 13

ABOUT COMPREHENSIVE MANAGEMENT	13
CONNECTING TO A SERVER.....	13
CONNECTION STRING KEYBOARD SHORTCUT	14
THE 3-PANE MANAGEMENT INTERFACE.....	15

Chapter 4: Search

Solution Design	17
THREE-TIER DESIGN.....	17
DATA REPOSITORY TIER.....	17
CLIENT TIER.....	18
INDEXING AND SEARCHING TIER	18

Chapter 5: Nodes..... 21

NODES AS DOCUMENTS	21
NODES AS DOCUMENT FRAGMENTS.....	21
ELEMENTS OF A NODE	21
NODE IDS	21
SEARCHABLE CONTENT	22
NAMED REGIONS.....	22
METADATA	22
AGGREGATES	23

DIGGING DEEPER – HOW TEXT IS STORED WITHIN A NODE.....	23
---	----

Chapter 6: Indexes 25

INDEX TYPES.....	25
SIMPLE INDEXES	25
REPLICA INDEXES.....	26
INDEX REPLICAS	26
INDEX SEGMENTS.....	26
INDEX ARCHITECTURE EXAMPLES	27
MAXIMIZING INDIVIDUAL QUERY PERFORMANCE	27
MAXIMIZING FAULT TOLERANCE.....	29
BALANCING PERFORMANCE AND FAULT TOLERANCE	30

Chapter 7: Data

Providers	31
-----------------	----

Chapter 8: Node

Transformations	33
-----------------------	----

Chapter 9: Queries 35

QUERY TARGETS.....	35
QUERY EXPRESSIONS	35
NAMED REGIONS.....	38
METADATA FILTERING EXPRESSIONS	38
AGGREGATES	39
ORDERING CRITERIA	39
QUERY PROCESSING: WORDS VS. TOKENS.....	39
SEARCHING FOR TOKENS.....	40

Chapter 10: Sample

Search Solution Walk- Through.....	41
SELECTING AND PREPARING A DATA REPOSITORY	41
ABOUT WEB CRAWLING	41
CREATING A NEW SIMPLE INDEX.....	41

TABLE OF CONTENTS

ADDING THE WEB CRAWLER AS A DATA PROVIDER.....	43
CONFIGURING THE WEB CRAWLER	44
BE CAREFUL!	45
STARTING THE WEB CRAWLER.....	46
QUERYING THE INDEX	46

Development

Chapter 11: Client-Side Development49

REFERENCING THE CLIENT SIDE LIBRARY 49 |

CONNECTING TO A SERVER.....49

EXECUTING A QUERY.....50

SORTING RESULTS.....	51
IMPROVING PERFORMANCE THROUGH PAGINATION.....	54

HANDLING QUERY RESULTS – THE QUERYRESPONSE OBJECT.....55

THE TOTALNUMBEROFMATCHES PROPERTY	55
THE QUERYEXECUTIONDURATION PROPERTY	55
THE SEARCHRESULTS PROPERTY	56
THE SEARCHRESULT.SCORE PROPERTY	56
THE SEARCHRESULT.METADATA PROPERTY	56
THE AGGREGATES PROPERTY	56

DELETING NODES.....61

UPDATING METADATA IN REAL TIME.....61

Chapter 12: Basic Data Provider Development ..63

REFERENCING THE ESS LIBRARIES 63 |

“ON DEMAND” DATA PROVIDERS.....63

THE ONDEMANDDATA PROVIDER PIPELINE	64
CREATING A NEW ONDEMANDDATA PROVIDER	64
CUSTOM PROPERTIES.....	65
INITIALIZE METHOD	65

NODE CONTENTS.....66

NODEIDCONTENT	66
TEXTCONTENT	67
TOKENCONTENT.....	67

ONDEMANDDATA PROVIDER YIELDNEXTNODE METHOD 68 |

DATA PROVIDER DEPLOYMENT.....69

Chapter 13: Advanced Data Provider Development71

ODDP PROCESSING PIPELINE REVISITED.....71

ODDP’S ARE NODE TRANSFORMS.....72

LIFECYCLE OF AN ODDP.....72

THE DATA PROVIDER CONTROLLER.....	72
SIMPLIFIED ODDP MODEL	73

NODE CONTENTS REVISITED 74 |

POSTING RUNTIME MESSAGES.....75

Chapter 14: Node Transform Development77

REFERENCING THE ESS LIBRARIES..... 77 |

THE NODETRANSFORM BASE CLASS..... 77 |

THREAD SAFETY OF NODE TRANSFORMS.....	78
--	----

THE INODETRANSFORM INTERFACE 78 |

THE INITIALIZE METHOD	78
-----------------------------	----

TABLE OF CONTENTS

THE ONNODESTART METHOD	78
THE ONNODEEND METHOD	79
THE CLEANUP METHOD	79
THE WRITE METHOD	79

THE NODETRANSFORM BASE CLASS

79

ADVANCED PROCESSING VIA NODECONTENT.PROCESSINGINFO RMATION	81
--	----

Chapter 15: Scorer Development.....

83

REFERENCING THE ESS LIBRARIES

83

THE ISCORER INTERFACE

84

SCORING CONTEXTS.....	84
-----------------------	----

A SIMPLE SCORING EXAMPLE.....

84

RELEVANCY RANKING.....

85

IFULLTEXTINDEX.COMPUTERELE VANCYSTATISTICS	86
TF·IDF SCORER EXAMPLE	88

Appendixes

Appendix A: ESS Query Language (ESSQL) Reference

93

ESSQL QUERIES

93

ESSQL EXPRESSIONS	93
WORDS.....	93
TOKENS.....	94
WILDCARDS	95
PHRASES	95
NAMED REGIONS.....	95
EXPRESSION GROUPING	95
COMPARISON EXPRESSIONS	96
QUERY TERM CORRELATION	96

ESSQL OPERATOR REFERENCE

97

AFTER	97
AND	97
ATLEAST.....	97

ATMOST.....	97
BEFORE	97
BETWEEN	97
CONTAINS	97
NEAR.....	98
NOT	98
OR	98
PRECEDES.....	98
SUCCEEDS	98
WITHIN.....	98

COMPARISON EXPRESSIONS OPERATOR REFERENCE

98

=	98
!=	99
<	99
<=	99
>	99
>=	99

VALUE EXPRESSIONS REFERENCE.....

99

().....	99
- (UNARY).....	99
*	99
/	100
+.....	100
- (BINARY)	100
INTEGER	100
REAL	100
STRING	100
DATE/TIME.....	100

Appendix B: Index Administration

101

SIMPLE INDEXES

101

Appendix C: Data Providers Reference....

103

COMMON PROPERTIES

103

CRAWLRATE PROPERTY	103
NODESBETWEENFLUSHES PROPERTY	103

FILE SYSTEM CRAWLER.....

104

NODE IDS	104
NODE CONTENTS.....	104

TABLE OF CONTENTS

CRAWLROOT PROPERTY	104
SIMPLE SQL SERVER CRAWLER	104
NODE IDS.....	105
NODE CONTENTS.....	105
CONNECTIONSTRING PROPERTY.....	105
ENDOF LASTRUN PROPERTY	105
IDCOLUMNS PROPERTY	105
NODEIDFORMATSTRING PROPERTY	106
SELECTSTATEMENT PROPERTY	106
STARTOFLASTRUN PROPERTY	106
TREATDUPLICATESASUPDATES PROPERTY	106
USEKEYASID PROPERTY	106
WEB CRAWLER	107
NODE IDS.....	107
NODE CONTENTS.....	107
CRAWLROOTS PROPERTY.....	107
PERMITDOMAINS PROPERTY	107

Appendix D: Node

Transforms Reference . 109

DUPLICATE NODE DETECTOR	109
IFILTER DOCUMENT CONVERTER.....	110
LOWER CASE NODE TRANSFORM ..	110
PROPERTY TAGGER	110
SIMPLE CONCEPT EXTRACTOR	112
DICTIONARYFILE PROPERTY	113
STRUCTURED STORAGE PROPERTY EXTRACTOR.....	113
URI READER	113

Appendix E: Scorer

Reference..... 115

SIMPLE SCORER	115
SIMPLERELEVANCYSCORER	115
RELEVANCYCRITERIA SUPPORT	116
ADVANCEDRELEVANCYSCORER....	116
RELEVANCYCRITERIA SUPPORT	117

Appendix F:

Troubleshooting 119

CANNOT CONNECT TO SERVER	119
TOO MUCH/TOO LITTLE EXCEPTION DATA AT CLIENT	119
IFILTER DOCUMENT CONVERTER NOT CONVERTING DOCUMENTS	120


Introduction


Welcome to the Daxat Extensible Search Server 2005.

Thank you for selecting the Daxat Extensible Search Server 2005 (ESS). ESS is one of the most powerful and extensible enterprise search platforms available. Built on-top-of Microsoft .NET, ESS may be customized in ways unlike any other enterprise search engine, all the while maintaining an extremely high level of developer productivity thanks to the abundance of world-class .NET development tools and libraries.

Organization of this Manual

ICON KEY

 Valuable information

 Test your knowledge

 Programming example

 Review

The manual is divided into three major sections. The first section, **Overviews**, contains overview materials for each of the major technological areas of the Daxat Extensible Search Server. This section is valuable to both newcomers to the enterprise search space and veterans who need to learn Daxat's terminology.

The second section, **Development**, is a step-by-step walkthrough on how to design and develop plug-ins for the Daxat Extensible Search Server. This section should be used in conjunction with the "ESS 2005 Developer's Reference" API reference guide.

The final section, **Appendixes**, covers a wide range of topics; in particular, several appendixes are reference guides to the plug-ins included with the Daxat Extensible Search Server.

I. Overviews

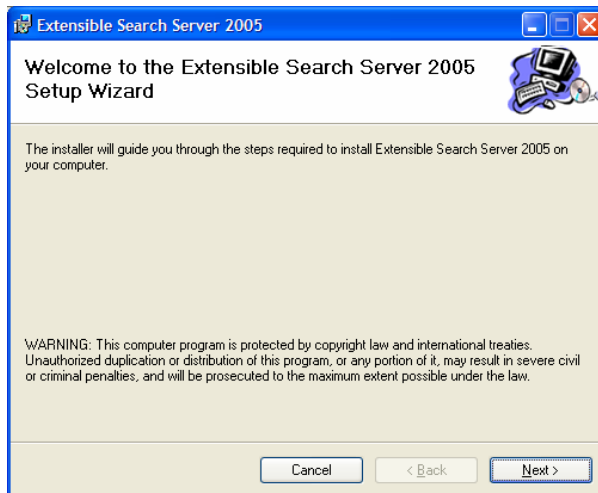
Installation

Server, Manager and Client Installation Instructions.

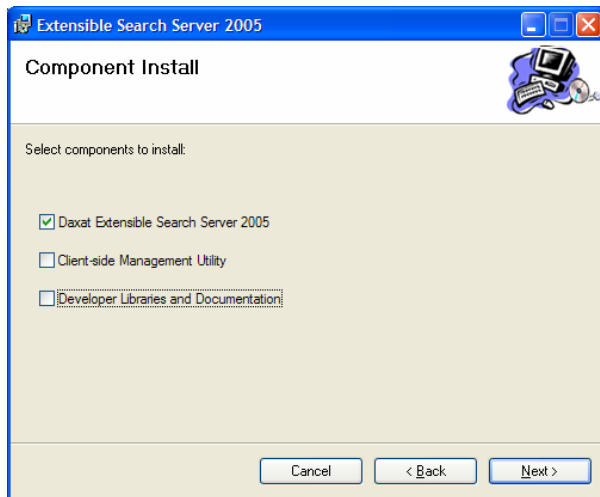
There are three distinct ESS installation types – Server, Manager and Client. Any combination of these three types is acceptable on any given machine. A Server installation will install and start running an instance of the Daxat Extensible Search Server on the target computer. A Manager installation will install the ESS Manager, which may be used to manage any local or remote Daxat Extensible Search Server. A Client installation will install the libraries (dlls) required to communicate and extend a Daxat Extensible Search Server.

Server Installation

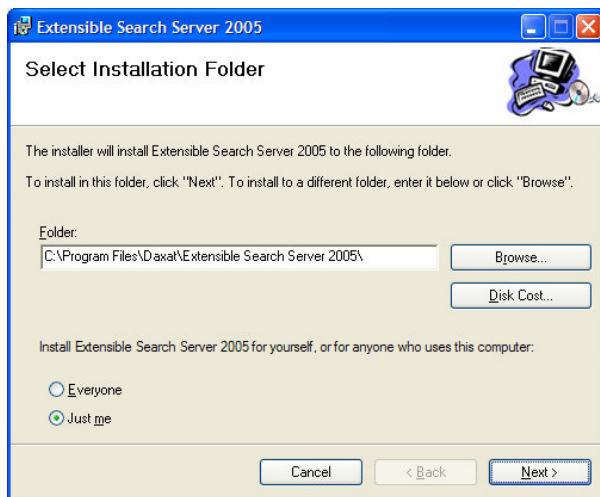
In order to install an instance of the Daxat Extensible Search Server, run `setup.exe`.



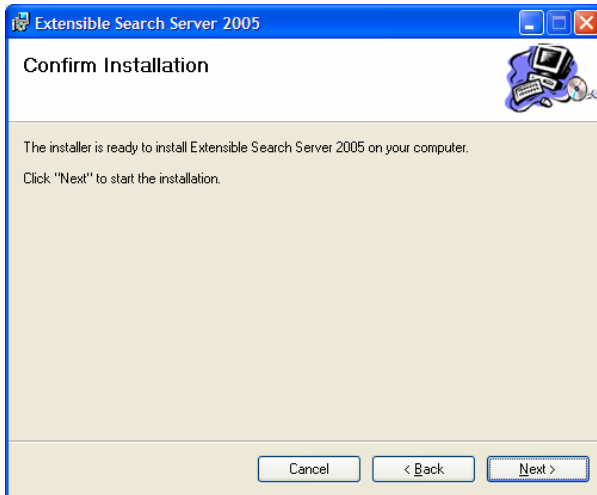
On the initial dialog click the Next button.



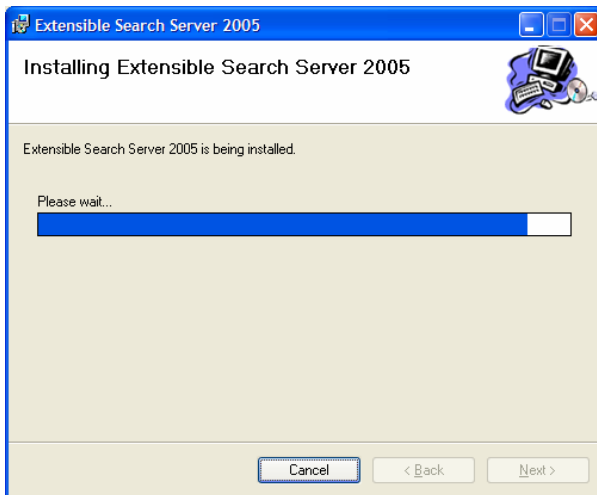
This will bring up the “Component Install” dialog. Ensure that the “Daxat Extensible Search Server 2005” component is checked. The other two components are optional.



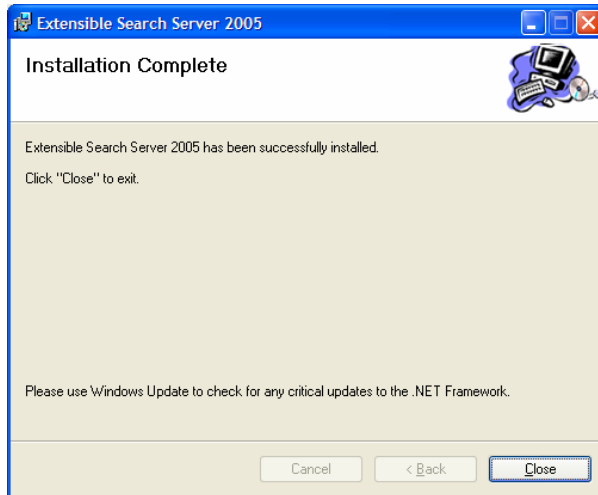
Clicking the Next button reveals the “Select Installation Folder” dialog. Modify the installation path if desired and click Next.



The installer now has all the information necessary to complete the installation. Click Next.



The server files will be copied to the computer.



In a matter of moments, the installation will be complete.

The Windows service **Daxat ESS** is now installed on the computer. When installation is complete, **the installer will start the service, and set the startup type to 'Automatic.'**

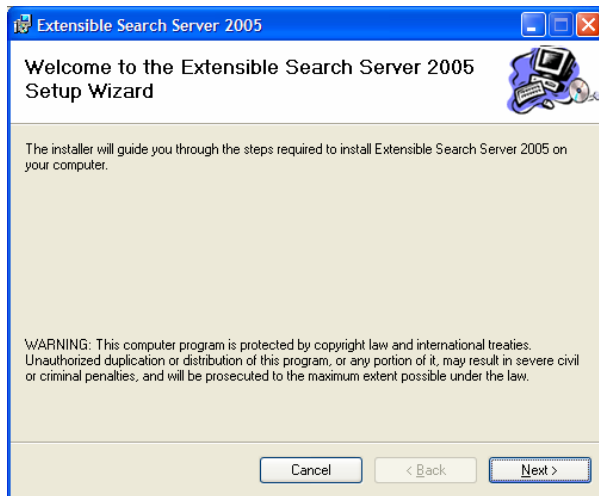


About Service Security

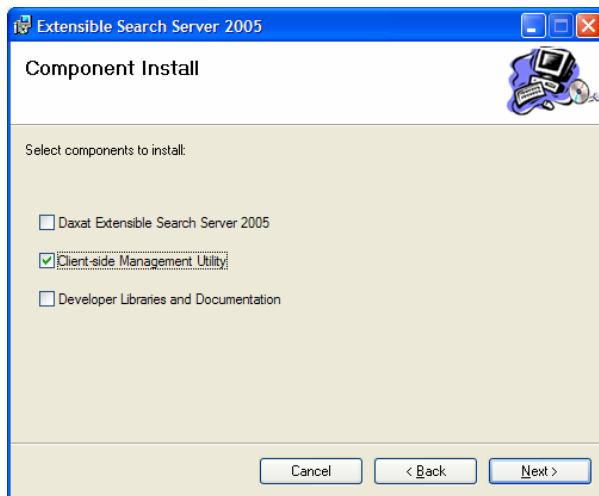
The Daxat ESS service is installed as the LOCAL SERVICE account on Windows Server 2003 and Windows XP, and as the LOCAL SYSTEM account on Windows 2000. This may directly impact the ability of ESS to extract information from data repositories and to create indexes. For example, by default the LOCAL SERVICE account has very few rights on the local system and none off of the local machine. If ESS is running as LOCAL SYSTEM and you desire to create a new index in D:\MyIndex, LOCAL SYSTEM will need to have read/write access to D:\MyIndex. You can easily change the log on account for Daxat ESS via the Windows Services administration tool.

Manager Installation

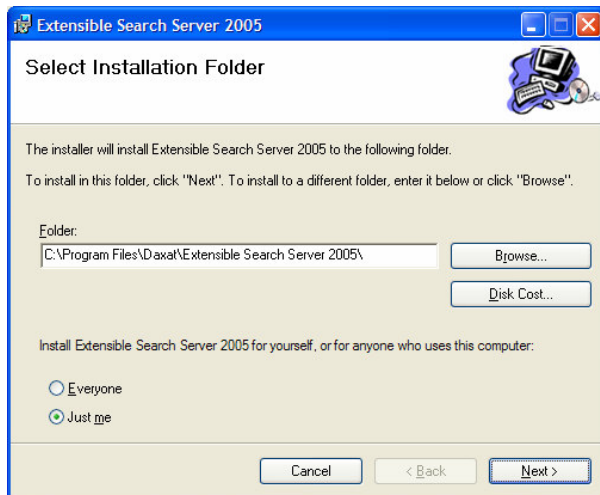
The ESS Manager is a graphical user interface management utility for managing Daxat Extensible Search Servers, both installed locally and remotely. To install the manager, run `Setup.exe`.



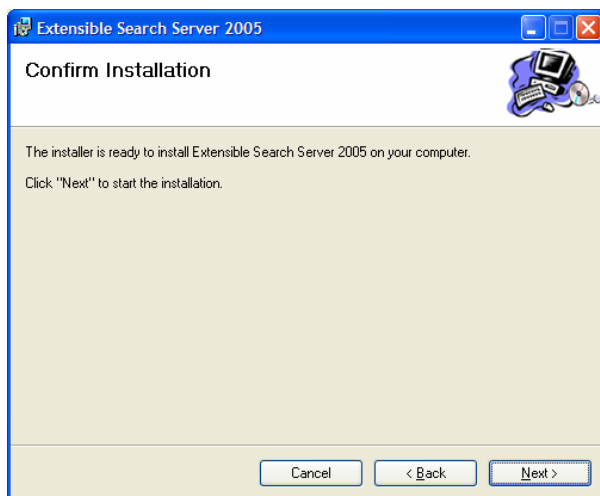
On the initial dialog click the Next button.



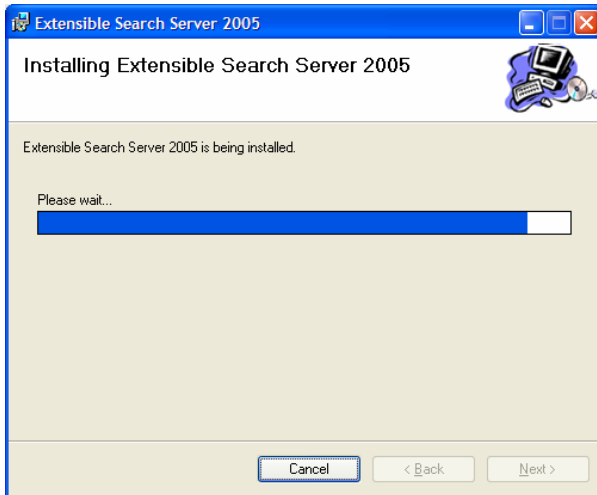
This will bring up the “Component Install” dialog. Ensure that the “Client-side Management Utility” component is checked. The other two components are optional.



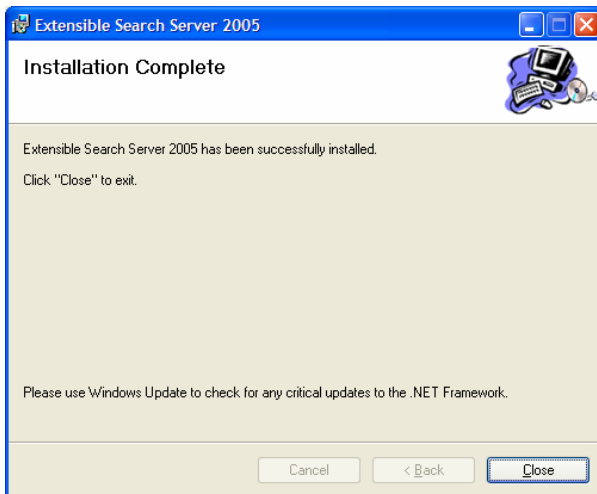
Clicking the Next button reveals the “Select Installation Folder” dialog. Modify the installation path if desired and click Next.



The installer now has all the information necessary to complete the installation. Click Next.



The management application files will be copied to the computer.

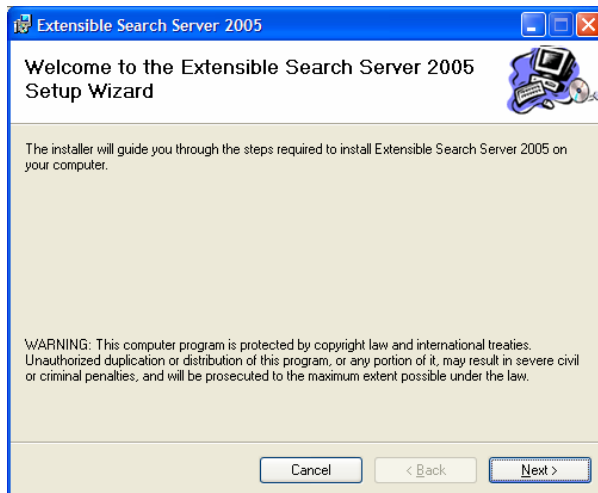


In a matter of moments, the installation will be complete.

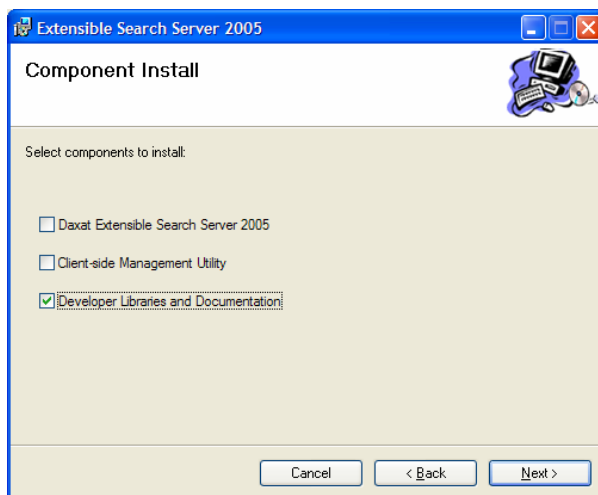
The ESS Manager is now installed on the local machine. The manager is located in the Start menu as Daxat → ESS → ESS Manager.

Client Installation

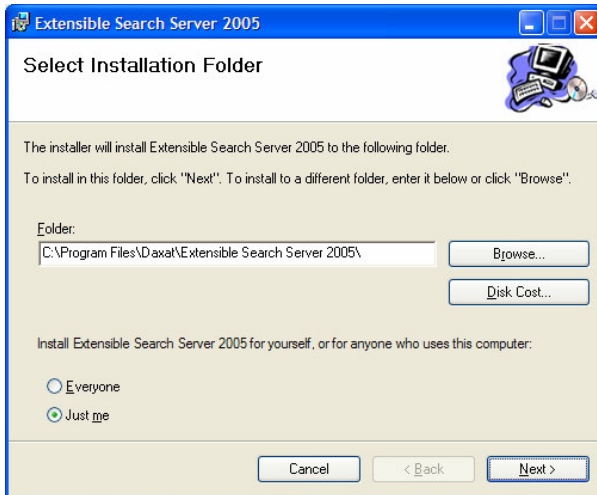
In order to develop extensions for the Daxat Extensible Search Server, or simply to provide connectivity between an application (such as ASP.NET) and a remote Extensible Search Server, it is necessary to install the client-side libraries. Start by running `Setup.exe`.



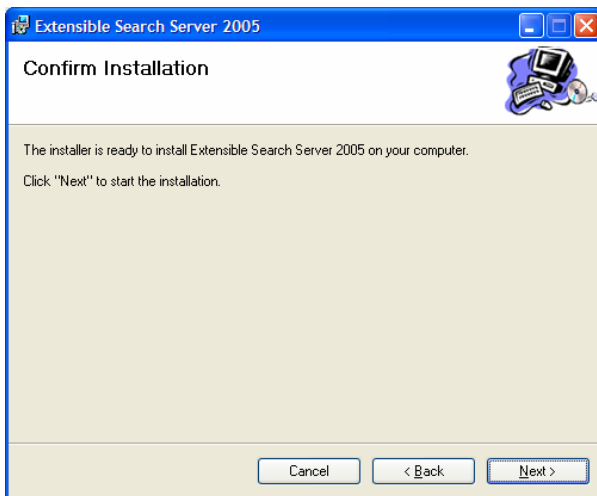
On the initial dialog click the Next button.



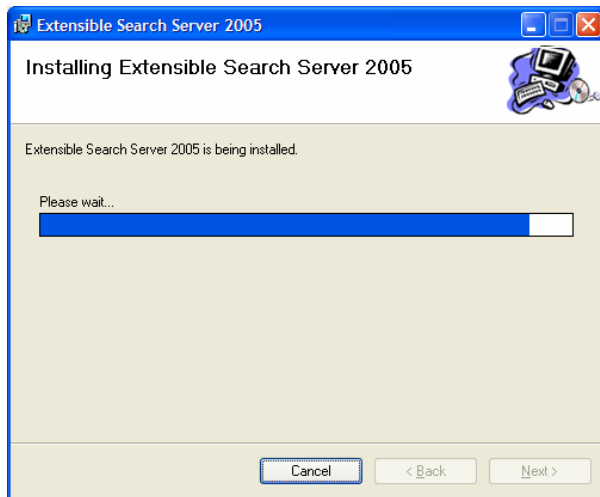
This will bring up the “Component Install” dialog. Ensure that the “Developer Libraries and Documentation” component is checked. The other two components are optional.



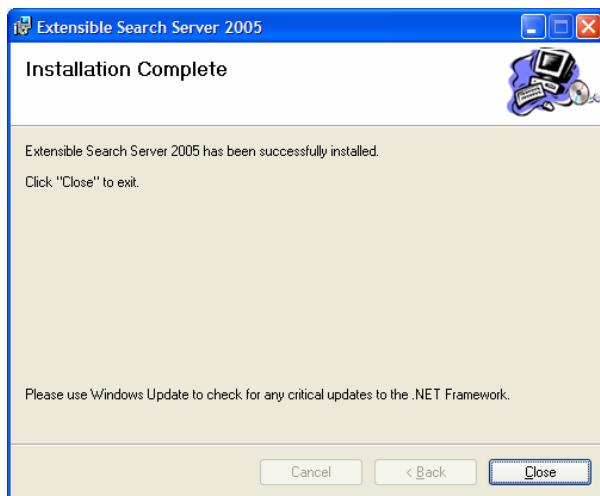
Clicking the Next button reveals the “Select Installation Folder” dialog. Modify the installation path if desired and click Next.



The installer now has all the information necessary to complete the installation. Click Next.



The management application files will be copied to the computer.



In a matter of moments, the installation will be complete.

A development reference guide will be installed. It can be found in the Start menu as Daxat → ESS → ESS 2005 Developer's Reference.

ESS Manager

A Quick Overview of the ESS Management Application.

The ESS Manager provides a quick and easy means to manage one or more Daxat Extensible Search Servers. Although this graphical utility may be used to perform the most common management functions, it does not provide comprehensive access to every possible management operation.

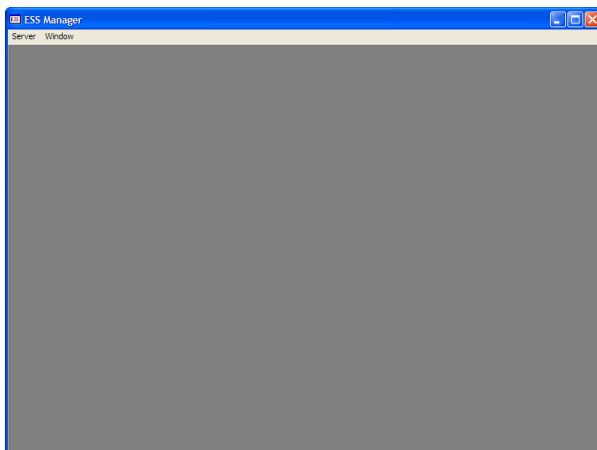


About Comprehensive Management

Each Daxat Extensible Search Server provides for comprehensive management via the developer APIs. The ESS Manager is built-on-top of these publicly exposed and fully documented APIs. Every management operation that can be performed with the ESS Manager can also be performed by calling one or more of the management APIs. This makes integrating complex ESS management operations into other tools simple and easy.

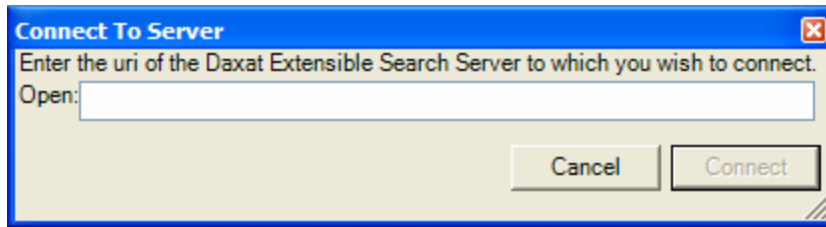
Connecting to a Server

To start the manager, from the Start menu select Daxat → ESS → ESS Manager.



When the manager is started, it is not connected to any Extensible Search Servers and therefore has a blank interface. Assuming there is a server installation available, either on the local machine

or on the local network, from the Server menu choose Connect. This will bring up the Connect to Server dialog.



Connect to Server dialog box.

Daxat Extensible Search Servers are referenced by .NET Remoting Compliant URIs. Exactly what that means will be covered in the advanced server configuration section. To connect to an Extensible Search Server installed with the system defaults, the connection string is

```
tcp://hostname:20869/ess
```

where *hostname* is the name of the machine hosting the Extensible Search Server, or *localhost* to connect to the Extensible Search Server running on the same machine as the ESS Manager.. Type in the connection string and click Connect.

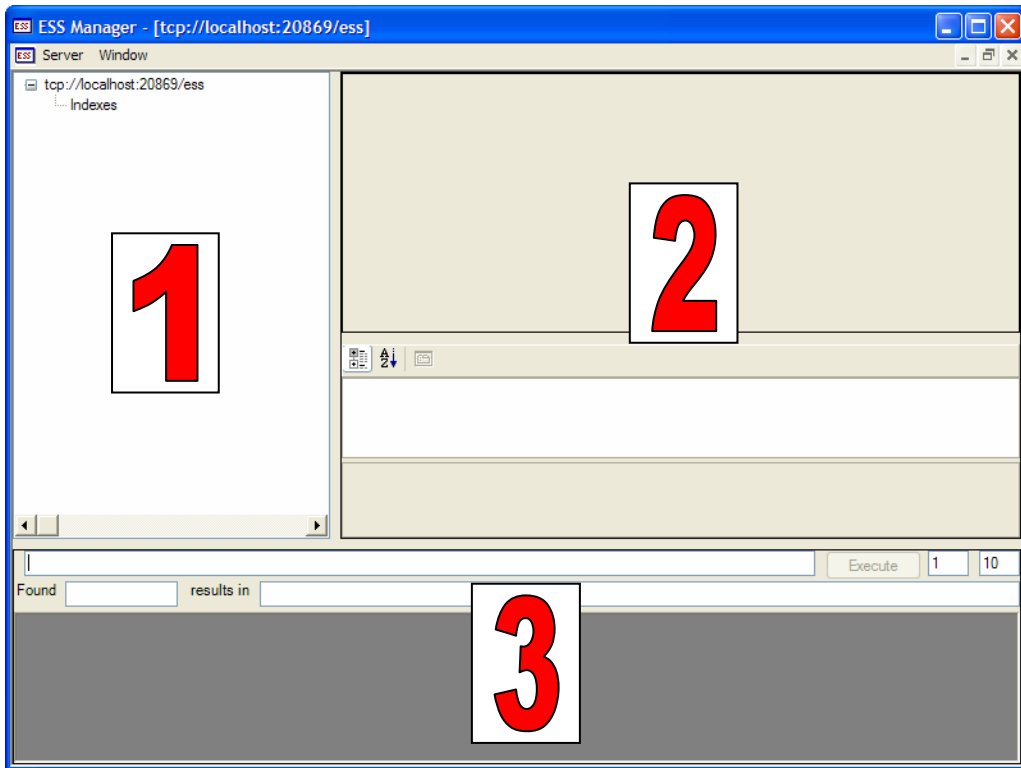


Connection String Keyboard Shortcut

To quickly generate the default connection string to any machine, simply type in the name of the machine and press CTRL+ENTER. The connection string will automatically expand to `tcp://hostname:20869/ess`.

The 3-Pane Management Interface

Once connected to the server, the manager displays a multi-pane interface for the server. The interface is divided into three main parts.



Pane #1 contains a tree control which lists the various elements of the server which may be managed. Pane #2 displays the properties for the item currently selected in pane #1. Pane #3 is a query interface for submitting queries against the server.

When an Extensible Search Server is first installed, it contains no information. In order to put information in the server, so that information may be searched, it is necessary to create an index.

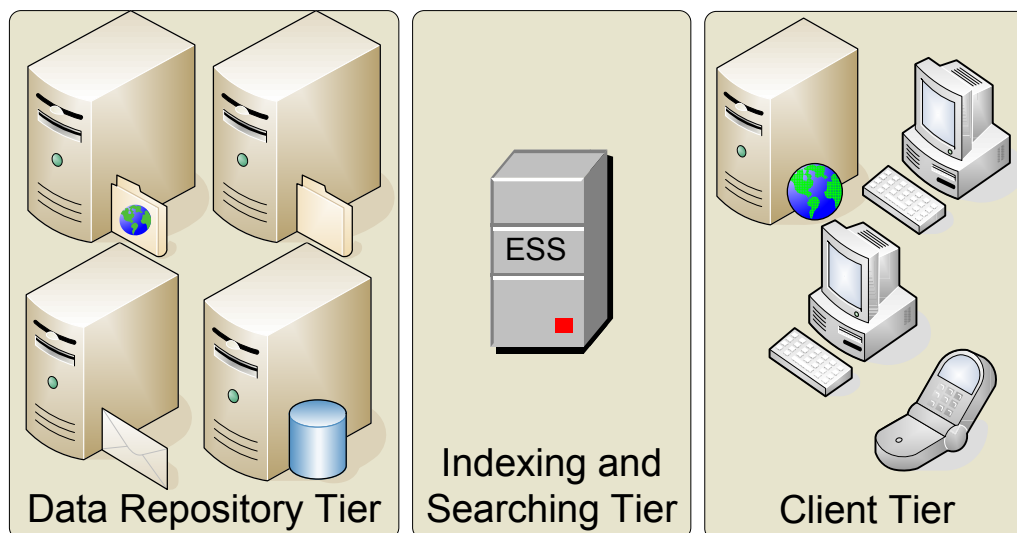
Search Solution Design

A High-Level View of the Design of a Search Solution.

Search solutions of virtually any size and scale have a very similar design. There are three major tiers to any search solution. Search solutions built to handle large numbers of documents, large numbers of concurrent users or for fault-tolerance simply expand upon one or more of the three basic tiers.

Three-Tier Design

Most search solutions have three tiers: the Data Repository Tier, the Indexing and Searching Tier and the Client Tier.



Data Repository Tier

The data repository tier contains the data for which the search solution will be built. When clients submit queries to the system, those queries will return search results for matches against documents contained within the data repository.

The data repository tier can contain virtually any kind of information, provided that information can be expressed in a textual format. For example, a Microsoft Word document would first need to be stripped of all formatting and graphics, leaving just the plain text of the document. A photograph would have to have some associated textual data, such as a description of the scene or the date and time when the picture was taken. An audio file may need to be converted into written text by voice recognition software. *Data Providers*, running inside of the Extensible Search Server, are responsible for collecting the information from the data repository, and coordinating with transformation programs which convert the data from its native format into plain text. The Daxat Extensible Search Server ships with data providers which can extract data from common repositories such as web servers, file servers and relational databases, and transformations which can read and convert common file formats, such as Microsoft Office documents; it is a reasonably easy task to write custom data providers and transformations which can access data repositories and convert file formats not supported out-of-the-box.

Client Tier

The client tier contains consumers of the search technology. This can include everything from web servers which expose custom search pages for web browsers to custom applications which communicate directly to the Extensible Search Server. There is virtually no limit to the type of devices which can communicate with the search server, how the search results are displayed or manipulated, and the channel over which the communication takes place or the security and authentication applied to the communication channel.

Indexing and Searching Tier

The indexing and searching tier is where the Daxat Extensible Search Server lives. The server is responsible for coordinating data providers, which read from the data repository tier, transformations, which convert native file formats into text formats, and client queries. At the heart of the server is an *index*. An index is a special representation of the data collected from the data repository tier stored in such a way to permit extremely fast and complex searches against that data. When a client submits a query to the server, although the experience is one of searching the data repository, in fact the query is against the index of the data repository. When a data provider fetches data from a repository, it packages that data as a *node* which is submitted to the Extensible Search Server for inclusion within some index. Typically, a node directly corresponds to a single document or a single web page, but a node can represent as little as a fragment of some source data (e.g. a single paragraph from a long document) or may represent a complex combination of multiple pieces of data (e.g. a join of several relational database tables and the contents of one or more associated files). Most importantly, when a query is submitted to the server, the search results will directly correspond to indexed nodes; the node is the finest level of granularity for a search result. If search results should directly map to entire documents, then each document would be indexed as a single node. If, instead, the desire is to have search results which map to individual sentences extracted from multiple documents, then each sentence should be indexed as a unique node.

There is no practical limit to the number of nodes, the number of different types of data or the number of data sources which may be stored in a single index, and there is no practical limit to the number of different indexes which may be maintained by an Extensible Search Server. The only limiting factors are the available resources (disk, RAM, CPU and network bandwidth) of the computer hardware.

When the amount of data to be indexed or the number of concurrent users grows too great to be handled by a single server, multiple machines may be combined together to create indexing and searching farms. The farm architecture can also be used to address issues of fault tolerance, facilitating the design of search applications which can survive one or many machine failures.

Nodes

An Overview Extensible Search Server Nodes.

Nodes are the basic unit of storage of searchable content contained within an Extensible Search Server index. There is a one-to-one correspondence between a search result and a node. When a query is submitted to ESS, the returned search results represent individual nodes.

Nodes as Documents

In the simplest case, a single node represents a single document. For example, if a search solution is designed to provide searching for a web site, each web page would be stored in the search index as a unique node, and queries against that index would result in search results which map directly to the original web pages. This is how classic internet search engines, such as the Google search engine, operate.

Nodes as Document Fragments

As search results corresponded one-to-one with nodes, it is important to decide when designing a search solution what level of granularity is required within the search results. For example, a Shakespearean play research tool may be designed so that searches match individual character quotes; in this case, each node would not simply represent a single Shakespearean play but instead an individual passage from a play. Shakespeare wrote 38 plays, each containing hundreds of passages. Therefore, this example index would contain thousands of nodes, not just 38.

Elements of a Node

What, exactly, is a node? A node is a representation of data culled from a data repository consisting of three major elements – the node id, searchable content and metadata.

Node ids

The *node id* is the unique identifier of the node. Every node must have a unique node id. The node id may be just about anything – in programming terms, it may be any valid string. For example, an index of web pages will likely contain nodes whose ids are simply the urls of the indexed web pages. Under normal usage it is not necessary to be concerned about node ids – the data provider will automatically assign them. For example, the included web crawler data provider

which is used to build an index for a web site automatically uses each web page's url as the node id. Each individual search result will include the node id of the matching node. In the web page searching example, this means that every search result will include the url of the matching web page, making it extremely easy to create a search interface whereby the user may click on a search result and be directed to the matching web page.

Searchable Content

The *searchable content* of a node is the text against which user queries are normally performed. For example, a query for baseball would return a set of search results for all nodes which contain the word baseball in their searchable content.

Searchable content is stored within the index in way which is highly optimized for searching yet is impractical to return as part of a search result. Therefore, while the baseball query will return search results *representing* the nodes containing the word baseball within their searchable content, the search results will not normally include the actual contents of the indexed data. If it is desired to display the full text of one or more search results, it will normally be necessary to access the original source document. This is how classic internet search engines operate – the search results include the urls of the source web pages, but not the full contents of the original web pages. Some search engines maintain cached copies of the source documents; the Daxat Extensible Search Server can do the same, but this cache is not maintained within the searchable content of a node.

Named Regions

Searchable content can be further subdivided into *named regions*. Named regions, also known as fields, are pieces of the searchable content marked with a given name. For example, a node may have a named region called `Title`, the contents of which are the title of the source document. When searching for nodes, queries can be limited to particular named regions. For example, it is possible to find all nodes containing the word `shrub` which also have either the word `maintenance` or `care` in the named region `Title`. There is no limit to the number of named regions associated with a given node. Each node can have a unique set of associated named regions. A named region may be declared more than once within a given node (e.g. a node may have multiple `Author` named regions, each one containing the name of a different co-author of the source document). Named regions can be nested (e.g. find all nodes which contain `Robert` in the `Author` named region contained within the `PriorVersion` named region.)

Metadata

Along with the node id, each node can also optionally contain metadata. Metadata is data which describes the node. Examples of metadata include the title of the document represented by the node, the date and time which the node was added to the index, the size of an image, in bytes, represented by the node, etc. There are no limits on which metadata may be associated with a given node, and each node can have a unique set of metadata – even within the same index. For example, an index containing information about images, where some images are scanned and other are photographed, may include metadata about the camera settings for just the photographs.

Metadata may be strongly typed – that is to say, dates can be stored as true dates within the index; numbers can be stored as true numerical values. When processing metadata, it may be processed as strongly typed data.

If metadata has been assigned to a node, then that metadata will be returned with any search result representing the node. For example, an index of classical works of literature may contain one node for each literary work, and the metadata for each of those nodes can contain information such as the title of the work, the author, and the publication date.

Although searchable content is searchable but not returnable, the reverse is not true for metadata. While metadata is certainly returnable as part of a search result, metadata is also searchable. Given that metadata is searchable, what is the value of searchable content? There are two major differences between metadata searching and searchable content searching. First off, searching against searchable content is typically significantly faster than searching against metadata. Secondly, metadata searching does not support the rich textual searching operators which searchable content supports.

Metadata searching does have some advantages over searchable content searching. The main advantage is that metadata searching can be strongly typed. For example, if it is possible to search for all nodes whose metadata for PublicationDate is between 5-Apr-2003 and 11-Jul-2004. As such, metadata searching is best used to perform *filtering* on top of a traditional full-text search. For example, to “find all documents containing the word baseball and published between 5-Apr-2003 and 11-Jul-2004” or to “find all catalog items whose description contains the word chair or seat, whose color is red and whose price is between \$145.00 and \$200.00.” The Extensible Search Server query language makes it easy to combine traditional searching and metadata searching as part of a single query.

Aggregates

Aggregates are a special application of metadata. Any piece of metadata may be aggregated at query time. When metadata is aggregated, the search results will include a count of the number of unique values found for the metadata element contained within the search results. For example, a query for blue SUV against an index of used cars which aggregates on CarMake could display not only that there are 43 matching vehicles, but also that 27 are Honda, 15 are Lexus and 1 is Ford. The aggregate, in turn, can then be used for “drill-down navigation,” whereby a click on Lexus reduces the search results from the 43 matching blue SUVs down to just the 15 matching blue Lexus SUVs.

Digging Deeper – How Text is Stored within a Node

At a high-level, the searchable content of a node is simply the words which make up the node – for document-based nodes, it is easy to think of the searchable content as the same as the unformatted text comprising the source documents. In reality, the searchable content of a node is more complex, comprised of positional tokens.

The process of converting plain text to searchable content begins by breaking the source text into *words*. A word is a sequence of alphanumerical characters. White space and non-alphanumerical characters are considered to be word boundaries. For example, the text

He purchased 5 pressure valves, item #ABC-123, for a total of \$87.50.

consists of 14 words: He purchased 5 pressure valves item ABC 123 for a total of 87 50. Each of the words is positioned one after the other; He is at position #1, purchased at position #2, etc., up through 50 at position #14.

Every word, in turn, is mapped to one or more tokens. A token is an atomic representation of a piece of text; a token is made up of a sequence of *any* characters. The standard behavior of the Daxat Extensible Search Server is to create at least one, and potentially two tokens for each word. The first token created will be an exact copy of the word. If the word contains any upper-case characters, then a second token will be created based upon the all lower-case version of the word. In the previous example, the 14 words would produce 16 tokens: He he purchased 5 pressure valves item ABC abc 123 for a total of 87 50.

For a positional standpoint, there are still only 14 positions. When a word produces more than one token, those multiple tokens occupy the same position. Both He and he are located at position #1, and both ABC and abc are located at position #7.

The importance of tokens and positions is a direct consequence of the way the Daxat Extensible Search Server processes queries, as will become evident in during the discussion regarding queries.

Note that words only contain alphanumerical characters, but tokens may contain any characters. Since tokens are normally derived from words, it may appear impossible to create tokens containing non-alphanumerical characters. In many common search solution scenarios this is true and desired behavior. However, it is possible to place tokens directly within a node which are not derived from words. For example, a search solution containing information about programming languages can be built which will preserve tokens such as C++ and C#, rather than reducing those down to the word 'C'.

Indexes

An Overview of Extensible Search Server Indexes.

Whereas nodes represent individually searchable items, *indexes* represent composite searchable collections of nodes. A query against an Extensible Search Server is directed at an index; a single server may host multiple indexes.

Data providers read data from data repositories (web sites, databases, etc.), package that data into individual nodes, and submit those nodes to an index for storage. Queries are performed against a given index and return search results representative of the matching nodes found within the index.

Index Types

There are two types of indexes – *Simple Indexes* and *Replica Indexes*. A simple index is an index which is hosted by a single Extensible Search Server and lacks any formal form of fault tolerance or scalability. By their very nature, simple indexes are easy to set up and maintain. Replica indexes are more complex, but have the advantages of being able to span multiple machines to support “scale-out” scalability and fault tolerance.

For the most part, from both the end-user’s perspective and a programmer’s perspective, there is no difference between a simple index and a replica index. The interfaces to both types of indexes are exactly the same. Moreover, it is a simple task to convert a simple index into a replica index, should there ever be a need for ESS-managed fault tolerance or scalability.

Simple Indexes

Simple indexes are indexes which are both easy to create and maintain, at the expense of being fault tolerant or supporting a scale-out architecture. That is not to say that simple indexes do not perform well; on the contrary, a typical server can host a simple index containing hundreds of thousands of documents and handle multiple, simultaneous queries. A single query against a simple index is a single threaded operation; in other words, adding more CPUs to a machine will not increase the speed at which individual queries are processed against a simple index. However, simultaneous queries may run on different CPUs against a simple index, so adding additional CPUs will increase the concurrent query capacity of the server.

A simple index is defined by two properties – a unique name on the server and a file directory into which the index will store its data. For example, an index which is to contain corporate email may be named `CorporateEmail` and may be stored in `C:\Indexes\CorpEmail`. There are no other properties associated with an index – any index may contain any type of information, and any combination of information. The amount of data which may be stored within a single index is governed by the resources available on the machine; there are no practical hard limits on the number or size of the nodes stored within an index.

Once defined, an index will remain empty until at least one data provider is assigned to it and that data provider begins to populate the index with nodes.

Replica Indexes

Replica indexes are the basis for both fault tolerant and scale-out architectures. Replica indexes are comprised of one or more *index replicas*; index replicas are comprised of one or more *index segments*.

Index Replicas

An index replica is a copy of an index. A replica index comprised of a single index replica contains one ‘copy’ of the index; in other words, the index is not fault tolerant because it has not been duplicated. A replica index comprised of two or more index replicas is fault tolerant; a copy of every node placed into the index is placed into each index replica. Should an index replica fail, the other index replicas can continue to respond to queries.

A replica index must have at least one index replica. Each index replica, in turn, must have at least one index segment.

Index Segments

An index segment is a partial slice of an index. An index replica must contain at least one index segment. If an index replica contains only one index segment, then that index segment contains the entire contents of the index replica; if an index replica contains multiple index segments, each segment contains a unique piece of the index replica. Operations performed against an index replica must be performed against every segment of the replica; often, the results of the operations must be combined to produce a single returned result. For example, a query against an index replica segmented into 2 segments must be performed against both segments individually, and then the search results returned from both segments must be combined into one set of search results. Operations against index segments may be performed in parallel; a single query against a segmented index may make use of multiple CPUs, as many as one CPU per index segment. Therefore, adding additional CPUs may increase the performance of individual queries against a segmented index.

There are two types of index segments – local index segments and remote index segments. Both types of segments are defined the same way – via a URI. If the URI for an index segment is the name of a directory on storage, then the index segment is a local index segment and is exactly the same as a simple index. However, if the URI for an index segment is the URI of an Extensible

Search Server, then the index segment is a remote index segment and operations performed on the index segment will be forwarded to the named Extensible Search Server. **The URI of a remote index is the URI of the Extensible Search Server hosting the index, appended with #indexname.** For example, the URI for the index named CorporateEmail hosted on a server located at tcp://ess-1:20869/ess is tcp://ess-1:20869/ess#CorporateEmail.

Index Architecture Examples

All simple indexes have the same architectural design; they exist within a single Extensible Search Server, have a name and are associated with a storage directory reachable by the server.

Machine name: ESS1

Physical Index, Name: WebSite

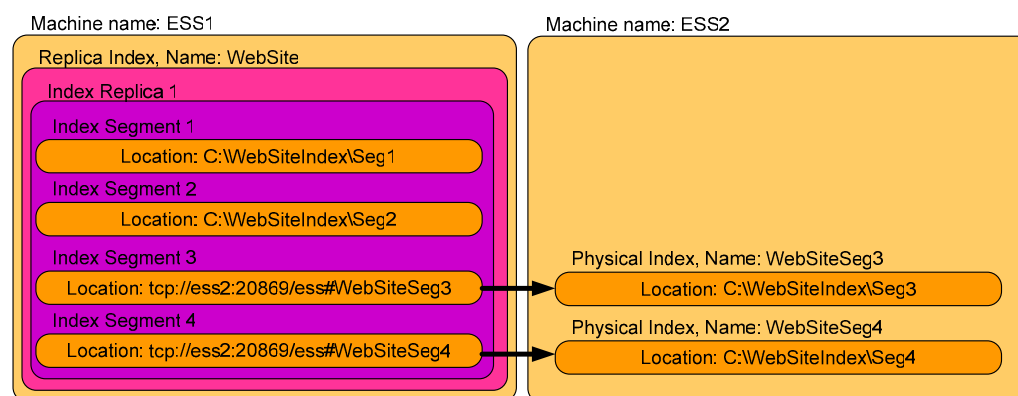
Location: C:\WebSiteIndex

Simple Index architecture

A two-machine replica index can be laid out in multiple ways to achieve different goals. For example, consider two servers each with two CPUs – a total of four CPUs. If the performance of individual queries is paramount, then the index can be split into four segments – two segments each per machine. This will permit a single query to make use of all four CPUs simultaneously. If reliability is paramount, the index can be replicated two or four times, providing fault tolerance.

Maximizing Individual Query Performance

There are many ways to arrange 4 segments into a top-level replica index, each with various pros and cons. The easiest design involves a single top-level replica index on the first machine with a single index replica containing four segments, two local and two remote. The remote segments point to two physical indexes on the second machine.



One index replica, four segments

Although this design is easy to set up and will balance well (i.e. a single query will make use of all 4 CPUs and each segment will be populated with approximate $\frac{1}{4}$ of the total nodes), it is not necessarily the best design for future growth. For instance, adding 2 more machines to the farm would require defining four more remote segments on ESS1. If those machines have 4 CPUs instead of 2, then 8 additional remote segments will need to be defined.

An alternative design is to assign index segments to the available machines in a divide-and-conquer methodology. Each machine will host a replicated index which, in turn, is segmented to match the number of CPUs on the machine. A top-level replica index will then bind all of the machine specific replica indexes together, containing one segment per machine. Adding a new machine simply requires adding a single new segment to the top-level replica index, regardless of the number of CPUs in the new machine.

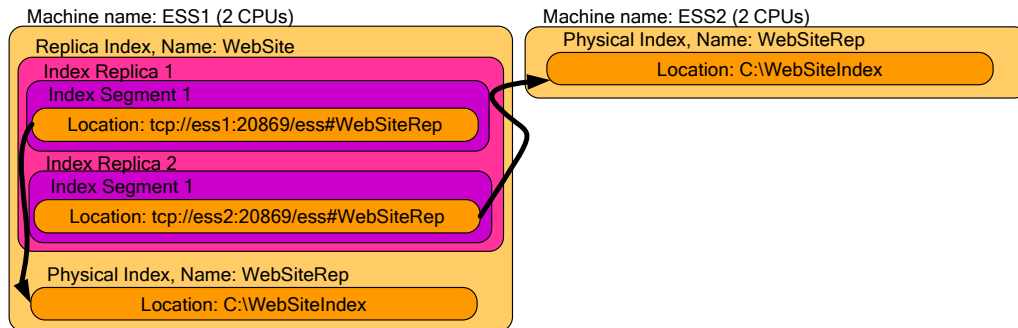


Replica of replicas

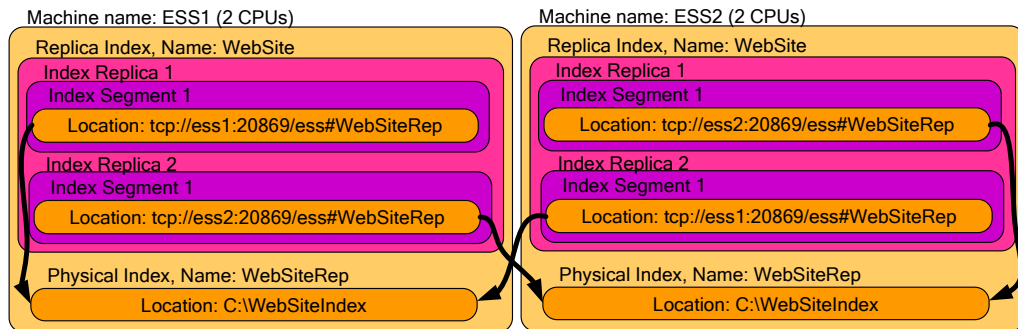
Maximizing Fault Tolerance

Often system up-time is more important than system performance. The importance of doubling the reliability of a system may outweigh the importance of doubling the performance of that system, especially if the majority of queries are resolved with sub-second response times.

Again consider the scenario of two, dual-processor search servers. While either machine could fail, either due to hardware or software fault, it is highly unlikely that both machines would fail at the same time. In the simplest fault-tolerant design, each machine would host a replicated copy of the index, with the top-level replica index existing on one of the machines.



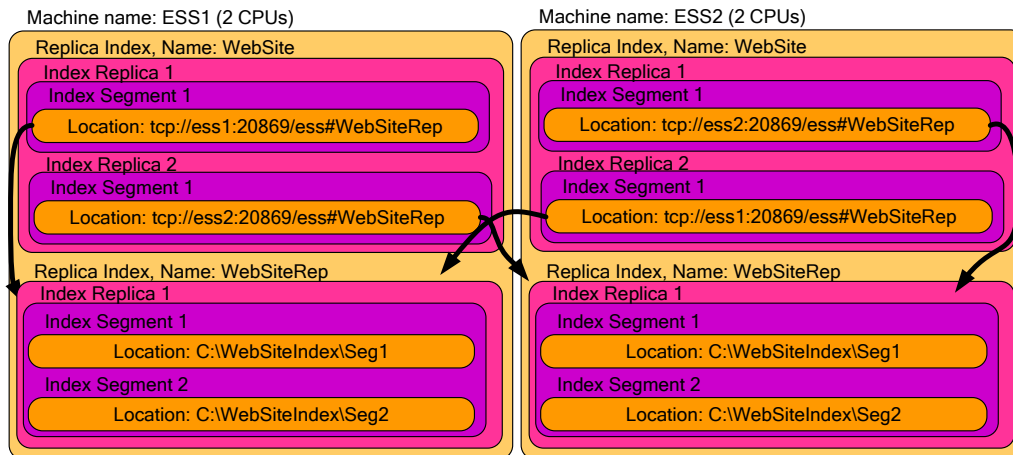
In order to protect the top-level index replica, two instances can be created, one per machine.



In this configuration, the index WebSite may be queried against on either server ESS1 or server ESS2. The query, in turn, may be processed by either server – i.e. a query submitted against ESS1 may actually be processed by ESS2, and vice-versa.

Balancing Performance and Fault Tolerance

In the two previous configuration examples, either performance or fault tolerance was maximized, each at the expense of the other. Another design is possible, which strikes a balance between these two extremes. Starting with the fault tolerant design, rather than hosting a single physical index, each server can host a segmented index. As each server is a dual-processor server, it makes sense to segment each hosted index into two index segments.



Data Providers

An Overview of Extensible Search Server Data Providers.

Data providers are responsible for reading data from one or more data repositories, packaging that data as a series of nodes, and submitting those nodes to an index for storage.

Data providers run within one or more Extensible Search Servers; the Daxat Extensible Search Server 2005 ships with a number of data providers to handle common tasks, such as crawling web sites and extracting data from relational databases. Often, the data provider has to perform very little translation of the source data; massaging the data into an index-friendly node is usually performed by *node transformations*. For example, a data provider need not be concerned about how to convert a Microsoft Word document into plain-text; a node transformation can handle this task. Therefore, rather than submitting nodes directly to the index, a data provider will normally submit nodes to a *transformation chain* so that the raw data will be translated into an index-ready format. Search administrators can freely modify the transformation chain associated with each data provider instance.

Data providers need not be concerned with the physical architecture of the destination index – simple indexes and replica indexes both appear as the same abstract notion of an index. Nodes submitted by a data provider to a fault-tolerant index will automatically be stored in a fault tolerant manner; the data provider is not involved with the fault tolerant process.

When an instance of a data provider is created, the search system administrator may set any custom properties exposed by the data provider. For example, a web crawler needs to be given a list of ‘seed’ urls from which crawling is to begin.

Each data provider instance may also be associated with a simple or complex runtime schedule. For example, a data base data provider may be instructed to check for new records added to the database once every five minutes.

There are several ways to create custom data providers. The Daxat Extensible Search Server exposes a low-level data provider API, but this will generally be ignored in favor of subclassing high-level objects which have been prebuilt to handle most of the communications between the data provider and ESS. Furthermore, the data providers shipped with the Daxat Extensible

Search Server are available in source code form; these providers are invaluable training aids, and may be used as the basis for development of custom data providers.

Node Transformations

An Overview of Extensible Search Server Node Transformations.

When a data provider collects data from a data repository, often that data is collected in a raw, native format. Although a data provider is free to translate that data into an index-friendly format, it is often easier for a data provider to simply hand off the raw data to one or more *node transformations*, which will handle the data translation on behalf of the data provider.

Node transformations are reusable software components which, when handed a node, may freely modify that node before passing the node back. Node transformations are arranged into ordered *node transformation chains*, which, in turn, are associated with a data provider instance.

The Daxat Extensible Search Server ships with several node transformations for handling such common tasks as reading the contents of remote URIs, converting Microsoft Office and Adobe PDF documents into plain text and stripping HTML tags from HTML documents. Thanks to the extensibility model, it is easy to develop custom node transforms.

Queries

An Overview of Extensible Search Server Queries.

An index is useless if it cannot be effectively and efficiently queried against. The Daxat Extensible Search Engine has a comprehensive query parser and optimizer; extremely complex queries can be expressed with ease, and the results of those queries are computed quickly.

A *query* consists of three parts – a *target*, a *query expression* and *ordering criteria*. These elements describe against what to query, what to find and how to order the results.

Query Targets

The query target is simply the name of the index against which to run the query. The index can either be a simple index or a replica index.

Query Expressions

The query expression is what most people think about when they think about queries. Daxat Extensible Search Server query expressions are a combination of token, wildcard, Boolean, positional and occurrence expressions, with additional support for filtering based upon strongly-typed metadata values and defining aggregations.

By convention, operators will be spelled out in CAPITAL LETTERS in all query expression examples, but operators are case insensitive.

The simplest queries are made up of one or more words (technically, tokens, not words), lacking any explicit operators. The base case, a single term, matches all nodes whose searchable content contains the term. For example, the query expression `apple` matches all nodes whose searchable content contains the word `apple`. When a query lacks operators, the query terms are treated as a *conjunction*. For example, `apple pie` is equivalent to `apple AND pie`, and matches all nodes containing both the word `apple` and the word `pie`. The words `apple` and `pie` need not be adjacent to one another in the node, nor must `apple` exist in the node before `pie`. A node containing “He enjoyed the fresh **apple pie**,” would be a match, as would “**Apple** Computer is represented in the chart as the blue **pie** slice.”

Boolean operators are the most common way to construct more complex queries. The three Boolean operators are AND, OR and NOT. Like all operators, Boolean operators may be

combined with parenthesis to create complex, compound expressions. For example, (budget AND federal) OR (trade AND commerce) would match all nodes containing both budget and federal and also all nodes containing trade and commerce. Parentheses may be nested to arbitrary deep levels, as in ((tree AND (cut OR slice)) AND NOT apple) OR (NOT maple AND syrup).

Positional operators are used to find nodes where the ordering and relative distance of the query terms within the node is important. The most basic positional operator are double quotes, also know as the phrase operator. A phrase matches nodes containing the terms of the phrase in the same order as the phrase and within sequential order in the node. For example, the query expression "hot apple pie" only matches those nodes containing the exact phrase hot apple pie. The query expression compact car "automatic transmission" is equivalent to the query expression compact AND car AND "automatic transmission". Double quotes may also be used around single terms which would otherwise be interpreted as a query operator. For example, "and" OR "not" matches nodes containing either the words and or not.

The BEFORE and AFTER operators are used to find matches whereby the relative ordering of the two terms is important, but the distance between the two terms is not. For example, chicken BEFORE egg matches all nodes containing at least one instance of the word chicken before some instance of the word egg. This does not preclude the existence of the reverse, however. For example, a node containing "The **chicken** laid one **egg** every five minutes, making it the most productive **chicken** on the farm" would be a valid match, even though the single instance of egg precedes the second instance of chicken. In order to mandate the precise ordering, consider a query expression such as chicken BEFORE egg AND NOT egg BEFORE chicken. The sample node does not match against this query.

The NEAR operator matches all nodes where the two terms are within 10 positions of one another. For example, search NEAR server matches all nodes where the word search is within 10 word positions of the word server. Ordering is **not** important; the word server can appear up to 10 positions before or after the word search.

NEAR is a special case of the more robust WITHIN operator. The expression *expr1* WITHIN *n* WORDS OF *expr2* matches all nodes where some instance of expression *expr1* is within *n* word positions of some instance of the expression *expr2*. For example, the expression search NEAR server is equivalent to search WITHIN 10 WORDS OF server. As with NEAR, the ordering of the terms within the node is not important; only their relative distance matters.

When both ordering and distance are important, use the PRECEDES and SUCCEEDS operators. PRECEDES is used to find nodes where some term is positionally before another term, and those terms are either within a certain number of word positions of each other, or an exact number of word positions apart. The expression extensible PRECEDES server WITHIN 2 WORDS matches all nodes where an instance of the word extensible occurs before an

instance of the word `server` by zero, 1 or 2 word positions. "The Daxat **Extensible Search Server** is a power enterprise tool" matches this query expression. `PRECEDES BY` is used to mandate an exact number of word position between the two terms. For example, "Their search for an **extensible server** was now over" matches the previous query expression, but does not match `extensible PRECEDES server BY 2 WORDS` because `extensible` precedes `server` by only 1 word position.

A phrase expression can be written as a chain of `PRECEDES` operators. For example, the query expression "hot apple pie" is equivalent to `hot PRECEDES (apple PRECEDES pie BY 1 WORDS) BY 1 WORDS`.

Occurrence operators are sensitive to the number of occurrences of a term within a node. The `ATLEAST n expr` operator matches nodes which contain a minimum of *n* occurrences of the expression *expr*. Similarly, `ATMOST` matches nodes containing no more than a certain number of occurrences of the expression. For example, `blue Ford ATMOST 4 repair` would match all nodes containing the words `blue` and `Ford`, and having no more than 4 occurrences of the word `repair`.

Other, more esoteric query operators exist; the full set of query operators may be found in the reference materials.

Wildcards may be used to find matches against partial words. The wildcard `*` matches zero or more unspecified characters, whereas `?` matches exactly one unspecified character. For example, `the*` matches `the`, `there`, `their`, `them`, etc. whereas `the?` only matches 4-letter words such as `them` and `then`. Wildcards can appear at any point in a term, and any combination of wildcards is valid. For example, `un*v?*` would match any word starting with `un`, containing a `v` any number of characters after the `un`, with a least one letter following the `v`. Care must be taken when a query term starts with a leading wildcard – these searches can be substantially slower than all other searches. For example, the query expression `?th*` will likely take substantially longer to compute than `th*`.

Wildcards may also be used within phrases. For example, the phrase query expression `"dax* extens?ble sea??h ser*r"` would match nodes containing the phrase "Daxat Extensible Search Server."

The query expression `*` is a special case – that query expression will very quickly match all nodes within the index. For instance, it is a convenient way to quickly count the number of nodes within the index.

Query terms may be correlated when complex, related matches need to be found. **Query term correlation** is a special feature of the Daxat Extensible Search Server which not only distributes terms over query term operators, but also correlates the instances of those terms during query evaluation. Consider the following two queries:

1. (troubled NEAR sea) AND (tost NEAR sea)
2. (troubled AND tost) NEAR sea

At first blush, these queries may appear to be equivalent. Due to query term correlation, however, there is a subtle but importance difference between these two queries. Query #1 will match all nodes which contain an instance of the word `troubled` near any instance of the word `sea`, and also contain an instance of the word `tost` near any instance of the word `sea`. Query #2, on the other hand, will only match those nodes where both `troubled` and `tost` are near the **same** instance of the word `sea`. Although the difference is subtle, the power exposed by query term correlation can be profound. Query term correlation has many uses, such as matching proper names when it is unknown if the documents to be searched have consistently used all parts of the proper name.

Named Regions

Any query expression can be limited to a named region by preceeding the query expression with *NamedRegionName*::. For example, (apples OR oranges) AND author:(Sam or Sally) may be used to find all nodes containing either apples or oranges, with an author of either Sam or Sally.

Metadata Filtering Expressions

The Daxat Extensible Search Server query processor is capable of using node metadata at query time. This functionality can be used to search for nodes with metadata matching certain criteria, but, for performance reasons, is best used in conjunction with a traditional query expression.

Metadata query expressions are embedded within query expressions using the `<% ... %>` escape sequence. For example, the query expression `red convertible <% [Price] <= 9000 %>` may be used to find all nodes containing the words `red` and `convertible`, with metadata named `Price` and a value of less than 9000. Metadata query expressions are true expressions; they may be used anywhere a query expression may be used. For example,

```
((red convertible <% [Price] <= 9000 %>) OR (blue suv  
<% [Price] <= 14000 %>)) AND <% [ModelYear] >= 2001 %>
```

would match all nodes containing vehicles whose model year is 2001 or later and which are red convertibles for less than \$9,000 or SUVs for less than \$14,000.

Metadata query expressions also support run-time mathematical operators; see the reference materials for a complete description of all valid metadata query expressions.

Aggregates

As mentioned in 'Elements of a Node', search results may be aggregated by unique metadata values. Calculating an aggregation for a given query expression is straight-forward; simply append AGGREGATE to the query and then list the names of the metadata values to aggregate. For example, "automatic transmission" convertible AGGREGATE [CarMake], [CarColor] will match all nodes containing the phrase "automatic transmission" and the word convertible, and will also compute the unique values for both metadata values CarMake and CarColor. So whereas the search results will include references to every matching node, the CarMake aggregate will contain a small set of values, such as Honda, Ford and Audi, coupled with counts indicating how often each one of those values occurs within the result set.

Ordering Criteria

The third and final part of query is the optional ordering criteria. Ordering criteria is used to order the results in the result set. The criteria consists of a software component, called as *Scorer*, combined with parametric values for that component. Any number of Scorers may be used in combination to create a multilevel sort. For example, search results could be primarily sorted by date, and then sub-sorted by relevancy.

Query Processing: Words vs. Tokens

The previous examples appeared to be searching on words. As discussed in the Nodes chapter, there is a difference between a word and a token. **Query expressions match tokens, not words.**

The major impact of this fact is that, by default, **searching is partially case-sensitive**. Consider a node created from the words "This is a test." Under normal circumstances, this will result in a node containing 5 tokens: This this is a test. A search for either This or this will match the node. However, a search for Test will not match the node, as the token Test does not exist in the node. A search for test will match the node. Likewise, a search for THIS will not match the node.

In general, if a search term is all lower case, it will match all instances of the same word. If a search term contains any upper-case characters, it will only match those tokens indexed with the same case. This feature can be powerful when searching for proper names which are also common nouns. For example, a search for gates will match both Gates and gates, whereas a search for Gates will only match Gates. This can help to limit the search results to the person Gates, rather than including content discussing "a structure that can be swung, drawn, or lowered to block an entrance or a passageway."

If the desire is to have case-insensitive searching, the easiest way to achieve this goal is to convert all queries to lower-case before submitting them to the Daxat Extensible Search Server. Moreover, if the search solution will only support case-insensitive searching, then a smaller index can be created by converting all of the searchable content to lower-case before indexing. This will result in a smaller index due to the fact that only one token will be created for each word.

Searching for Tokens

As explained in the Nodes chapter, although tokens may be comprised of any sequence of characters, including white space and non-alphanumeric characters, under normal circumstances such tokens are not created within the searchable content of a node. If such tokens are created, however, it is possible to search for them. The **token operator** { } can be used much like the double quotes of the phrase operator. Everything between the two curly brackets will be interpreted as a single token. For example, the query expression {a weird token} will match those nodes containing the single token a weird token, including the white space. This can be useful for searching for content with non-alphanumeric characters, such as "tips and tricks" `ProgrammingLanguage:({c++} OR {c#})`.

Sample Search Solution Walk-Through

An Example of a Complete Search Solution.

The Overview section of this manual concludes with a step-by-step walk-through of the creation of a complete search solution. This extended example will create a search solution for the pages of a web site.

Selecting and Preparing a Data Repository

In this sample, we will be indexing all of the documents found on a web site. Within this sample, the web site will be referenced as www.example.com, but any web site may be used.



About Web Crawling

The web crawler which ships with the Daxat Extensible Search Server is powerful and efficient. It is capable of crawling all of the pages on a web site, and all pages references by that web site, at very high speed. **It is imperative that you use caution when defining which web site you want to crawl.**

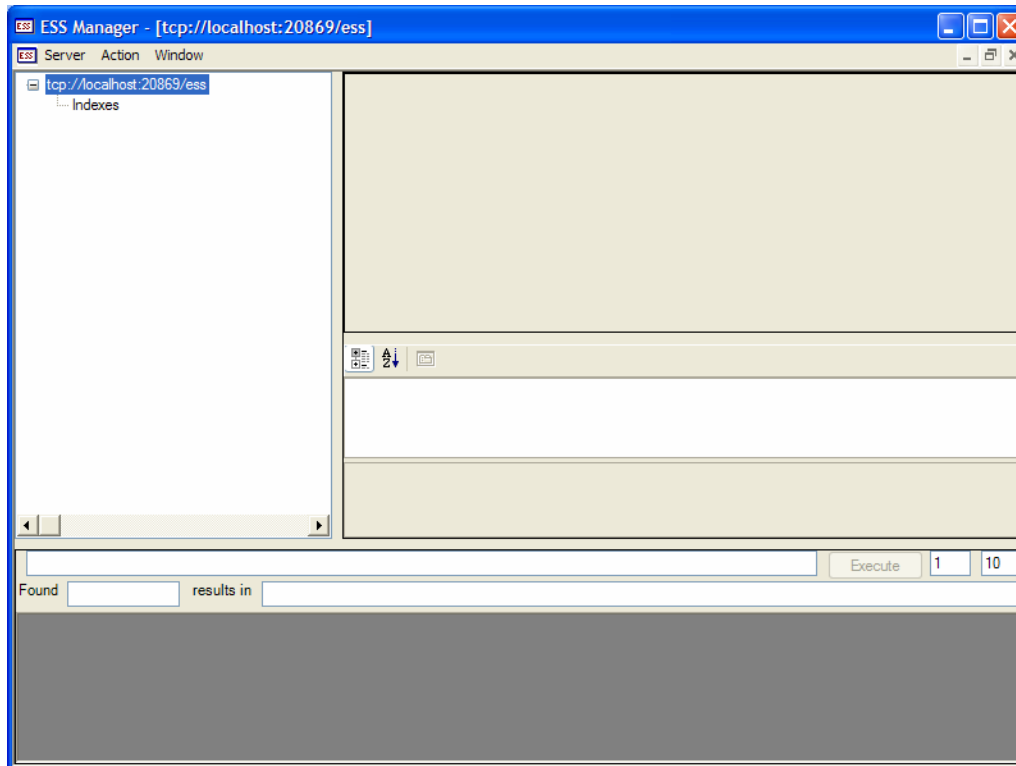
Failure to do so could result in the web crawler placing an extremely high load on the target web server. The configuration instructions that follow will contain explicit warnings when you must exhibit care.

In this sample, the target web site will be crawled and indexed, but all external links off of the web site will not be followed. There is no explicit preparation required on the web server to permit web crawling; however, the ESS web crawler will respect the web server's `robots.txt` file. If the crawler is prohibited from crawling the target web site, the index will remain empty.

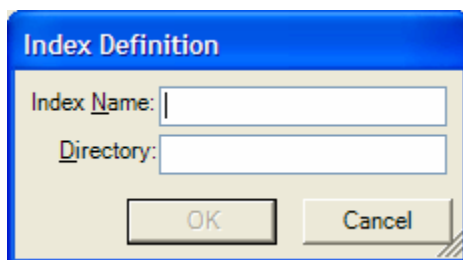
Creating a new Simple Index

Begin by starting the ESS Manager. From the Server menu, select Connect... and connect to the Extensible Search Server you wish to manage. If running the manager on the server itself, in the default configuration, the connection string is `tcp://localhost:20869/ess`.

Assuming the managed server is a new installation, the manager will be sparse:



Highlight the Indexes node in the upper-left tree control, and then select “New simple index...” from the Action menu. The Index Definition dialog box will appear.



Recall that a simple index has two defining properties – a unique name and a storage location. Set the index name to `WebSiteIndex` and the directory to any suitable disk location, such as `c:\Indexes\WebSiteIndex`. **This path will be resolved by the server process, not by the ESS Manager.** In other words, if you are administering a remote Extensible Search Server, the `c:\` drive refers to the `c:\` drive on the server, not where you are running the ESS Manager. Moreover, **the Extensible Search Server must have full access rights to the directory location.** If any part of the directory does not exist, it will be created.

One way to handle these issues, especially if the Extensible Search Server is running as `LOCAL SERVICE`, is to first create the `c:\Indexes` directory and grant `LOCAL SERVICE`

full access rights to that directory. Once that is complete, the Extensible Search Server will be able to create index directories underneath the `c:\Indexes` directory without incident.

Once the values have been set in the Index Definition dialog box, click OK.

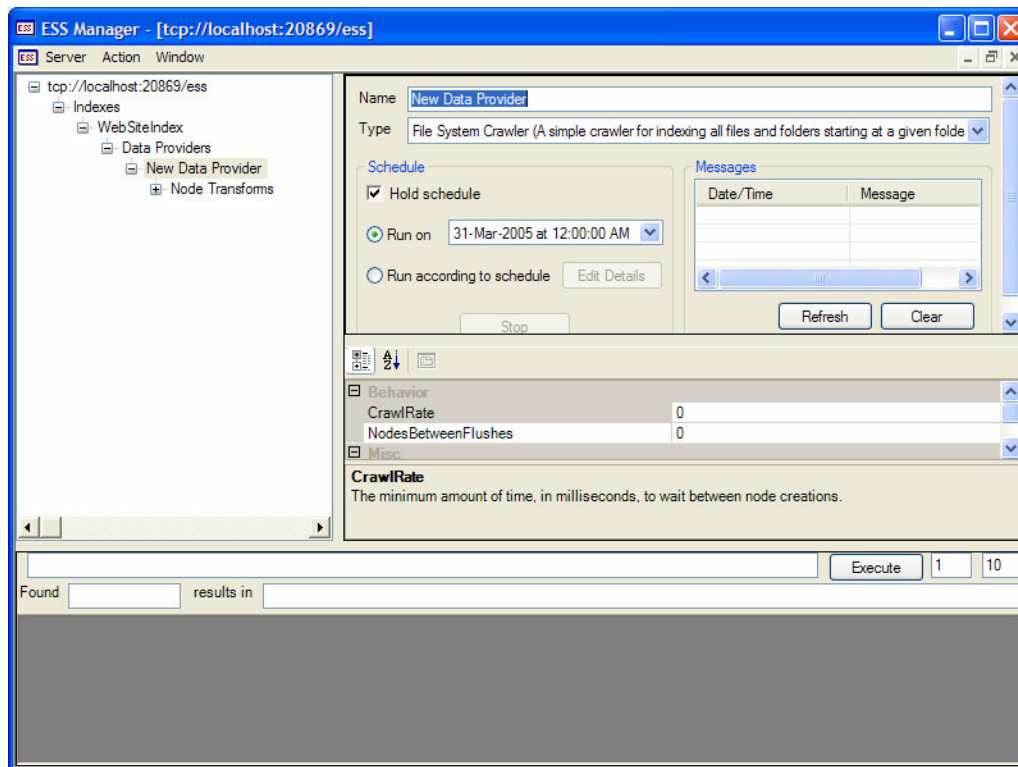
Immediately the Extensible Search Server will create a new, empty index at the specified location. **If an index already exists at the specified location, it will be ‘attached’ to the server and ready for use.** In other words, this same procedure may be used to open existing indexes, provided the index to attach is not current in use.

A `WebSiteIndex` node will now be available under the `Indexes` node in the ESS Manager. If you highlight the `WebSiteIndex` node, you will be able to submit queries to the index using the query pane located at the bottom of the ESS Manager. However, as the index is currently empty, no query will return any results.

Adding the Web Crawler as a Data Provider

In order to populate the index with some nodes, it is necessary to instantiate a data provider and start it running. Since we want to crawl our web site, we will create an instance of the web crawler.

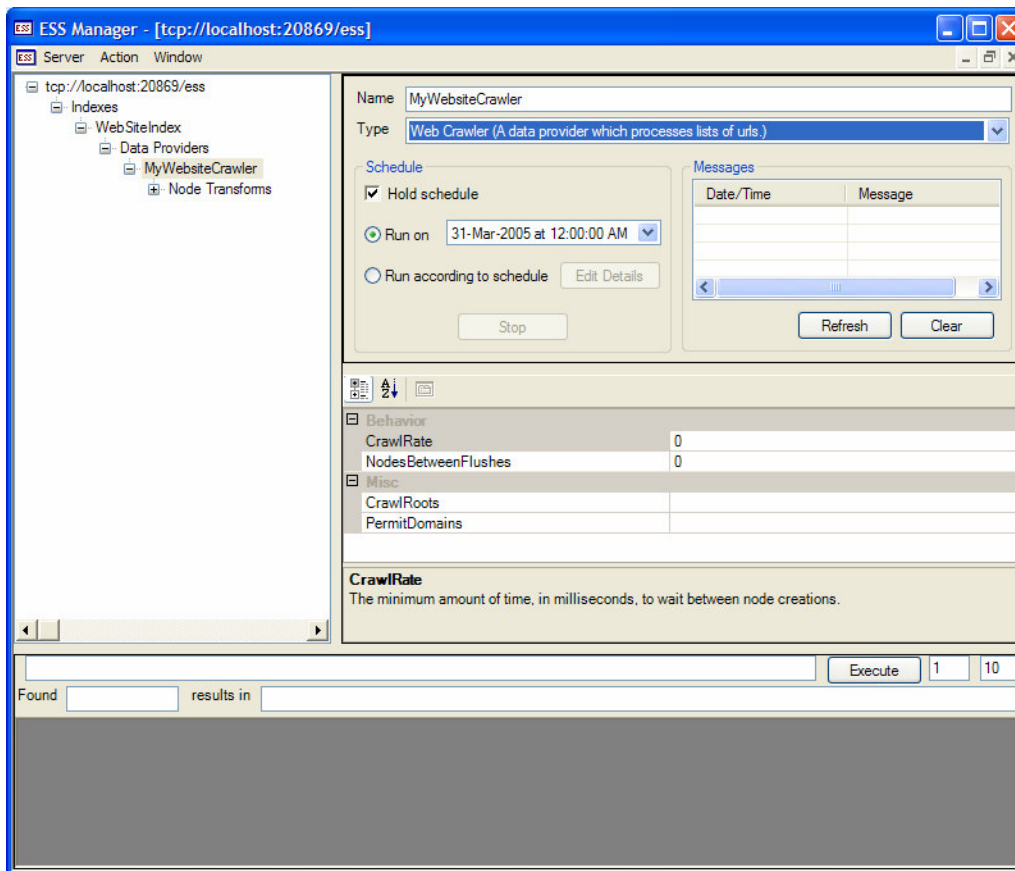
Under the `WebSiteIndex` node you will find a `Data Providers` node. Highlight the `Data Providers` node and select `Add Data Provider` from the Action menu. Almost immediately a new data provider instance named `New Data Provider` will be created and associated with the index.



At this point it may be necessary to expand the ESS Manager window from its default size. Notice that the property pane on the right hand side of the manager is subdivide into two smaller panes. The upper pane is known as the Definition Pane and the lower pane is known as the Custom Properties Pane. You may adjust the splitter between these two panes to fully expose the Definition Pane.

The Definition Pane contains all of the common properties and controls related to the type of item selected in the left hand side tree control. Since a data provider is currently selected, the Definition Pane contains the properties and controls common to all data providers.


Change the name of the data provider to something meaningful, such as MyWebsiteCrawler, and change the type of the data provider to Web Crawler. The manager should now look similar to this:

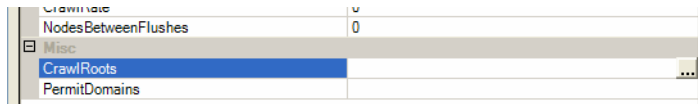


The web crawler cannot run without some extra information; therefore, it is necessary to set some values in the Custom Properties Pane.

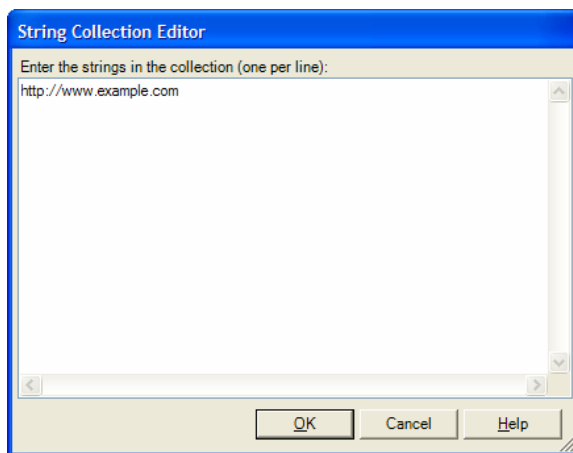
Configuring the Web Crawler

The options for the web crawler are completely described in the Administration section of this guide; these are the settings necessary to complete this sample.

The CrawlRoots property is a list of urls from which the web crawler should start crawling. If you click on the CrawlRoots property, you will not be able to directly enter a value; this is due to the fact that this property takes a list of values. Click on the  located at the end of the value entry box



An editor will pop-up into which you can enter the list of starting urls. Since we simply want to crawl all content available starting at <http://www.example.com>, just enter that one url into the editor and click OK.



We want to restrict the crawler to just the example.com domain; following a similar procedure, set the PermitDomains to the single value of .example.com. (Note the leading dot.)



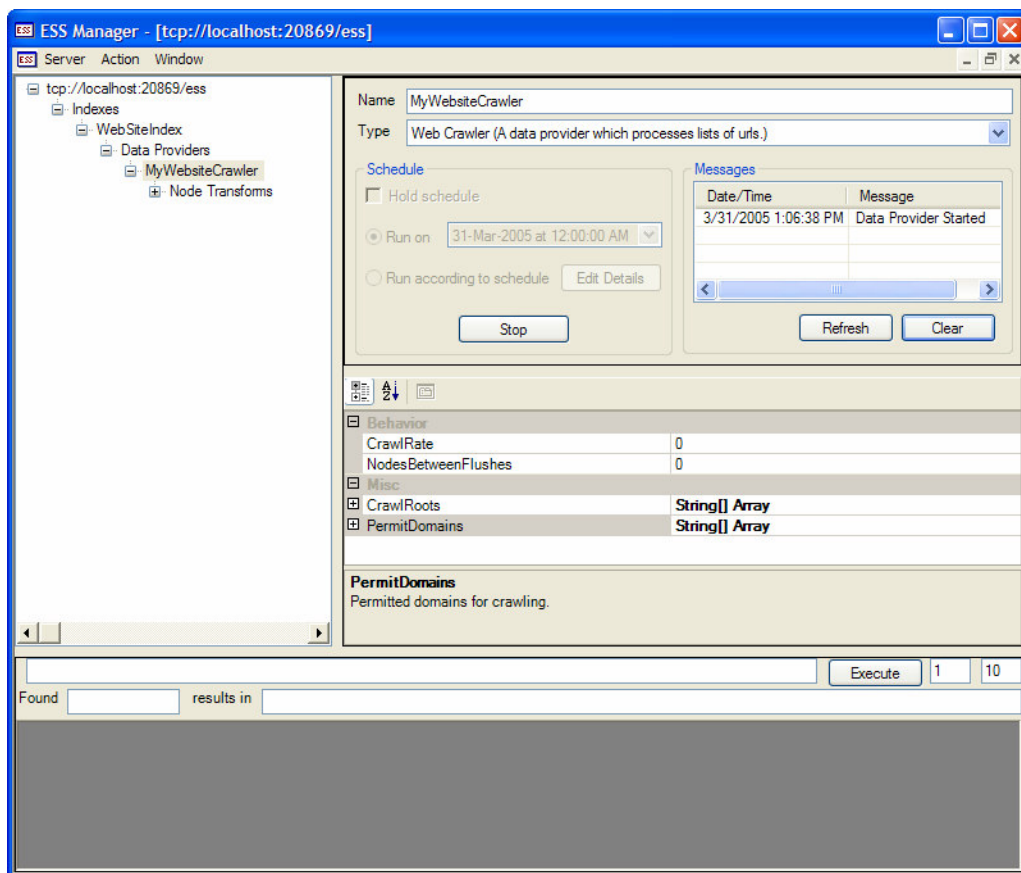
Be Careful!

If you elect to crawl a website different than www.example.com, be aware that the crawler is going to aggressively attack the target website in the default configuration. The CrawlRate property may be used to slow down the crawler – this value is the amount of time, in milliseconds, between each hit on the target web server. For example, a value of 2000 will not pull documents from the web server any faster than 1 document every 2 seconds. Also, due to the complex caching mechanisms utilized by the Daxat Extensible Search Server, it may take a substantial amount of time before a crawl is started and nodes are searchable within the index. By setting the NodesBetweenFlushes to a low value (e.g. 30), you can ensure that, soon after crawling begins, nodes will be searchable within the index. This is not necessary if the crawl will complete in a short period of time; regardless of caching, as soon as a data provider finishes running, all nodes added to the index by that data provider are available for searching.

Starting the Web Crawler

In the Definition Pane is an area labeled Schedule. These controls permit the creation of both simple and complex runtime schedules for the data provider. The Hold schedule checkbox is checked, indicating that the currently assigned schedule is ready to be run. The default schedule will start the data provider immediately, run it to completion, and then place the schedule back on hold. In other words, it will run the data provider once.

Uncheck the Hold schedule checkbox to start the web crawler running. The messages window will show that the data provider has started.

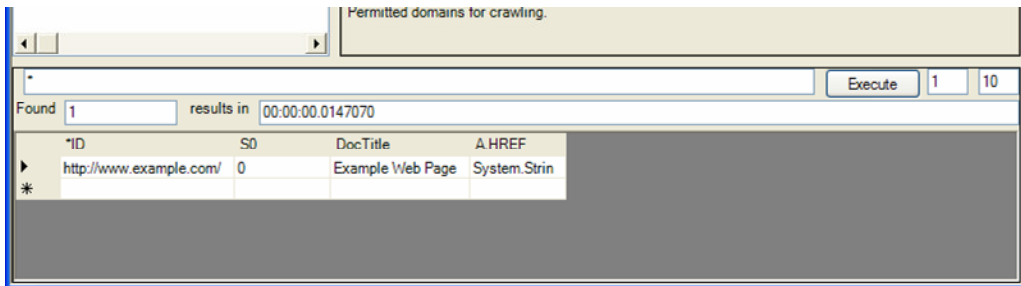


The data provider will run to completion; the amount of time this will take is highly dependent upon the complexity of the crawl and the available bandwidth between the Extensible Search Server and the target web server. The simplest crawl (1 web page) will take up to 30 seconds due to some internal timing and synchronization mechanisms of the web crawler. When the crawl is complete, a message will be posted to the messages list in the Definitions pane.

Querying the Index

Assuming the crawl ran successfully, there should be at least one node stored within the index. In the Query pane type the query expression `*` and click Execute. This query will return a reference to every node in the index, however the user interface, by default, will only show the first ten

results. If you crawled www.example.com, which only contains a single web page, the results should be similar to this



Congratulations! You just built your first complete search solution using the Daxat Extensible Search Server.

II. Development

Client-Side Development

How to Develop Clients for the Daxat Extensible Search Server.

Every instance of the Daxat Extensible Search Server exposes a client-side programming interface via standard .NET Remoting. The inclusion of an extremely light-weight assembly in a .NET-based application makes it extremely easy for that application to interact with any reachable Daxat Extensible Search Server.

Referencing the Client Side Library

The client-side accessible functionality of the Daxat Extensible Search Server is defined in the assembly `Daxat.Ess.ClientLibrary`, found in `Daxat.Ess.ClientLibrary.dll`. By default, this DLL is found in `C:\Program Files\Daxat\Extensible Search Server 2005`. Adding a reference to this assembly to a .NET application makes the client-side types of the `Daxat.Ess` namespace available to the application. The vast majority of the types defined in `Daxat.Ess.ClientLibrary` are interfaces; very little implementation code is included within the assembly. As such, the assembly is extremely light-weight.

Connecting to a Server

Before any search-related work can be performed, it is necessary to connect to an available Daxat Extensible Search Server. Extensible Search Servers are exposed to clients via .NET Remoting. A server may listen on practically any valid .NET Remoting channel, but, by default, Extensible Search Servers listen on the `tcp` channel over port 20869. The default address for an Extensible Search Server is `tcp://hostname:20869/ess`.

The majority of the server's client functionality is exposed via the `Daxat.Ess.Server.IServer` interface. Connecting to an Extensible Search Server simply involves creating a proxy for the server-side implementation of `IServer`, as show below.

```
using System;
using Daxat.Ess.Server;

public class ConnectToServerTest
{
    public static void Main()
    {
        // Connect to the ESS instance on this machine
        IServer ess = (IServer)Activator.GetObject(
            typeof(IServer),
            "tcp://localhost:20869/ess");

        // Disconnect
        ess = null;
    }
}
```

Executing a Query

`IServer` is a composite interface, aggregating several other interfaces. A query expression cannot be submitted directly to `IServer`; instead, a query must be executed against `IServerQuery`. (Alternatively, a query may also be executed directly against a reference to an index object.) Any instance of `IServer` can simply be recast as `IServerQuery`.

The heavily overloaded `IServerQuery.ExecuteQuery()` method performs the actually query processing, returning a `Daxat.Ess.Results.QueryResponse` object containing the results of the query. The `QueryResponse` object contains some simple statistics regarding the query (e.g. the total execution time), plus an array of `Daxat.Ess.Results.SearchResult`, with each element of the array representing a unique search result. If the search results are requested to be ordered (e.g. in relevancy order), then the array will be returned in that order.

The following complete client example shows how to connect to an ESS instance running on the same machine as the client, grab a reference to the `SampleIndex` index within the server, execute a search for apples OR oranges, and display the search results with some simple statistics.

```
using System;
using Daxat.Ess.Server;
using Daxat.Ess.Results;

public class SimpleQueryExample
{
    public static void Main()
    {
        // Connect to the ESS instance on this machine
        IServer ess = (IServer)Activator.GetObject(
            typeof(IServer),
            "tcp://localhost:20869/ess");
        // Query the 'SampleIndex' index for 'apples OR oranges'
        QueryResponse qr =
            ((IServerQuery)ess).ExecuteQuery(
                "SampleIndex",
                "apples OR oranges");
        // Display the total number of results found
        // and the query execution time
        Console.WriteLine("Found {0} results in {1} seconds.",
            qr.TotalNumberOfMatches,
            qr.QueryExecutionDuration.TotalSeconds);
        // Display the node id and title of each search result.
        for (long i=0; i < qr.SearchResults.LongLength; i++)
            Console.WriteLine("Result #{0}. Node ID {1}, Title {2}",
                i+1,
                qr.SearchResults[i].Metadata.ID,
                qr.SearchResults[i].Metadata["DocTitle"]);
    }
}
```

Sorting Results

The Daxat Extensible Search Server is capable of efficiently ordering the search results according to virtually any criteria. When the result set is computed, each individual result can be assigned an ordered set of strongly-typed scores. The only requirement for the type of a score is that it implements the `System.IComparable` interface. There is no limit to how many scores may be assigned to a search result, provided all search results have the same number of scores and the *n*th score of any result may be compared to the *n*th score of any other result. When more than one score is assigned to a search result, the scores are compared in order against the scores of the other results to determine which search result is ranked higher. For example, consider a set of search results scored by document publication date (ignoring time of day) in descending order and then number of pages in ascending order. The search results will be ordered in reverse chronological order, but documents with fewer pages will be listed before documents with more pages published on the same day.

Document Id	Publication Date	# of Pages
627182	5-Apr-2005	9
621717	5-Apr-2005	23
555362	3-Apr-2005	46
555312	3-Apr-2005	97

Nodes are scored by **Scorers** -- .NET types which implement the `Daxat.Ess.Scoring.IScorer` interface. A scorer has access to the positional data of the tokens comprising the node, the metadata of each search result, statistics about the index against which the query was executed, runtime information passed to the algorithm via `ExecuteQuery()` and potentially other external information; all of this information may be used to compute a score for each search result. In the simplest case, an element of the strongly typed metadata of each node – for example, the `PublicationDate` – can be used to ‘compute’ the score (in this case, the date is simply copied out of the metadata and assigned as the score). In more advanced cases, such as relevancy computation, statistical information about the frequency of the tokens found in each search result versus their occurrences within the index, may be brought to bare when computing the search result’s score.

The Daxat Extensible Search Server ships with a powerful relevancy computation algorithm, implemented as the scorer type `Daxat.Ess.Scoring.Relevancy.AdvancedRelevancyScorer`. This scorer, fully documented in the reference sections of this guide, attempts to give higher scores to those search results which appear to be the most relevant. This scorer demonstrates the ability of a scoring algorithm to make use of complex information, including runtime information supplied via `ExecuteQuery()`.

Search result sorting is accomplished by passing a list or an array of `Daxat.Ess.Scoring.ScorerDefinition` objects to an appropriate overload of `ExecuteQuery()`. An *n*-level sort requires a list or an array of *n* `ScorerDefinition` objects. A `ScorerDefinition` is a combination of a scorer type name (following the conventions of `System.Type.GetType(string)`), optional runtime data to be passed to the scorer, and a sort direction (ascending or descending). In the following example, a single-level sort is performed, whereby each search result is scored for relevancy. Runtime relevancy criteria is included which specifies that search results containing query terms within the `title` named region should be considered more relevant than search results containing the search terms elsewhere within the searchable content of the node.

```

using System;
using Daxat.Ess.Server;
using Daxat.Ess.Indexes;
using Daxat.Ess.Results;
using Daxat.Ess.Scoring;
using Daxat.Ess.Scoring.Relevancy;

public class RelevancyQueryExample
{
    public static void Main()
    {
        // Connect to the ESS instance on this machine
        IServer ess = (IServer)Activator.GetObject(
            typeof(IServer),
            "tcp://localhost:20869/ess");
        // Get a proxy to the index named SampleIndex
        IIndex index = ess.GetIndex("SampleIndex");
        // Criteria for relevancy - base relevancy score off of terms
        // used in the query; give double weight to terms found
        // within the 'title' named region
        RelevancyCriteria rc = new RelevancyCriteria(
            null, true,
            new TermWeightPair[] { new TermWeightPair("title", 2.0) },
            true);
        // Create a ScorerDefinition which scores with the
        // advanced relevancy scorer using the above criteria.
        ScorerDefinition sd = new ScorerDefinition(
            "Daxat.Ess.Scoring.Relevancy.AdvancedRelevancyScorer, " +
            "Daxat.Ess.Scoring.Relevancy.AdvancedRelevancyScorer",
            rc, SortOrder.Descending);
        // Query the index for 'chicken AND recipe AND NOT fish' and
        // sort by the ScorerDefinitions
        QueryResponse qr =
            index.ExecuteQuery("chicken AND recipe AND NOT fish", sd);
        // Display the total number of results found
        // and the query execution time
        Console.WriteLine("Found {0} results in {1} seconds.",
            qr.TotalNumberOfMatches,
            qr.QueryExecutionDuration.TotalSeconds);
        // Display the title and relevancy score of ea. search result.
        for (long i=0; i < qr.SearchResults.LongLength; i++)
            Console.WriteLine("Result #{0}. Title {1}, Relevancy {2}",
                i+1,
                qr.SearchResults[i].Metadata["DocTitle"],
                searchResult.Score);
    }
}

```

The previous example also demonstrated calling the `ExecuteQuery()` method of the `Daxat.Ess.Indexes.IIndex` interface, rather than calling `IServerQuery.ExecuteQuery()`. `IIndex` is the common interface for all indexes, regardless of type (simple or replica), and defines various methods for directly manipulating and interacting with indexes.

Improving Performance through Pagination

Most users are accustomed to a pagination-based user interface for displaying search results. This is particularly important if a user's query will often result in hundreds, if not thousands or more, of matching search results. A pagination-based user interface typically displays 10 search results per page, with a navigation mechanism to move back and forth through the result set.

Creating a pagination-based user interface requires a means for determining which search results should be displayed on a particular page. There are two obvious ways to accomplish this goal. The first method involves returning all matching search results to a pagination cache, and then driving the user interface off of that cache. At first blush, this may appear to be a sound design – for any given query, the Daxat Extensible Search Server performs a single search; as the user pages through the data set, the displayed search results are pulled from the pagination cache. In reality, there are two potential problems with this strategy. First, the execution time a single query is directly related to the number of search results returned for that query. If the user's query returns thousands or millions of search results, all of those search results must be pulled into the pagination cache. The query execution time can potentially be long. Moreover, if the search results are sorted, all of the results must be sorted, placing an additional execution burden on the server. Second, the pagination cache could easily grow quite large. If the pagination cache lives on a shared resource, such as a web server, relatively few users could quickly consume the resources of the server. Worse still, if the user's connection to the interface is stateless, as it is with a web server, then there is no easy way for the interface to determine when the cached data is no longer needed.

An alternative strategy which solves these problems is to use the pagination features of the `ExecuteQuery()` method. When performing a query, it is possible to request search results starting at position i , where 0 is the first result, and to request no more than n total results. For example, $i=0$ and $n=10$ would return the first 10 search results.

Using this feature has several advantages. First, performance is increased because no more than n search results are returned; the time it takes to execute a query is directly proportional to n . Second, if the search results are sorted, the sorting algorithm has less work to do. **The ordering of sorted results when using server-side pagination is the same as when pagination is not used.** The key to understanding why the sorting performance is increased is to realize that once the position of a sorted search result is determined to be outside of the range of $i \dots (i+n-1)$, it is not necessary to compute the precise ordered position of that search result. The performance of this specialized sorting algorithm is $O(im)$, where m is the total number of matching search results. As i is normally a constant factor, the overall performance of the sort is linear.

Several overloads of the `ExecuteQuery()` method accept `startAt` and `atMost` parameters, which directly relate to i and n , respectively.

The following code snippet extends the previous example by including pagination information.

```

...
int resultsPerPage = 10;
int pageToDisplay = 0; // The first page is page #0
int startAt = pageToDisplay * resultsPerPage;
// Query the index for 'chicken AND recipe AND NOT fish',
// limit the results to the current page,
// and sort by the ScorerDefinitions
QueryResponse qr =
    index.ExecuteQuery(
        "chicken AND recipe AND NOT fish",
        startAt,
        resultsPerPage,
        sd);
// Display the total number of results found
// and the query execution time
Console.WriteLine("Found {0} results in {1} seconds.",
    qr.TotalNumberOfMatches,
    qr.QueryExecutionDuration.TotalSeconds);
// Display the title and relevancy score of ea. search result.
for (long i=0; i < qr.SearchResults.Result; i++)
    Console.WriteLine("Result #{0}. Title {1}, Relevancy {2}",
        startAt+i+1,
        qr.SearchResults[i].Metadata["DocTitle"],
        searchResult.Score);
}
}

```

Handling Query Results – The QueryResponse Object

The `ExecuteQuery` method returns an object of type `Daxat.Ess.Results.QueryResponse`. A `QueryResponse` object contains not only the matches to the query, but also various piece of information regarding the result set and its processing.

The TotalNumberOfMatches Property

The `TotalNumberOfMatches` property is a `Long` value which indicates how many nodes matched the query **regardless of pagination**. For example, a query which matches 1173 nodes but, due to pagination, only returns the first ten results, will produce a `QueryResponse.TotalNumberOfMatches` of 1173.

The QueryExecutionDuration Property

The `QueryExecutionDuration` property is a `TimeSpan` value indicating the total time spent processing the query. Time spent for network transmission between the client and the server is not included in this time computation, but this can easily be computed directly on the client. If the queried index is distributed, the time span will include the time to handle the network communication between the involved Extensible Search Servers.

The SearchResults Property

The `SearchResults` property is an array of `Daxat.Ess.Results.SearchResult`; there is one element in this array for every result returned, respecting pagination. When pagination is used, it is entirely possible for the number of elements in this array to be substantially smaller than the value of `TotalNumberOfMatches`. A query which matches 3654 nodes yet, via pagination, is limited to results 30 through 39 will result in a `TotalNumberOfMatches` of 3654 and a `SearchResults` array of 10 elements.

Each `SearchResult` object represents a single matching node from the index. There are two general interest properties contained within each search result.

The SearchResult.Score Property

The `Score` property is an array of `System.IComparable`. The number of elements in this array corresponds to the number of `Scorers` passed to `ExecuteQuery`; each element of `Score` represents the score assigned by the corresponding `Scorer`. For example, if `ExecuteQuery` is called with a single relevancy scorer, then each `SearchResult` will contain a one-element `Score` property, and the value of `Score[0]` will be the relevancy score assigned to the search result (typically a `Double` value).

The SearchResult.Metadata Property

The `Metadata` property is a `Daxat.Ess.Results.Metadata` object and exactly matches the metadata assigned to the corresponding node. This property may be used to extract any of the strongly-typed data associated with the node. For example, it is common for nodes representing documents to have the document title stored as a `String` with a key of `"DocTitle"`. The title can be extracted for the matching node with a `String`-based `DocTitle` as `searchResult.Metadata["DocTitle"]`.

The Aggregates Property

The `Aggregates` property is a `System.Collections.Hashtable` object which maps `Daxat.Ess.Results.AggregateKeys` to a hashtable which maps unique values found for the aggregate to `Long` occurrences of the unique aggregate value. The value of this property may be null if no aggregates were requested (i.e. the query did not contain an `AGGREGATE` clause).

Fully understanding aggregates requires a complete understanding of node-associated metadata, as aggregates are computed from metadata. There are three major types of valid metadata values: basic built-in types (e.g. `Boolean`, `Int32`, `String`), arrays of basic types (e.g. `Boolean[]`, `Int32[]`, `String[]`) and `Hashtables`.

Aggregating basic built-in types is straight forward. Whenever an aggregated key for a search result's metadata contains a basic built-in type, the value is *collected into the aggregate*. If the value does not currently exist in the aggregate, then it is stored in the aggregate with a corresponding count of 1. If it currently does exist in the aggregate, then the aggregate count is increased by 1. This collection is maintained as a hashtable.

For example, consider the following metadata associated with a set of search results:

Result #	SearchResult.Metadata["Color"]
1	Black
2	Blue
3	Red
4	Red
5	Black
6	Red

If the query contains an `AGGREGATE [Color]` clause then the `QueryResponse.Aggregates` property will be a non-null hashtable and `QueryResponse.Aggregates[new AggregateKey("Color")]` will be a hashtable containing the following data:

Key	Value
Black	2
Blue	1
Red	3

In other words, within the result set, for the metadata named `Color`, there were 3 unique values assigned. The value `Black` occurs 2 times in the result set, the value `Blue` occurs 1 time and the value `Red` occurs 3 times.

Aggregating arrays of basic built-in types behaves similarly. The primary difference is that rather having a single value assigned to a given metadata key for a search result, the metadata may be assigned multiple values. When collected, each of these values is counted as an individual occurrence.

For example, consider the following metadata associated with a set of search results, which has an array of integers assigned to each instance of `ChildrenAges`:

Result #	SearchResult.Metadata["ChildrenAges"]
1	{ 9, 4, 2 }
2	{ 18, 17, 12 }
3	{ 15, 9 }
4	{ }
5	{ 18, 15, 2 }
6	{ 5, 4, 2 }

If the query contains an `AGGREGATE [ChildrenAges]` clause then `QueryResponse.Aggregates[new AggregateKey("ChildrenAges")]` will be a hashtable containing the following data:

Key	Value
2	3
4	2
5	1
9	2
12	1
15	2
17	1
18	2

In other words, within the result set, for the metadata named `ChildrenAges`, there were 8 unique values assigned. The values 5, 12 and 17 each occur once, the values 4, 9, 15 and 18 occur twice, and the value 2 occurs three times.

If the metadata contains a Hashtable, then there are two different ways to aggregate the values. If the hashtable is aggregated as a whole, then each individual value contained within the hashtable is collected into the aggregate; this is not unlike treating the values of the hashtable as an array and collecting the array.

For example, consider an index of nodes whereby each node represents an item in a catalog, and the metadata named `Certs` contains a hashtable mapping the name of a country to an array of

the names of government certifications for which that item has passed in that country. A query is executed which results in:

Result #	SearchResult.Metadata["Certs"]
1	France → { ige, adf } UK → { reu, nswm } USA → { iso, crp, ansi }
2	Germany → { gdf } UK → { nswm, llcs }
3	France → { ige, adf } USA → { iso, ansi, ul }
4	France → { ige } Germany → { hdo } USA → { crp }
5	USA → { iso, ansi }
6	UK → { nswm } Germany → { gdf }

If the query contains an `AGGREGATE [Certs]` clause then `QueryResponse.Aggregates[new AggregateKey("Certs")]` will be a hashtable containing the following data:

Key	Value
iso	3
crp	2
ansi	3
ul	1
ige	3
adf	2
reu	1
nswm	3
gdf	2
hdo	1
llcs	1

In other words, within the result set, for the metadata hashtable named `Certs`, there were 11 unique certifications across all countries. Of the 6 matching items, three of the items have ‘iso’ certification, two have ‘crp’ certification, three have ‘ansi’ certification, one has ‘ul’ certification, three have ‘ige’ certification, two have ‘adf’ certification, etc.

Alternatively, the aggregation may be performed against an individual key of the hashtable. The `Aggregates` property, which is keyed by `AggregateKey` values, will be keyed by 2-level `AggregateKeys`. The first level of the key indicates the key into the metadata, and the second level of the key indicates the key into the contained hashtable. Aggregation is performed upon the values found in the hashtable in the exact same manner as described previously for non-hashtable bound data; therefore, the values of the hashtable are restricted to the basic built-in types or arrays of those types.

Using the same example data and result set as before, the aggregation can be restricted to the USA certifications by using an `AGGREGATE [Certs.USA]` clause; `QueryResponse.Aggregates[new AggregateKey("Certs", "USA")]` will be a hashtable containing the following data:

Key	Value
iso	3
crp	2
ansi	3
ul	1

In other words, within the result set, for the metadata hashtable named `Certs` and the subkey `USA`, there were 4 unique certifications. Of the 6 matching items, three of the items have ‘iso’ certification, two have ‘crp’ certification, three have ‘ansi’ certification and one has ‘ul’ certification.

Deleting Nodes

Just as easily as nodes can be found within an index, they may be deleted from the index. To delete one or more nodes via a list of node ids, call the `IIndex.DeleteNodesByNodeIds` method. Alternatively, to delete all nodes which match a given arbitrary query, use `IQueryable.DeleteNodesSatisfyingQuery`. For example, this sample will delete all nodes contained within the index whose status named `region` contains `rejected`:

```
public void DeleteRejections(IIndex index)
{
    // Query the index for nodes with rejected status and
    // delete any matching nodes
    QueryResponse qr = index.DeleteNodesSatisfyingQuery("status:rejected");
    // Display the total number of nodes deleted
    // and the query execution time
    Console.WriteLine("Deleted {0} nodes in {1} seconds.",
        qr.TotalNumberOfMatches,
        qr.QueryExecutionDuration.TotalSeconds);
    // Display the node id of each deleted node
    for (long i=0; i < qr.SearchResults.LongLength; i++)
        Console.WriteLine("Deletion #{0}. Id {1}",
            i+1,
            qr.SearchResults[i].Metadata.ID);
}
```

Updating Metadata in Real Time

Often, data is not static. This can be particularly true about the metadata associated with a node. Consider for example an e-commerce search solution in which the nodes in an index represent items for sale and whereby the relevancy computation of a node considers the number of times that item has been purchased. In some circumstances, generating a new index nightly is satisfactory; the intraday selling of any one particular item is not important enough to warrant

updating the index in real time. In other circumstances, the intraday selling of an individual item may be very important, especially if the items are time sensitive, such as when new concert tickets go on sale, or the interest of an item may be directly related to randomly occurring events, such as the surge in sales of snow shovels during an unexpected snow storm. In the latter example, when a user searches for 'shovel' against an index containing many different types of shovels, it would be beneficial to both the consumer and the seller if snow shovels quickly assumed the top spot in the search results.

This desired result is easy to create with the Daxat Extensible Search Server. There are two major parts to the solution. First, the scores assigned to nodes at query time must be sensitive to the current number of sales of the item for which a given node represents. If the current daily sales for an item are stored within the node's metadata, this is easy to achieve either via `Daxat.Ess.Scoring.SimpleScorer` or a custom `IScorer` implementation. Second, the metadata for a given item's node must be updated each time one of those items is sold.

The `IQueryable.UpdateMetadata` method takes a node id and an instance of a `Metadata` object, and updates the index in-place and in real time with the new metadata. For example, the following method could be called every time a sale of an item is completed:

```
public void IncreaseSaleCount(IIndex itemIndex, string itemSku,
                             Metadata itemMetadata)
{
    itemMetadata["NumberSoldToday"] =
        (long)(itemMetadata["NumberSoldToday"]) + 1;
    itemIndex.UpdateMetadata(itemSku, itemMetadata);
}
```

Basic Data Provider Development

How to Develop Basic Data Providers for the Daxat Extensible Search Server.

Data providers are software components which interface between a data repository and the Daxat Extensible Search Server. Data providers are responsible for pulling data out of the data repository and creating a series of nodes representative of that data.

Referencing the ESS Libraries

As data providers execute within the Daxat Extensible Search Server yet manipulate data which is ultimately returned to an ESS client, data provider development requires references to both the ESS client and server libraries. The client-side accessible functionality of the Daxat Extensible Search Server is defined in the assembly `Daxat.Ess.ClientLibrary`, found in `Daxat.Ess.ClientLibrary.dll`. The extensible server-side functionality is implemented in the assembly `Daxat.Ess.ServerLibrary`, found in `Daxat.Ess.ServerLibrary.dll`. By default, these DLL are found in `C:\Program Files\Daxat\Extensible Search Server 2005`. Adding a reference to these assemblies to a .NET project makes the extensible types of the `Daxat.Ess` namespace available to the project.

“On Demand” Data Providers

Although the minimal requirement for a data provider is that it implements the `Daxat.Ess.DataProviders.IDataProvider` interface, implementing this interface can be tedious and the data provider may not be able to participate in all of the rich features common to the data providers which ship with ESS, such as scheduling and transformation chains. An alternative and generally easier solution is to subclass the `Daxat.Ess.DataProviders.OnDemandDataProvider` abstract class. Data providers inheriting from this base class automatically gain such rich support.

By way of example, this chapter will focus on the development of a simple file system crawler – a data provider which, given a starting directory, indexes all files contained within the directory and all sub directories.

The OnDemandDataProvider Pipeline

A Data provider based upon `OnDemandDataProvider` (ODDP) follows a simple, but powerful, processing model. An ODDP may expose administrative properties which govern its operation – for example, a file system crawler can expose a property for the starting directory. At some point, the data provider starts running, initiated in any number of ways – manually by a system administrator, according to a schedule, programmatically, etc. An ODDP developer need not worry about the mechanics of how the data provider is started – the framework manages this operation. Just before being started, all custom properties are populated within the ODDP (e.g. the starting crawl directory) and the `Initialize` method is called. This method may be overridden to handle custom initialization code.

Next, a *node stream* is created between the ODDP and its associated index. A node stream is similar to other .NET Framework streams (although not a subclass of `System.IO.Stream`) and facilitates the creation of nodes within an index.

The management of system resources, committing nodes to disk, throttling node production and various other factors is a complex task. ODDP handles all of these automatically, placing only one requirement on the ODDP developer – create the next node when asked (“on demand”). Whenever the framework is ready to process a new node, it calls the `YieldNextNode` method. The implementation of `YieldNextNode` should either push the details of a node into the node stream, or return an indication that there are no more nodes to process.

As the details of a node are pushed into the node stream, those node contents are passed through a series of user-defined *node transformations*. Node transformations take the incoming stream of data and potentially modify the stream before passing it along. Transformations can be as simple as converting upper-case text into lower-case or as complex as converting the binary contents of a Microsoft Word document into plain text.

Eventually, the data arrive at a last and final node transform which places the data directly into the index.

Creating a new OnDemandDataProvider

Begin by subclassing `OnDemandDataProvider`. Using the `DataProviderDisplayName` attribute, give a name and a description to the data provider; this information will be displayed in the ESS Manager.

If the data provider is thread safe – if a single instance of the data provider can support multiple, simultaneous calls to the `YieldNextNode` method – then the `DataProviderThreadSafe` attribute can be used to tell the framework that the provider is thread safe. Thread safe data providers can potentially perform substantially faster on servers with multiple CPUs.

The example file system crawler will be thread safe, so an appropriate class definition is

```
using System;
using System.IO;
using System.Collections;
using Daxat.Ess.Nodes;
using Daxat.Ess.Nodes.NodeContents;
using System.Diagnostics;
using System.ComponentModel;

[DataProviderDisplayName("File System Crawler", "A simple crawler for
indexing all files and folders starting at a given folder.")]
[DataProviderThreadSafe(true)]
public class ExampleFileSystemCrawler : OnDemandDataProvider
{
    ...
}
```

Custom Properties

Returning to the example, the file system crawler requires a user-specified starting directory, the data provider needs a way to collect that information from the user. Any publicly exposed class property with both a setter and a getter will be presented to the user in the ESS Manager application via a standard property grid. Standard component model attributes, such as `DescriptionAttribute` and `CategoryAttribute`, will be applied. There is no support for custom designers, however, so it will be necessary to limit the type of the property to the common base types which are readily handled by `System.Windows.Forms.PropertyGrid`.

```
string _crawlRoot;
/// <summary>
/// Gets or sets the root directory where crawling will begin.
/// </summary>
/// <remarks>
/// The directory from which crawling will begin.
/// </remarks>
[Description("Root directory from which to start crawling.")]
public String CrawlRoot
{
    get { return _crawlRoot; }
    set { _crawlRoot = value; }
}
```

Initialize method

Since it is the job of the `YieldNextNode` method to create the next node to be pushed into the node stream, it is necessary to be able to determine which file should be crawled next by this file system crawler. More importantly, the list of files to crawl needs to be seeded somehow.

A queue of files and directories is a simple solution to this problem, resulting in a breadth-first traversal of the directory tree. The seed for this queue is the value of `CrawlRoot`.

```
private Queue _crawlQueue = null;

/// <summary>
/// Initializes the instance of the data provider prior to
/// starting a new run.
/// </summary>
protected override void Initialize()
{
    base.Initialize();
    _crawlQueue = new Queue();
    EnqueueDirectory(_crawlRoot);
}

private void EnqueueDirectory(string directoryName)
{
    string[] files = Directory.GetFiles(directoryName);
    lock (_crawlQueue.SyncRoot)
    {
        for (long i=0; i < files.LongLength; i++)
            _crawlQueue.Enqueue(files[i]);
        string[] directories =
            Directory.GetDirectories(directoryName);
        for (long i=0; i < directories.LongLength; i++)
            _crawlQueue.Enqueue(directories[i]);
    }
}
```

Node Contents

Before an implementation of `YieldNextNode` can be given, it is necessary to better understand the node stream into which data will be pushed. `Daxat.Ess.Nodes.NodeStream` is the pipeline in which nodes are created, transformed and ultimately indexed.

Much like a `System.IO.Stream`, a `NodeStream` can be written to. However, whereas `Stream.Write` takes a series of bytes, `NodeStream.Write` takes `Daxat.Ess.Nodes.NodeContents.NodeContent`. `NodeContent`, in turn, is an abstract base class for various types of data which may be placed into a node.

NodeIDContent

The most important type of node content is **NodeIDContent**. Every node must have a unique identifier, and that identifier is assigned when a `NodeIDContent` is written to the node stream. **A `NodeIDContent` object must be written to the node stream before any other node content.** `NodeIDContent` must also indicate how a duplicate node id should be handled. If the `ReplaceExisting` property is set to `true` and the node id matches a node

id already stored within the index, then the new node replaces the old node. If this property is set to `false`, however, then when this `NodeIDContent` is written to the node stream a `Daxat.Ess.Indexers.FullTextIndexers.DuplicateNodeIdException` will be thrown. Example usage:

```
nodeStream.Write(new NodeIDContent("ThisIsAUniqueId"), false).
```

TextContent

The most common node content is **TextContent**, representing searchable text to be associated with the node. `TextContent` is so common that an overload of `NodeStream.Write` takes a `String` as a parameter as a shortcut to creating a `TextContent` object and writing it in the stream. The text added this way to an index is rudimentarily parsed – for example, individual words are extracted from the string, as in `nodeStream.Write("These individual words are added to the index")`. `TextContent` may be optionally tagged with additional attributes indicating the text's locale or defining the text as a particular standardized property (such as a title).

TokenContent

TokenContent is used to place a single token into the stream. It is similar to `TextContent`, but not parsing takes place – the string is added to the index as-is. For example `nodeStream.Write(new TokenContent("This entire string is a single token"))`.

NamedRegionContent and the pair **StartNamedRegion** and **EndNamedRegion** are two means for creating named regions within a node. For example, `nodeStream.Write(new NamedRegionContent("Title", "This is the title of the document"))`.

Those node content types are also known as *basic* node content types, for they have a one-to-one correspondence with what may be legally placed within a node. The ODDP pipeline need not have any associated node transformation for this content to be placed within an index. However, there are several other node content classes, and these are known as *extended* node content types; content of these types must be handled by a node transformation, converted into one or more basic node content types. If any extended node content reaches the end of the ODDP pipeline, it is discarded.

One of the extended node content types is **FileContent**. `FileContent` represents a file name and, optionally, a MIME type associated with the file name. By default, ODDP pipelines include the `IFilterDocumentConverter` node transform. Node transforms will be fully described later; it is sufficient now to understand that when `FileContent` passes through this particular node transform, the actual file is read, metadata is extracted and the body of the document is converted into plain text. The metadata is placed into the pipeline for processing by other node transforms. The plain text is added to the pipeline as `TextContent`.

OnDemandDataProvider YieldNextNode method

The `YieldNextNode` method is now relatively easy to implement. Every time `YieldNextNode` is called, the next element in the query is removed. If the query is empty, there are no more files to be indexed and the method can return `false`, indicating that processing is complete. Otherwise, the element is examined to determine if it is a file name or a directory name. If it is a directory name, then the contents of that directory can be placed into the queue via the previously defined `EnqueueDirectory` and loop to try `YieldNextNode` again. Otherwise, the element is a file name; a new node should be created and its contents populated with the contents of the file.

This translates into writing two pieces of `NodeContent` into the node stream – a mandatory `NodeIDContent`, for which the full file path and name makes a perfect identified, and a `FileContent` representing the file. Provided the appropriate node transforms are attached to the node stream (and they are by default), all of the files starting from `CrawlRoot` will be added to the index – including support for rich document types such as Microsoft Word and PDF.

```
/// <summary>
/// Writes the next document found in the file system to the node
/// stream for indexing.
/// </summary>
/// <returns><b>true</b> if there are more files to be indexed,
/// or <b>false</b> if all files have been crawled.</returns>
protected override bool YieldNextNode()
{
    while (true)
    {
        string path;
        // Although this data provider is thread safe,
        // Queue is not. Lock() the queue.
        lock (_crawlQueue.SyncRoot)
        {
            // If queue is empty, we are done
            if (_crawlQueue.Count == 0) return false;
            path = (string)_crawlQueue.Dequeue();
        }
        if (File.Exists(path))
        {
            NodeStream.Write(new NodeIDContent(path, true));
            NodeStream.Write(new FileContent(path));
            // Node is complete, return true
            return true;
        }
        else if (Directory.Exists(path))
            EnqueueDirectory(path);
    }
}
```

That completes the development of the simple file system crawler.

Data Provider Deployment

Deploying a data provider simply requires copying the assembly to the program files directory on the server, which, by default, is C:\Program Files\Daxat\Extensible Search Server 2005. As soon as the assembly is copied, its contained data providers are available for use. Using the ESS Manager to add a new data provider to an index should list the new data provider as one of the available data providers.

Advanced Data Provider Development

How to Employ Advanced Data Provider Features.

Although it is relatively easy to create a simple data provider, the Daxat Extensible Search Server fully supports complex data providers. Yet data providers whose functionality is complex do not necessarily have a comparably more complex design. This is due to the rich support provided by the `OnDemandDataProvider` (ODDP) base class.

ODDP Processing Pipeline Revisited

As discussed in ‘The `OnDemandDataProvider` Pipeline’ on page 64, an ODDP instance writes node content to a node stream, which it then passes that content through a series of node transforms and ultimately places the content into the index associated with the instance of the data provider.

The node transforms and their order attached to the pipeline, called the *transformation chain*, is typically controlled by the Daxat Extensible Search Server administrator via the ESS Manager. By default, the transformation chain is `UriReader`, `StructuredStoragePropertyExtractor`, `IFilterDocumentConverter`, `PropertyTagger` and `DuplicateNodeDetector`; each of these is fully defined within the appendix, but, in a nutshell, these transforms resolve remote URLs; extract standard document properties, such as title and author, from Microsoft Office and similar documents; convert various native document formats, such as Microsoft Word and PDF, to plain text; create named regions and metadata for the various extracted properties; and identify nodes which appear to be duplicates of other nodes.

Regardless of how the transformation pipeline is modified, all ODDP’s have two unchangeable final node transforms. The last node transformation of the pipeline is always `IndexingNodeTransform`. This special transform is responsible for responding to the node content and calling the appropriate low-level indexing APIs to populate the index. The penultimate node transform is the instance of the ODDP itself.

ODDP's Are Node Transforms

An ODDP is also a node transform, and is always the last node transform in the transformation chain just prior to the content being placed within the index. As an ODDP is located at both the start and the end of the transformation chain, it has total control over how the content it produces ultimately appears within the index.

Node transform development is fully described in its own chapter; everything described there is applicable to an ODDP. Why it's valuable for an ODDP to also be a node transform is best describe by way of an example. Consider an ODDP-based web crawler. Web crawlers take a list of starting URLs, extract the content of the URL, and index that data. But web crawlers also need to follow any links found on the crawled web pages.

An ODDP-based web crawler does not need any code to read the contents of a web page; it can simply make use of the `UriReader` node transform. The parsing of an HTML web page is handled by the `IFilterDocumentConverter` node transform, and HTML properties are placed into the node stream by the `PropertyTagger`. Ultimately, an ODDP-based web crawler need only write URLs, via new `UriContent`, to the node stream to maintain a crawl.

In order for the web crawler to know which URLs to crawl next, it needs a way to determine which URLs are contained within the currently crawled page. Thanks to the default transformation chain, the metadata of the current node includes the list of URL links found within the current web page. As an ODDP is also a node transform and is located at the end of the transformation chain, it can easily read the metadata written to the node stream and extract the list of URLs. This list can be placed into a queue of pages to be crawled, thereby making it possible for the web crawler to continue to work. Some additional work is required to avoid endless loops (a.html links to b.html, and b.html links back to a.html), but, otherwise, the development of an ODDP-based web crawler is extremely simple.

Lifecycle of an ODDP

A Daxat Extensible Search Server data provider has two distinct halves. One half of a data provider is the implementation of the data provider; any ODDP is an example of a data provider implementation. The other half of the data provider is the *data provider instance definition*, a class which maintains the user-supplied configuration for a given data provider instance. For example, consider a search solution with one index and two web crawlers. Although both web crawlers share the same implementation (the data provider), each is associated with a unique data provider instance definition.

The Data Provider Controller

The lifecycle of a running data provider is controlled by the *Data Provider Controller*, which itself runs within the Daxat Extensible Search Server. Whenever a new data provider instance is defined, for example via the ESS Manager, an instance of the data provider is created by the Data Provider Controller by calling the default constructor of the data provider implementation. Any user-exposed configuration for the data provider instance is maintained in a separate

`IDataProviderInstanceDefinition` object. As changes are made to the configuration via the ESS Manager or programmatically, the changes are placed within the associated `IDataProviderInstanceDefinition` object.

When the data provider is run, the associated `IDataProviderInstanceDefinition` object is ‘pushed’ onto the data provider instance via a process which assigns all of user exposed configuration. A data provider instance does not see its properties being updates in real-time; it simply gets its properties assigned in bulk just prior to executing.

Once the properties have been assigned, the `IDataProvider.Run` method is called. This method should not return until the data provider has finished running, either due to running to completion or forced to stop.

A running data provider maintains an operational state – start pending, running, pause pending, paused, stop pending or stopped. A data provider must also carefully maintain any spawned threads, and cooperate with other threads created within both the application and system thread pools. This maintenance is complicated and error prone.

Simplified ODDP Model

As has been demonstrated before, one of the advantages of the ODDP is that most of these operational details disappear. The ODDP base class abstracts away most of these details.

When an ODDP instance is run, the virtual `Initialize` method is called first. Regardless of the reported thread safety of the ODDP (via the `DataProviderThreadSafe` attribute), the call to the `Initialize` method is made only once per instance of the data provider.

Next, one or more threads are created for the data provider. If the `DataProviderThreadSafe` attribute is `false`, only one thread is created. If it is `true`, more than one thread may be created, depending upon the number of CPUs in the server, available resources, etc.

These threads, known as indexing threads, share a single data provider instance but have independent node streams and independent transformation chains. An instance of every node transform assigned to the transformation chain is created for **each** node stream; if two indexing threads were created, then two instances of each assigned node transform are created, plus two independent node streams. The node streams are ultimately populating the same index, however.

The indexing threads next enter into a loop whereby they first check to see if a stop request has been posted to the data provider. If not, the current crawl rate is checked. All ODDP share some common properties, one of them being the crawl rate. The crawl let permits an administrator to slow down a data provider; this is desirable when the crawled resource is simultaneously being shared by users, such as a web server. Aggressively crawling a web server could render the web server unresponsive to user interaction; slowing down the crawl rate ensures that the web crawler doesn’t overly tax the web server.

After throttling the crawl as necessary, the `OnNodeStart` method is called on the node stream. This signals the node transforms in the transformation chain that a new node is likely to be written to the node stream. Next, the virtual method `YieldNextNode` is called on the ODDP. The data provider implementation should either write a new node to the node stream and return `true`, or return `false` indicating that there are no more nodes to add to the index. Once `YieldNextNode` returns, the `OnEndNode` method of the node stream is called signaling the end of the current node and providing an appropriate time for node transforms to finish any per-node work before the node is actually added to the index.

Assuming that the data provider was not given a stop signal and `YieldNextNode` did not return `false`, the indexing thread repeats the process.

Data providers marked as thread safe have to correctly contend with the fact that a single instance of the data provider may have `YieldNextNode` called simultaneously from multiple threads. Moreover, **each** thread will need to have `false` returned by `YieldNextNode` for them run to completion.

Node Contents Revisited

It is the role of `YieldNextNode` to write node content to the node stream which is representative of the data extracted from the data repository. Basic node content and `FileContent` were described in ‘Node Contents’ on page 66. Several other forms of extended node content are defined in the `Daxat.Ess.ServerLibrary` assembly, and developers are free to define their own by subclassing `Daxat.Ess.Nodes.NodeContents.NodeContent`.

UriContent simply defines a Uniform Resource Identifier (URI), with the intent that a node transform should read the content pointed at by the URI and add it to the node stream. For example,

```
nodeStream.Write(new UriContent("http://example.com/test.htm"))
```

indicates that the contents of the web page `example.com/test.htm` should be indexed as part of the current node.

ValueContent is defined by a value object of any type and various property identifiers. These identifiers can be plain-text names, numerical identifiers, GUID, or indexes into any of these. Properties also support locales.

Properties are typically used to define common metadata across all document types. For example, the `DocTitle` property contains the title of any Microsoft Office document, such as Microsoft Word or Powerpoint. Other document formats likewise create the same property, thereby making it possible to perform a search against the titles of documents, regardless of the original types of those documents. Properties are defined within the Extensible Search Server’s configuration file; administrators may add new property definitions to this list.

Posting Runtime Messages

A running data provider may post runtime messages. These messages appear in the ESS Manager on the property page for the given data provider instance. Runtime messages should be treated as an expensive resource and should be limited to exceeding sparse and useful administrative information. Alternatively, a data provider could define a public message verbosity level configuration property which would, in turn, be exposed in the ESS Manager permitting an administrator to 'dial up' or 'dial down' the noise produced by the data provider.

All ODDPs have a `Definition` property, which references the `IDataProviderInstanceDefinition` associated with the instance of the data provider. The `Definition.PostRuntimeMessage` method may be used to post a new runtime message; care must be taken, however, for the methods of `Definition` are not thread safe. For example:

```
lock (Definition)
    Definition.PostRuntimeMessage(
        new RuntimeMessage("This is visible in the Manager"));
```


Node Transform Development

How to Develop Node Transforms for the Daxat Extensible Search Server.

Node transforms are an important part of the process of indexing data. Data providers write node content to a node stream, both basic and extended node content. Extended node content cannot be indexed directly; instead, it must be converted by a node transform into one or more instances of basic node content.

Some node transforms are useful in nearly any search solution and therefore are assigned to the initial transformation chain of every data provider, although an administrator is free to reorder or remove any of these default node transforms. Other node transforms are special purpose or computationally expensive, and therefore will only be part of a transformation chain is specifically added to the chain.

Referencing the ESS Libraries

As node transforms execute within the Daxat Extensible Search Server yet manipulate data which is ultimately returned to an ESS client, node transform development requires references to both the ESS client and server libraries. The client-side accessible functionality of the Daxat Extensible Search Server is defined in the assembly `Daxat.Ess.ClientLibrary`, found in `Daxat.Ess.ClientLibrary.dll`. The extensible server-side functionality is implemented in the assembly `Daxat.Ess.ServerLibrary`, found in `Daxat.Ess.ServerLibrary.dll`. By default, these DLL are found in `C:\Program Files\Daxat\Extensible Search Server 2005`. Adding a reference to these assemblies to a .NET project makes the extensible types of the `Daxat.Ess` namespace available to the project.

The NodeTransform Base Class

Although the minimal requirement for a node transform is that it implements the `Daxat.Ess.Nodes.NodeTransforms.INodeTransform` interface, implementing this interface may be slightly tedious. An alternative and generally easier solution is to subclass the `Daxat.Ess.Nodes.NodeTransforms.NodeTransform` abstract class.

By way of example, this chapter will focus on the development of a simple node transform which ensures that only lower-case text is placed within the index. This is a useful operation if case-

sensitive searches are not desired in the search solution, for indexing only lower case text will result in a smaller index.

Thread Safety of Node Transforms

In general, instances of node transforms need not worry about thread safety as each instance will have thread affinity. However, node transform implementations must support multiple concurrent instances.

The except to this rule are `OnDemandDataProviders` (ODDPs) acting as node transforms. By their very nature, ODDPs may be multithreaded. The major threading issue which must be avoided is concurrent writes to the associated node stream. The ODDP alleviates this problem by exposing a protected `NodeStream` property. Writes to this stream are thread safe, and correctly target the in-context node stream.

The `INodeTransform` Interface

An instance of a node transform is created whenever it is assigned to a transformation chain and the transformation chain is created; node transform creation involves calling the default constructor of the node transform. Typically, this occurs when an `OnDemandDataProvider` (ODDP) starts running and creates an indexing thread. Sometimes an ODDP will start more than one indexing thread, and multiple node transformation chains will be created, each containing unique instances of any assigned node transforms. Recall that an ODDP places itself at the penultimate location of the transformation chain, followed by a special indexing node transform in the final location. An ODDP running on multiple threads will have the **same** instance acting as a node transform on **each** transformation chain associated with the data provider instance.

The `Initialize` Method

Once the transformation chain is created, the `INodeTransform.Initialize` method is called on each node transform instance. This method passes some objects to the node transform which may be useful during the life time of the node transform, such as a reference to the `NodeStream` to associate with the node transform; it is also an appropriate time for node transforms to perform any per-instance initialization. Node transforms derived from either `NodeTransform` or `OnDemandDataProvider` which override the `Initialize` method should take care to call the base implementation of the method.

After initialization, a node transform will experience a repeating series of method calls: `OnNodeStart`, zero or more calls to `Write`, and finally `OnNodeEnd`. This series will repeat until the data provider is complete, at which point `Cleanup` will be called and the node transform instance will be made available for garbage collection.

The `OnNodeStart` Method

A call to the `INodeTransform.OnNodeStart` method signals the node transform that a new node may be written to the node stream. In certain circumstances, a new node will not be

created, but this is easily detected because `OnNodeEnd` will be called without any calls to `Write`. A node transform can override this method to perform any per node initialization.

The OnNodeEnd Method

A call to the `INodeTransform.OnNodeEnd` method signals that the current node is complete; a node transform may override this method to perform any per node cleanup.

The Cleanup Method

The `INodeTransform.Cleanup` method is called when the data provider no longer needs the node transform, typically indicative that the data provider has completed running. This method may be overridden to handle per-instance cleanup.

The Write Method

All of the `INodeTransform` methods described so far are only useful for responding to lifecycle events of the node transform, providing convenient places to allocate and deallocate resources. The `INodeTransform.Write` method is where real work can be done.

When `NodeContent` is written to a `NodeStream`, that content is written to each `NodeTransform` contained within the data provider's transformation chain, in the order defined by the transformation chain. Each node transform, in turn, can respond to the node content in whatever way is appropriate, or simply ignore the node content. For every call to `Write`, a node transform must return either `true` or `false`, indicating if processing of the node content should continue. Returning `true` indicates that the node content should be passed on to the next node transform; returning `false` indicates that the node content has been fully handled and should be removed from the transformation chain. As such, it will not reach the final indexing node transform and will not end up in the index.

The implementation of `Write` is free to call `NodeStream.Write` to insert additional content into the current node stream. Any content added this way immediately starts to pass through the transformation chain, starting at the beginning of the chain. As such, the new content may pass through the same node transform which created the content, and care must be taken to avoid infinite node content processing loops.

The NodeTransform Base Class

The `NodeTransform` base class simplifies the development of a node transform in two ways. First, it implements reasonable default behavior for every `INodeTransform` method. Second, it replaces the `Write` method into a more event-based model.

The implementation of `NodeTransform.Write` looks at the type of the written node content, and calls an appropriate virtual method based upon the node content type. For example, if `TextContent` is written to `NodeTransform`, then the virtual method `NodeTransform.OnTextContent` is called. As a result, classes derived from `NodeTransform` only need override the methods related to the type of node content against

which the node transform can operate. Unhandled node content is simply passed on to the next node transform in the transformation chain.

Armed with these tools, implementing the example of a node transform which only indexes lower-case content is relatively simple:

```
using System;
using Daxat.Ess.Nodes.NodeTransforms;
using Daxat.Ess.Nodes.NodeContents;

public class LowerCaseNodeTransform : NodeTransform
{
    public override bool OnTextContent(TextContent textContent)
    {
        // Compute lower-cased version of current
        // block of text to be index.
        String textAsLower = textContent.Text.ToLower();
        // If the text was already all lower-case,
        // return true so that it will be processed
        // by the remainder of the transformation chain.
        // This avoids endless loops
        if (textAsLower.Equals(textContent.Text))
            return true;
        // Otherwise, write the all lower-case version of
        // the text to the NodeStream
        NodeStream.Write(textAsLower);
        // And return false so that the source text is
        // not processed and therefore does not end
        // up in the index.
        return false;
    }
}
```

This example takes special care to check to see if the text to be added to the index is already in all lower case. If it is, then the text is simply passed on to the next node transform, whereby it should eventually end up in the index. The exception to this would be in the event that a node transform further down the transformation chain elects to discard the content.

If this check did not exist, and the node transform were written as follows, an infinite loop will result.

```
public override bool OnTextContent(TextContent textContent)
{
    // DON'T DO THIS - THIS CODE IS FLAWED!

    // Write the all lower-case version of
    // the text to the NodeStream
    NodeStream.Write(textContent.Text.ToLower());
    // And return false so that the source text is
    // not processed and therefore does not end
    // up in the index.
    return false;
}
```

The reason why this code is flawed is due to the fact that writing to a node stream places the node content at the start of the node transformation pipeline. Therefore, the lower-cased text written by this node transform will eventually return to this same node transform, be converted to lower case, and placed back into the node stream at the start of the transformation chain. This endless loop will continue until either system resources are exhausted or there is a stack fault due to too many recursive calls.

Advanced Processing via NodeContent.ProcessingInformation

Avoiding these types of endless node transformation loops is critical. In some instances, such as the lower case example, it is possible to determine whether or not to create new node content based upon the content itself. In other cases, it is not so easy and alternative mechanisms must be found.

The `NodeContent` base class has a public `Hashtable` property named `ProcessingInformation` whose initial value is `null`. This member exists so that node transforms may tag any piece of node content with additional information, either to communicate with themselves in the event of a write loop, or to communicate with other node transforms in an act of collusion.

A node transform is free to make use of `ProcessingInformation` in any way, but, in an effort to avoid collisions with other node transforms which may desire to use `ProcessingInformation`, care should be taken to ensure that information placed within the hashtable by other node transforms remains intact.

One technique for addressing this issue is to assign a private GUID to a node transform class, and then use this GUID as a key into `ProcessingInformation`. The value object keyed off of the GUID can be considered to be privately owned by the node transform.

Here's an alternative solution to the lower-case node transform which makes use of `ProcessingInformation` and a private GUID:

```
using System;
using System.Collections;
using Daxat.Ess.Nodes.NodeTransforms;
using Daxat.Ess.Nodes.NodeContents;

public class LowerCaseNodeTransform : NodeTransform
{
    private static Guid MyGuid =
        new Guid("{E7363D44-5024-49bd-A134-32C7A83B8027}");

    public override bool OnTextContent(TextContent textContent)
    {
        // If MyGuid is in ProcessingInformation, this
        // content has already been handled; ignore and
        // pass to next node transform.
        if (textContent.ProcessingInformation != null &&
            textContent.ProcessingInformation[MyGuid] != null)
            return true;
        // If ProcessingInformation is null, create it
        if (textContent.ProcessingInformation == null)
            textContent.ProcessingInformation =
                new Hashtable();
        // Create a new TextContent which is lower-case
        // version of current TextContent
        TextContent lowerText =
            new TextContent(textContent.Text.ToLower());
        // Place MyGuid in ProcessingInformation, indicating
        // this content has been handled by this node trans.
        lowerText.ProcessingInformation[MyGuid] = "handled";
        // Write the all lower-case version of
        // the text to the NodeStream
        NodeStream.Write(lowerText);
        // And return false so that the source text is
        // not processed and therefore does not end
        // up in the index.
        return false;
    }
}
```

Although for this simple example using `ProcessingInformation` is likely overkill, the technique is sound for more complex situations.

Scorer Development

How to Develop Custom Ranking Algorithms for the Daxat Extensible Search Server.

Nearly any modern search engine can find matches to a query both accurately and quickly. What sets such search engines apart is their ability to intelligently order the results displayed to the user. The process of ordering the results is often referred to as *relevancy ranking*. In many commercial search engines, the algorithms which rank the results are a closely guarded secret. Sometimes these engines expose parameters which permit the behavior of the ranking algorithm to be tweaked; in the worst cases the ranking algorithm is a black box which exposes no external controls. Relevancy ranking is just one form of ranking; sorting by value, such as a reverse chronological sort of documents by publication date, is another form of ranking.

The Daxat Extensible Search Engine adopts an extensible model for ranking search results. Any algorithm capable of producing an `IComparable` result for each search result may be used; moreover, Daxat's supplied algorithms are fully documented and available in source code form. This source code may be used as the starting point for developing custom scorers, or a scorer may be written from scratch. Like data providers, deploying a scoring implementation is as easy as dropping the assembly within the server's program directory.

Referencing the ESS Libraries

As scorers execute within the Daxat Extensible Search Server yet manipulate data which is ultimately returned to an ESS client, scorer development requires references to both the ESS client and server libraries. The client-side accessible functionality of the Daxat Extensible Search Server is defined in the assembly `Daxat.Ess.ClientLibrary`, found in `Daxat.Ess.ClientLibrary.dll`. The extensible server-side functionality is implemented in the assembly `Daxat.Ess.ServerLibrary`, found in `Daxat.Ess.ServerLibrary.dll`. By default, these DLLs are found in `C:\Program Files\Daxat\Extensible Search Server 2005`. Adding a reference to these assemblies to a .NET project makes the extensible types of the `Daxat.Ess` namespace available to the project.

The IScorer Interface

A *scorer* is any type which implements the `Daxat.Ess.Scoring.IScorer` interface. The interface is extremely simple, comprising a single method `Score()`. The `Score` method takes a single argument of type `Daxat.Ess.Scoring.ScoringContext` and does not return a value. The job of a scorer is to iterate over the search results references by the scoring context and assign to each search result a score, typically dependent upon the details of each individual search result and the query which produced the search result set. All of these factors are available through the scoring context.

Scoring Contexts

The `ScoringContext` object contains all of the information necessary to assign meaningful scores to each search result. The most important property is `QueryResponse`, which contains the query response for the executed query. The scorer should assign scores to the `Score` property of `QueryResponse.SearchResults[]`. Recall that the `Score` property is actually an array of values, supporting multi-level sorting. The sorting level to which the scorer is being applied is available via the `ScoringContext.ScoreIndex` property; in other words, scores should be assigned to `scoringContext.QueryResponse.SearchResults[i].Score[scoringContext.ScoreIndex]`.

The `ScoringContext.Query` property contains the query which produced the result set and the `Terms` property contains the individual terms parsed from the query. The `ScoreObject` property is an object which may be passed in at query time and may be used by the scorer in any way desired; it is typically used to pass custom arguments to the scorer. The `Index` property provides special access to the internal index against which the query was executed, permitting the scorer to make use of information about the index itself. This will be particularly important for evaluating relevancy algorithms.

A Simple Scoring Example

An extremely simple scorer is one which ranks each search result by a named metadata element. For example, if each node's metadata includes a key named `Price` assigned to a numerical value, then scoring on `Price` will sort the search results by `Price`. Similarly, scoring on a date value will sort the nodes in chronological order.

The following sample scores each search result based upon some named metadata key. The metadata key is passed to the scorer via the `ScoringContext.ScoreObject` property, demonstrating how a scorer can accept custom parameters.

```
public class SimpleScorer : IScorer
{
    public void Score(ScoringContext context)
    {
        QueryResponse queryResponse = context.QueryResponse;
        object scoreObject = context.ScoreObject;
        if (scoreObject == null)
            throw new ArgumentException(
                ("scoreObject must be a non-null key", "scoreObject"));
        foreach(SearchResult searchResult
            in queryResponse.SearchResults)
            searchResult.Score[context.ScoreIndex] =
                searchResult.Metadata[scoreObject] as IComparable;
    }
}
```

Using this scorer is relatively straightforward. Here, the scorer is used and supplied `Price` as the metadata key:

```
ScorerDefinition sd = new ScorerDefinition(
    "Daxat.Ess.Scoring.SimpleScorer, Daxat.Ess.Scoring.SimpleScorer",
    "Price", SortOrder.Descending);
QueryResponse qr = index.ExecuteQuery("foo", sd);
```

Relevancy Ranking

Relevancy ranking algorithms attempt to score search results such that the highest ranking search results are, with great probability, the best matches for the given query. Consider the query `apple pie`. This will match all nodes containing the words `apple` and `pie`. In a sufficiently large index, this query could return hundreds, if not thousands, of search results. A user would likely consider a node containing a recipe for apple pie to be a highly relevant search result, whereas a node containing information on constructing pie charts that happens to also have the word `apple` buried deep in some obscure example would likely be considered to be irrelevant.

How a computer program determines that the first example node is significantly more relevant than the second is an open area of on-going research. Over the years, several techniques have been used. For example, a node containing more occurrences of the query terms is often more relevant than a node containing fewer occurrences. In the previous example, a node containing 10 occurrences of the word `apple` and 12 occurrences of the word `pie` would be considered to be more relevant than a node containing 3 occurrences of the word `apple` and 5 occurrences of the word `pie`.

This simplistic approach has a whole set of issues. For example, what if node A contains 1 occurrence of ‘apple’ and 10 occurrences of ‘pie’, whereas node B contains 10 occurrences of ‘apple’ and 1 occurrence of ‘pie’? Which node is more relevant? The classic approach to this problem is to consider how common each query term is throughout all of the nodes in the index. Consider an index of food recipes. Of these, 5% are pie recipes, yet 30% of the recipes contain apples. The word ‘pie’ is a rarer term than ‘apple’, so node A would be considered to be more relevant than node B.

Other relevancy ranking techniques consider the location of the terms found within the node. For example, some relevancy algorithms consider words found closer to the beginning of the node to be more relevant than those found further into the node. A particularly powerful technique involves scoring higher words found within particular named regions of a node, such as the title. In the previous example, a node containing both ‘apple’ and ‘pie’ in the title would be considered more relevant than a node mentioning ‘apple’ and ‘pie’ within the body of the node.

Implementing relevancy algorithms such as these requires access to complex statistical information about the contents of an index. Recall that the `ScoringContent` object includes an `Index` property. This property, in turn, provides access to the `ComputeRelevancyStatistics` method which can efficiently compute the necessary statistics.

IFullTextIndex.ComputeRelevancyStatistics

The `ComputeRelevancyStatistics` method is capable of computing several different index-related statistics. For performance reasons, a scorer should call this method only once, and request the minimum set of statistics. Although it is possible to call this method more than once, requesting a different statistic each time, the implementation of `ComputeRelevancyStatistics` has been optimized to compute several statistics simultaneously.

The various statistics which can be computed are defined by the enumeration `Daxat.Ess.Scoring.Relevancy.ComputableStatistics`. The desired statistics should be logically or’d together and passed to the `ComputeRelevancyStatistics` method. Each available statistic is described in the following table.

Computable Statistic	Description
None	No statistics will be computed.
TotalNodes	Computes the number of nodes contained with in the index.
PositionsByTerm	Computes the positions of each term found in each search result; positions are returned on a term-by-term basis.
Positions	Computes the position of each term found in each search result; positions are returned as one set, not broken down by term.
RegionRanges	For each named region present in the original query, returns the range of positions of the named region for every search result.
TermCount	Computes the total occurrences in the index of every query term.
TermNodeCount	For each query term, computes the total number of nodes within the index containing that term.
All	All statistics will be computed.

`ComputeRelevancyStatistics` also requires a **Daxat.Ess.Scoring.Relevancy.RelevancyCriteria** object. Typically, a `RelevancyCriteria` object would be created by the caller of `ExecuteQuery` and passed via `ScoringContext.ScoreObject`. The `RelevancyCriteria` object provides a convenient and consistent way for `ExecuteQuery` callers to supply relevancy computation criteria to any relevancy algorithm. Implementers of custom relevancy algorithms are encouraged to accept `RelevancyCriteria` or a custom subclass of `RelevancyCriteria` as one of the means for callers to pass in relevancy criteria.

Relevancy criteria defines the factors which should be used to determine how relevant a search result is, and how those various factors should be weighed against one another. For example, relevancy criteria may state that the words 'apple' and 'pie' are relevant no matter where found they are within a node, but if either term is found in the title of the node then the weight is doubled.

If `RelevancyCriteria.IncludeTermsFromQuery` is `true`, then it is the job of the scorer to include all query terms as relevancy terms. If this property is `true` and relevancy terms are also supplied in the relevancy criteria, then the merge of the two sets of terms should be used during relevancy computation.

Relevancy criteria also instructs the relevancy scorer if the final scores assigned to each search result should be normalized via the `RelevancyCriteria.NormalizeScores` property. If requested, it is the job of the scorer to normalize the scores, although a scorer is free to ignore this request, especially if normalization isn't applicable (i.e. the scores are not numeric).

TF·IDF Scorer Example

Term Frequency Inverse Document Frequency (TF·IDF) is one of the most common relevancy ranking algorithms. The relevancy score assigned to a node i is the summation of the multiplication of the number occurrences of every query term T_j within node i (tf_{ij}) times the inverse of the number of nodes in the index containing T_j (the inverse document frequency). This algorithm assigns more weight to a search result which contains more occurrences of a term T_j than another search result, but also assigns more weight for terms which are rare within the index. Often, the IDF is computed as $\log_2(N/n_j)$, where N is the number of nodes in the index and n_j is the number of nodes containing at least one occurrence of the term T_j .

Therefore, the score S of node i can be expressed as

$$S(i) = \sum tf_{ij} * \log_2 \left(\frac{N}{n_j} \right)$$

In this equation, tf_{ij} can be deduced from the statistic `PositionsByTerm`, N is `TotalNodes` and n_j is `TermNodeCount`.

One potential implementation of this algorithm as a Daxat Extensible Search Server scorer:

```
using System;
using Daxat.Ess.Scoring;
using Daxat.Ess.Scoring.Relevancy;
using Daxat.Ess.Indexers.FullTextIndexers;
using Daxat.Ess.Results;

public class TfIdfScorer : IScorer
{
    public void Score(ScoringContext context)
    {
        IFullTextIndex index = context.Index;
        String query = context.Query;
        QueryResponse queryResponse = context.QueryResponse;

        RelevancyCriteria relevancyCriteria =
            context.ScoreObject as RelevancyCriteria;
        if (relevancyCriteria == null) throw new ArgumentException(
            "scoreObject must be of type RelevancyCriteria and non-null",
            "scoreObject");

        // Merge relevancy terms from supplied criteria with terms from the
        // executed query as necessary
        if (relevancyCriteria.Terms == null)
        {
            if (relevancyCriteria.IncludeTermsFromQuery && context.Terms != null)
            {
                relevancyCriteria.Terms = new TermWeightPair[context.Terms.Length];
                for (int i=0; i < context.Terms.Length; i++)
                    relevancyCriteria.Terms[i] = new TermWeightPair(context.Terms[i]);
            }
            else relevancyCriteria.Terms = new TermWeightPair[0];
        }
        else if (relevancyCriteria.IncludeTermsFromQuery &&
            context.Terms != null)
        {
            TermWeightPair[] fullTermsSet =
                new TermWeightPair[relevancyCriteria.Terms.Length +
                    context.Terms.Length];
            for (int i=0; i < relevancyCriteria.Terms.Length; i++)
                fullTermsSet[i] = relevancyCriteria.Terms[i];
            for (int i=0; i < context.Terms.Length; i++)
                fullTermsSet[i + relevancyCriteria.Terms.Length] =
                    new TermWeightPair(context.Terms[i]);
            relevancyCriteria.Terms = fullTermsSet;
        }
    }
}
```

```

// Call ESS to compute minimal needed set of statistics.
RelevancyStatistics relevancyStatistics =
    index.ComputeRelevancyStatistics(
        relevancyCriteria,
        queryResponse,
        ComputableStatistics.PositionsByTerm |
        ComputableStatistics.TotalNodes |
        ComputableStatistics.TermNodeCount);

Int64 N = relevancyStatistics.TotalNodes;
double maxScore = 0.0; // Used to normalize scores if requested
SearchResult searchResult;
// Loop over each search result
for (long i=0; i < queryResponse.SearchResults.LongLength; i++)
{
    searchResult = queryResponse.SearchResults[i];
    double score = 0.0;

    // Loop over every relevancy term
    for (int j=0; j < relevancyCriteria.Terms.Length; j++)
    {
        if (relevancyStatistics.PositionsByTerm[i,j] != null &&
            relevancyStatistics.PositionsByTerm[i,j].Length > 0)
        {
            // TFij is the number of occurrences of term j in node i,
            // the same as the number of positions for term j in node i.
            // PositionsByTerm[i,j] is an array of positions
            // for term j in node i,
            // so its length is the number of occurrences of term j in node i
            long TFij = relevancyStatistics.PositionsByTerm[i,j].LongLength;
            double IDFj =
                Math.Log(N/(double)relevancyStatistics.TermNodeCount[j],2);
            // Add TF*IDF for the current term to the summation for the
            // current search result
            score += TFij * IDFj;
        }
    }
    searchResult.Score[context.ScoreIndex] = score;
    if (score > maxScore) maxScore = score;
}

// Normalized results if requested
if (relevancyCriteria.NormalizeScores)
{
    for (long i=0; i < queryResponse.SearchResults.LongLength; i++)
    {
        searchResult = queryResponse.SearchResults[i];
        searchResult.Score[context.ScoreIndex] =
            (double)searchResult.Score[context.ScoreIndex]/maxScore;
    }
}
}
}

```


III. Appendixes

ESS Query Language (ESSQL) Reference

Query Language References for the Daxat Extensible Search Server.

On the surface, the query language of the Daxat Extensible Search Server appears to be nearly the same as other Boolean-based search engines. As such, anyone familiar with common search query syntax will be able to immediately make use of an ESS-based search solution. At the same time, the ESS Query Language (ESSQL) includes robust operators and features not found in other search engines, backed by the Daxat Extensible Search Server's optimizing query processor.

ESSQL Queries

ESSQL queries are of the form

```
expr [AGGREGATE metadataname+]
```

Where *expr* is an ESSQL expression and *metadataname*+ is a common-separated list of metadata key names contained within square brackets. For example, `ford OR honda AGGREGATE [color], [make]` is a valid ESSQL query. Like all ESSQL keywords, the AGGREGATE keyword is case-insensitive. The AGGREGATE clause is optional.

ESSQL Expressions

ESSQL expressions are comprised of terms and operator expressions. A term can be as simple as a word (e.g. `apple`) or as complex as a compound operator expression. By combining expressions and terms, both simple and complex queries can be formulated.

Words

The most basic ESSQL term is the *word*. Words are alphanumeric strings which are matched case-sensitively against node content. As indexed content is typically stored as both case-preserving and lower-case at the same position, queries containing lower-case words typically behave as case-insensitive searches. Non-alphanumerical characters are treated as white space and join the supporting alphanumerical characters as a phrase.

Examples:

`Apple` – single word query; matches all instances of `Apple` within the index case-sensitively.

`apple` – single word query; matches all instances of `apple` within the index case-sensitively. However, since it is common for words to be indexed as both case-preserving and as lower-case, this query expression will likely also match instances of `Apple`, `APPLE`, `ApPLe`, etc.

`dvc-200M.2` – a three word phrase, equivalent to `"dvc 200M 2"`. Word case-sensitivity rules apply individually to all three words.

Tokens

A *token* is the most elementary ESQQL term; it is matched against node content as a literal string match. Tokens are delimited by curly braces – `{ }`.

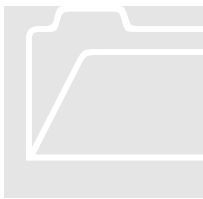
`{Apple}` – single token query; matches all instances of `Apple` within the index case-sensitively.

`{apple}` – single token query; matches all instances of `apple` within the index case-sensitively. However, since it is common for words to be indexed as both case-preserving and as lower-case, this query expression will likely also match instances of `Apple`, `APPLE`, `ApPLe`, etc.

`{sku dvc-200M.2}` – single token query; matches all instances of the token `sku dvc-200M.2` within the index. Note that this is decidedly different than matching against the phrase `"sku dvc 200M 2"`. The phrase spans 4 index positions, whereas a token is a single position. This expression will only match that precise token added to the index as token node content.

Tokens are useful for tagging nodes since it is possible to create tokens which are not resolvable as words and therefore generally unsearchable by users. For example, an e-commerce search solution could place the token `ON SALE` within the node content of those items which are on sale; user's searching for the words `on` or `sale`, or the word phrase `"on sale"` will not match this token. Yet a custom user interface could have an 'on sale' checkbox, and, when checked, the search solution could automatically append `{ON SALE}` to the user's query.

Evaluation of token expressions is one of the most efficient operations of the Daxat Extensible Search Server query processor, therefore tokens are used extensively internally by both the indexing and query engines. Tokens starting with a single space are, by convention, reserved by the Daxat Extensible Search Server. Although no error will be generated if such tokens are created, the results of using such tokens is unpredictable. For example, each node includes the token `·Inodeid`, where `·` is a leading space and `nodeid` is the unique node id assigned to the node. A search for `{ IHQW-5546/2 }` will find the node with node id `HQW-5546/2`, if it exists.



only.

This methodology for finding nodes by node id, although valid for the 2005 release of the Daxat Extensible Search Server, should not be considered to be reliable and is only described here by means of demonstrating how the internal engine makes use of reserved tokens; the actual implementation of how node ids are stored within the index is subject to change. Search solutions which must find content based upon node id should make use of the published APIs

Wildcards

Words may contain wildcards. The `?` wildcard matches any single character, and the `*` wildcard matches zero or more characters. Any combination of wildcards may be used within a single word, however query evaluation performance degrades for wildcards nearer the beginning of a word. For example, the query `st*` will likely perform slower than the query `sta*`. Leading wildcards are particularly susceptible to performance issues; expect query terms such as `?re` and `*ed` to demonstrate poor performance.

The exception to this rule is the term `* --` that is, the `*` wildcard all by itself. This term is optimized to quickly match every node within the index. It performs well even when combined with other query expressions, such as in `* AND foo`.

Phrases

Double-quotes `" "` may be used to delimit phrases. Phrases match the specified sequence of terms, in order. Phrases may consist of both words and tokens, and the word terms may include wildcards. For example, `"this is a compl?x phr* with a { &token++}"` will match, among other things, a node containing the text content `this is a complex phrase with a` followed by the token `&token++`.

Named Regions

`regionname:expr`

Limits the matching of the expression `expr` to the named region `regionname` within the node content. For example, `DocTitle:"moby dick"` matches those nodes which have at least one named region named `DocTitle` and that region contains the phrase `moby dick`.

Expression Grouping

`(expr)`

Parentheses `()` may be used to force an evaluation order of a sub-expression, or to overcome the precedence rules of the query processor. For example, consider

```
DocSubject:bird or fish
```

versus

```
DocSubject:(bird or fish)
```

The first query will match any node with the word `bird` in the named region `DocSubject` or the word `fish` anywhere within the node content, whereas the second query will match any node with either the word `bird` or `fish` within the `DocSubject` named region.

Comparison Expressions

```
<% cexpr %>
```

A comparison expression is delimited by `<% %>` and is typically used to use search against node metadata. The embedded *cexpr* is written using *comparison expression operators*, described later in this appendix.

Comparison expressions typically do not perform as fast as the other expressions and are not a substitute for a normal search. However, when used in conjunction with other query terms, comparison expressions perform well and provide a powerful additional layer of filtering. For example `blue padded chair AND <% [price] < 100.0 %>` will match nodes containing the words `blue`, `padded` and `chair`, and with metadata named `price` which can be evaluated as a numerical value less than 100.

Query Term Correlation

Query term correlation is the process by which the Daxat Extensible Search Server query processor resolves compound query expressions. As each query sub-expression is evaluated, it is assigned a current correlation position. This position reflects the location in the index at which the current instance of the expression was found.

For example, the `SUCCEEDS` operator is correlated to its first sub-expression; the expression `foo SUCCEEDS bar WITHIN 7 WORDS` is correlated to the locations of `foo`. Query term correlation makes it possible to write a phrase as a sequence of `PRECEDES` expressions. The query expression `"hot apple pie"` is equivalent to `hot PRECEDES (apple PRECEDES pie BY 1 WORDS) BY 1 WORDS`; the nested `apple PRECEDES pie` will be evaluated first, and will correlate to the locations of `apple`. The outer expression can therefore be thought of as `hot PRECEDES apple` for those instances of `apple` which exist just prior to `pie`.

Due to the way the terms are correlated, the location of the parentheses is extremely important. The expression `(hot PRECEDES apple BY 1 WORDS) PRECEDES pie BY 1 WORDS` is **not** an equivalent expression; the `hot PRECEDES apple` expression correlates to the locations of `hot`, thereby reducing the second expression to `hot PRECEDES pie` for those instances of `hot` immediately before `apple`. In other words, this expression matches those nodes with instances of `hot` immediately before both

apple and pie, with apple and pie occupying the same location in the index. In general, different words do not occupy the same location (although specialized search solutions may choose to co-index words at the same location), so this query will typically not match anything.

ESSQL Operator Reference

After

expr₁ **after** *expr₂*

Matches all nodes containing an instance of *expr₁* after an instance *expr₂*.

And

expr₁ **and** *expr₂*

Matches all nodes which satisfy both *expr₁* and *expr₂*.

AtLeast

(**atleast** | **at least**) *n* *expr*

Matches all nodes containing a minimum of *n* occurrences of *expr*.

AtMost

(**atmost** | **at most**) *n* *expr*

Matches all nodes containing a maximum of *n* occurrences of *expr*.

Before

expr₁ **before** *expr₂*

Matches all nodes containing an instance of *expr₁* before an instance *expr₂*.

Between

expr₁ **between** *expr₂* (inclusive | exclusive) and *expr₃*
(inclusive | exclusive)

Matches all nodes containing an instance of *expr₁* located between any instances of *expr₂* and *expr₃*. The inclusive and exclusive operators control whether or not *expr₁* is permitted to have the same position as *expr₂* or *expr₃*.

Contains

expr₁ **contains** *expr₂*

Matches all nodes containing an instance of *expr₁* positional contained within the span of positions defined by *expr₂*.

Near

*expr*₁ **near** *expr*₂

Shorthand for *expr*₁ within 10 words of *expr*₂

Not

not *expr*

Matches all nodes which do not satisfy *expr*.

Or

*expr*₁ **or** *expr*₂

Matches all nodes which satisfy either *expr*₁ or *expr*₂, or both.

Precedes

*expr*₁ **precedes** *expr*₂ (within | by) *n* words

Matches all nodes containing an instance of *expr*₁ exactly *n* word positions before *expr*₂ (by) or up to *n* word positions before *expr*₂ (within).

Succeeds

*expr*₁ **succeeds** *expr*₂ (within | by) *n* words

Matches all nodes containing an instance of *expr*₁ exactly *n* word positions after *expr*₂ (by) or up to *n* word positions after *expr*₂ (within).

Within

*expr*₁ **within** *n* words of *expr*₂

Matches all nodes containing an instance of *expr*₁ within *n* word positions of *expr*₂, either before or after.

Comparison Expressions Operator Reference

Comparison expressions are comprised of a comparison operator and one or more *value expressions*. Comparison expressions are Boolean. If a node satisfies a comparison expression, it is included in the result set.

=

*vexpr*₁ = *vexpr*₂

If *vexpr*₁ is equal to *vexpr*₂, true. Otherwise, false.

!=

$vexpr_1 \neq vexpr_2$

If $vexpr_1$ is not equivalent to $vexpr_2$, true. Otherwise, false.

<

$vexpr_1 < vexpr_2$

If $vexpr_1$ is less than $vexpr_2$, true. Otherwise, false.

<=

$vexpr_1 \leq vexpr_2$

If $vexpr_1$ is less than or equal to $vexpr_2$, true. Otherwise, false.

>

$vexpr_1 > vexpr_2$

If $vexpr_1$ is greater than $vexpr_2$, true. Otherwise, false.

>=

$vexpr_1 \geq vexpr_2$

If $vexpr_1$ is greater than or equal to $vexpr_2$, true. Otherwise, false.

Value Expressions Reference

Operators are listed in precedence order.

()

$(vexpr)$

Value expression grouping.

- (unary)

$-vexpr$

The negation of $vexpr$.

$vexpr_1 * vexpr_2$

The multiplication of $vexpr_1$ and $vexpr_2$.

/
 $vexpr_1 / vexpr_2$

The division of $vexpr_1$ by $vexpr_2$.

+
 $vexpr_1 + vexpr_2$

The summation of $vexpr_1$ and $vexpr_2$.

- (binary)
 $vexpr_1 - vexpr_2$

The difference of $vexpr_1$ and $vexpr_2$

Integer
Any sequence of digits

The integer value of the sequence of digits.

Real
Any sequence of digits with a single decimal point.

The real value of the sequence of digits.

String
"vexpr"

Evaluate $vexpr$ as a string.

Date/Time
#vexpr#

Evaluate $vexpr$ as a date/time value.

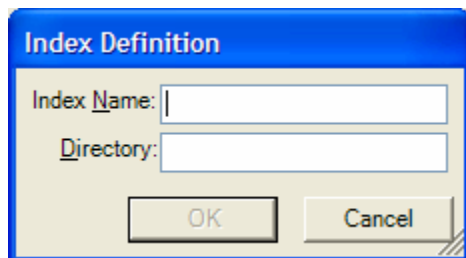
Index Administration

Index Creation and Maintenance.

For the most part, Daxat Extensible Search Server indexes are self-maintaining. Operations common to other search engines, such as index ‘compaction’ or ‘optimization’ are either not required by ESS, due to the superior index format, or are handled automatically by the system.

Simple Indexes

For the ESS Manager, there are two ways to create a new index. First, highlight either the root server node or the ‘Indexes’ node in the tree control. Next, either select ‘New Simple Index...’ from the Action menu or right-click the node and select ‘New Simple Index...’ This same functionality may be used to **attach to an existing index**. When filling out the index definition dialog



if the supplied directory already contains an index, then that index will be associated with the server. **Its contents will not be erased**; rather, the existing index will be opened and made available for queries and updates.

The Daxat Extensible Search Server requires read and write access to the index directory. Depending upon which security account the Daxat ESS service has logged in as, it may be necessary to modify the access rights of the directory before creating or attaching to an index.

The Daxat Extensible Search Server will attempt to create the entire path specified as the index’s directory, should the path not exist.

Data Providers Reference

How to Use the Included Data Providers.

The Daxat Extensible Search Server ships with several data providers, providing the ability to index data from the most common data repositories. These data providers share some common configuration properties and they also expose custom configuration properties specific to the type of repository which they crawl.

Common Properties

Every shipped data provider is an instance of an `OnDemandDataProvider`, and, as such, inherits some common configuration properties.

CrawlRate Property

Type: `int`

Default: 0

Read Only: `false`

`CrawlRate` controls the speed at which the data provider will attempt to pull data from the associated data repository. It is the minimum amount of time, expressed in milliseconds, which the data provider will wait between each access to the data repository. More precisely, it is the minimum amount of time between calls to `YieldNextNode`. For example, setting `CrawlRate` to 2000 on the web crawler data provider will prevent the crawler from hitting the crawled web server more than once every 2 seconds.

NodesBetweenFlushes Property

Type: `int`

Default: 0

Read Only: `false`

`NodesBetweenFlushes` forces a node set flush whenever that many nodes are crawled. A value of 0 turns off this feature, and flushes will occur at the discretion of the cache manager.

When building an index, it is important to strike a balance between total indexing time, physical index size and real-time availability. The cache manager is tuned to minimize *total* indexing time and index size. This is achieved by performing a minimum number of node set flushes. However, nodes placed within a node set (i.e. a node which has been added to a `NodeStream`) which has not yet been flushed to the index will not be available for searching. The cache manager attempts to use all available RAM to build as large a node set as possible before

committing to a node set flush. On a server with significant available RAM resources, this could equate to several hours of indexing time between node set flushes. If it is desired for crawled nodes to appear in the index faster, setting `NodesBetweenFlushes` to a reasonable low value will cause much more frequent flushes. The consequence of this is a physically larger index on disk and a likely longer total indexing time.

File System Crawler

Type: Daxat.Ess.DataProviders.FileSystemCrawler
Assembly: Daxat.Ess.DataProviders.FileSystemCrawler
File: Daxat.Ess.DataProviders.FileSystemCrawler.dll
Thread safe: `true`

The File System Crawler is used to index files located within a file system. The files must be readable by the Daxat Extensible Enterprise Server service; ensure that the Log On account assigned to the Daxat ESS service has sufficient properties to read the files.

Node Ids

The File System Crawler uses the full path and file name as the node id for each crawled file. If a node already exists with the same node id, the node is replaced.

Node Contents

The full path and file name are written to the node stream as `FileContent`.

CrawlRoot Property

Type: `string` Default: "" Read Only: `false`

The `CrawlRoot` is the full path and directory name where crawling is to being. The File System Crawler will attempt index all files within this directory and all sub directories. As the path will be resolved by the Daxat ESS service, it is important to use drive letters and UNC's reachable from the service process. For example, if the Daxat Extensible Search Server is running on machine `Server1` and the ESS Manager is being run on `Workstation1`, setting a `CrawlRoot` of `C:\Files` will cause the File System Crawler to attempt to index the files located in `C:\Files` on machine `Server1`, not on machine `Workstation1`.

Simple SQL Server Crawler

Type: Daxat.Ess.DataProviders.SimpleSqlServerCrawler
Assembly: Daxat.Ess.DataProviders.SimpleSqlServerCrawler
File: Daxat.Ess.DataProviders.SimpleSqlServerCrawler.dll
Thread safe: `false`

The Simple SQL Server Crawler is used to index data located within Microsoft SQL Server. A supplied query is executed, and each row of the sql result set is indexed as an individual node.

The data must be readable by the Daxat Extensible Enterprise Server service; credentials may be supplied within the connection string, or, if using integrated security, ensure that the Log On account assigned to the Daxat ESS service has sufficient properties to read the data. One way to accomplish this is to grant access to the *machinename*\$ account, where *machinename* is the name of the machine on which the Daxat Extensible Search Server is running, and set the Daxat ESS service Log On account to be the Network Service account.

Node Ids

See the `NodeIdFormatString` property.

Node Contents

Every column of data in the sql result set is written to the node stream as `NamedRegionContent` using the column name as the region name and the string conversion of the field value as `TextContent` contained within the region. Furthermore, metadata is assigned to the node, using the column name as the key and the strongly typed field value as the value.

ConnectionString Property

Type: `string` Default: `null` Read Only: `false`

The SQL Server connection string.

EndOfLastRun Property

Type: `DateTime` Default: `DateTime.Min` Read Only: `true`

The date and time of the completion of the last run of this instance of the data provider.

IdColumns Property

Type: `string[]` Default: `null` Read Only: `false`

An ordered list of column names, the values of which will be concatenated together to create the id for the current node. The id, in turn, is formatted by the `NodeIdFormatString` to create the node id. The property is ignored if the `UseKeyAsId` property is `true`.

For example, if `IdColumns` is set to `{"prodId", "rev"}`, `UseKeyAsId` is `false` and `NodeIdFormatString` is `"{0}"`, then, for each row the corresponding node will have a node id comprised of the value of the `prodId` column concatenated with the value of the `rev` column. If a row has a `prodId` of `"184-K7"` and a `rev` of `"1G"`, then the corresponding node will have a node id of `"184-K71G"`.

NodeIdFormatString Property

Type: `string`

Default: `"{0}"`

Read Only: `false`

The formatting string used to create the node id for the current node. The format item `{0}` corresponds to the id, as generated by either the `IdColumns` or `UseKeyAsId` properties. The format item `{1}` corresponds to the instance name of the data provider.

For example, if the data provider instance name is `"BooksCrawl"`, the id for a particular row is `"0-672-32037-1"` and the `NodeIdFormatString` is set to `"{1}:{0}"`, then the resulting node id will be `"BooksCrawl:0-672-32037-1"`.

SelectStatement Property

Type: `string`

Default: `null`

Read Only: `false`

The SQL query to be used to generate the crawled result set. More precisely, the query text passed to the `System.Data.SqlClient.SqlCommand` constructor. The `SelectStatement` property is treated as a format string. The format item `{0}` corresponds to the `System.DateTime` of the start of the last run of this instance, `{1}` corresponds to the `System.DateTime` of the end of the last run and `{2}` corresponds to the name of the data provider instance.

Using appropriate format specifiers permits the usage of the prior run date and time in the select statement. For example

```
SELECT Foo, Bar FROM Baz WHERE InsertTime >= #{0:d}#
```

StartOfLastRun Property

Type: `DateTime`

Default: `DateTime.Min`

Read Only: `true`

The date and time of the start of the current or last run of this instance of the data provider.

TreatDuplicatesAsUpdates Property

Type: `bool`

Default: `true`

Read Only: `false`

When `true`, new nodes with the same node id as existing nodes replace the existing nodes. When `false`, the new nodes are ignored and not indexed.

UseKeyAsId Property

Type: `bool`

Default: `false`

Read Only: `false`

When `true`, instructs the data provider to use the primary key of the result set as the id for the node, formatted according to the `NodeIdFormatString`.

Web Crawler

Type: Daxat.Ess.DataProviders.WebCrawler
Assembly: Daxat.Ess.DataProviders.WebCrawler
File: Daxat.Ess.DataProviders.WebCrawler.dll
Thread safe: **true**

The web crawler is used to index documents located on web servers. This data provider is actually quite simple, for the bulk of the work – extracting and converting the documents – is performed by node transforms. The web crawler merely keeps track of the urls to crawl, and ensures that urls are not crawled more than once.

Node Ids

The absolute url of the document is used as the node id.

Node Contents

The url of the document is written to the node stream as `UriContent`. Just prior to writing the `UriContent` the meta data key "WebCrawlerBaseUri" is assigned the absolute uri of the document. This key is removed by the web crawler's `OnEndNode` method and does not end up in the index.

CrawlRoots Property

Type: `string[]` Default: `null` Read Only: `false`

A list of fully qualified urls which are used to seed the crawl. Each of these urls will be crawled (if permitted by the web crawler's filtering properties). Any links found on these pages will then become eligible for crawling, and so on.

PermitDomains Property

Type: `string[]` Default: `null` Read Only: `false`

Every url placed into the queue of urls to crawl must pass two simple tests. The first test is that the url must not already exist in the index. The second test is that the fully qualified host name from the url must end with one of the strings contained within the `PermitDomains` array. For example, if `PermitDomains` includes "example.com", then the following urls are all permitted: <http://www.example.com>, <http://www.daxatexample.com> and <http://anotherdaxatexample.com>. If, instead, `PermitDomains` only includes ".example.com", then <http://www.example.com> is valid but <http://example.com> and <http://daxatexample.com> are not.

Node Transforms Reference

How to Use the Included Node Transforms.

The Daxat Extensible Search Server ships with several node transforms, enabling every data provider to handle rich content. There is no common set of properties for the included node transforms; to the contrary, most node transforms do not have any configuration properties.

When a data provider is assigned to an index using the ESS Manager, a default transformation chain is, in turn, associated with the data provider. The default transformation chain is, in order, Uri Reader, OLE Properties, IFilter Document Conversion, Property Tagger and Duplicate Node Detector.

Duplicate Node Detector

Type: Daxat.Ess.Nodes.NodeTransforms.DuplicateNodeDetector
Assembly: Daxat.Ess.Nodes.NodeTransforms.DuplicateNodeDetector
File: Daxat.Ess.Nodes.NodeTransforms.DuplicateNodeDetector.dll

The duplicate node detector attempts to flag nodes which contain the exact same content as any previously indexed node. As each node is indexed, the `TextContent` and `TokenContent` is passed through an MD5 hashing algorithm and the resulting hash value is stored within the node content as the token " `Hhashvalue`" (note the leading space; the quotes are not part of the token). When `OnEndNode` is processed, the computed hash value for the current node is searched within the index. If the same hash value is found, the node is tagged as a duplicate by adding the token " `DUPLICATE`" to the node content.

One use of this information is to eliminate duplicates from search results; augmenting a query with "AND NOT { `DUPLICATE` }" will achieve this result.

Nodes with the same text and token content yet different other forms of content or metadata will still be flagged as duplicates. Moreover, it is possible, although probabilistically unlikely, that two nodes with different text and token content will hash to the same hash value and therefore be flagged as duplicates.

IFilter Document Converter

Type: Daxat.Ess.Nodes.NodeTransforms.IFilterDocumentConverter
Assembly: Daxat.Ess.Nodes.NodeTransforms.IFilterDocumentConverter
File: Daxat.Ess.Nodes.NodeTransforms.IFilterDocumentConverter.dll

IFilters are standardized software components which can convert proprietary file formats into indexable text. IFilters are COM objects which implement the IFilter interface. Windows ships with several IFilters for common Microsoft formats such as Microsoft Office documents. IFilters for other formats, such as PDFs, are available both freely and commercially.

The IFilter Document Converter node transform brings the power of IFilters to the Daxat Extensible Search Engine. For example, installing Adobe's free PDF IFilter makes it possible to index the contents of PDF files. Thanks to the power of the Daxat Extensible Search Server indexing pipeline, these PDFs can be located in any repository for which there exists an data provider. PDFs crawled off of web servers can be indexed as easily as PDFs extracted from the file system.

Some IFilters produce `TextContent` or `ValueContent` which is tagged with document property information, such as “this block of text is the title of the document.” The content is ideally suited for processing by the Property Tagger node transform.

Lower Case Node Transform

Type: Daxat.Ess.Nodes.NodeTransforms.LowerCase
Assembly: Daxat.Ess.Nodes.NodeTransforms.LowerCase
File: Daxat.Ess.Nodes.NodeTransforms.LowerCase.dll

The Lower Case node transform converts all indexed `TextContent` to lower case. Provided that queries are converted to lower case before being executed by ESS, the net result is case insensitive searching with the added benefit of a smaller physical index.

Property Tagger

Type: Daxat.Ess.Nodes.NodeTransforms.PropertyTagger
Assembly: Daxat.Ess.Nodes.NodeTransforms.PropertyTagger
File: Daxat.Ess.Nodes.PropertyTagger.dll

The Property Tagger node transform converts standard Microsoft document properties (sometimes referred to as ActiveX Properties or Office Properties) into indexable content and metadata. Properties include such document facets as title, author, date modified, number of pages, etc. Both `TextContent` and `ValueContent` are capable of carrying property information.

Each property is defined by a property set guid and a property id. For instance, the guid f29f85e0-4ff9-1068-ab91-08002b27b3d9 and id 2 defines the standard DocTitle property. Information on the various properties which can be extracted from a file format is typically found in the file format's developer documentation.

Properties which can be handled by the Daxat Extensible Search Server are defined within the ESS Service.CLRHost.exe.config file inside the PropertyTagger tag. Every property has 7 attributes, as defined below; new properties may be defined by augmenting the configuration file.

Attribute	Description
propid	The guid and key of the property, separated by '/'.
name	The name of the property.
description	A description of the property.
type	The CLS-compatible type of the property.
indexAsContent	When true, the properties string value is indexed as TextContent. If the property has a name, then that TextContent is wrapped in a named region with the same name as the property.
indexAsMetadata	When true, the property has a name and collectAsMetadataArray is false, the value of the property is converted into the strong type defined by the type attribute and is assigned to the metadata key of the same name.
collectAsMetadataArray	When true, the property has a name, and indexAsMetadata is true, the value of the property is converted into the strong type defined by the type attribute and is added to an array of the same type which is assigned to the metadata key of the same name.

Consider, for example, the following property:

```
<Property
  propid="c82bf597-b831-11d0-b733-00aa00a1ebd2/A.HREF"
  name="A.HREF"
  description="Hyperlink"
  type="System.String"
  indexAsContent="false"
  indexAsMetadata="true"
  collectAsMetadataArray="true"
/>
```

This property is generated by the HTML IFilter whenever it finds a hyperlink in an HTML document. By the definition above, the hyperlinks are not indexed as searchable content, but they are added to the node's metadata. Moreover, all of the hyperlinks are added to a string array and assigned to the metadata key `A.HREF`. Armed with this knowledge, the Web Crawler data provider, during `OnEndNode` processing, can simply loop over this array to determine which links were embedded in the current web page. These links, in turn, are queued for further crawling.

Simple Concept Extractor

Type: `Daxat.Ess.Nodes.NodeTransforms.SimpleConceptExtractor`
Assembly: `Daxat.Ess.Nodes.NodeTransforms.SimpleConceptExtractor`
File: `Daxat.Ess.Nodes.NodeTransforms.SimpleConceptExtractor.dll`

The Simple Concept Extractor provides limited support for automatically tagging nodes with highly relevant keywords. These keywords, referred to as the node's *fingerprint*, are stored within the associated metadata under the `fingerprint` key. The fingerprint is an array of strings and represents those words found within the node which are statistically significant. In general, the fingerprint keywords are those words which are common in the node but not common in the index. For example, in an index of food recipes, the node containing the recipe for apple pie will likely contain a disproportionately higher number of instances of the words 'apple' and 'pie', so these words are likely to be included within the node's fingerprint.

One use of a node's fingerprint is to implement 'find more search results like this one' functionality. The fingerprint metadata of each search result can be used as the basis for a new query – a likely candidate is the disjunction of all of the fingerprint terms. For example, if the fingerprint contains the terms 'apple', 'pie' and 'bake', then a valid 'more like this' query is `apple OR pie OR bake`. When executed with an appropriate relevancy scoring algorithm, the expected outcome is that other apple pie recipes will dominate the top of the search results, along with other baked apple dishes and other pies.

In order for the simple concept extractor to compute fingerprints, it needs to calculate, on a term-by-term basis, the frequency of the node's words within the index. There are two ways this can be done – by computing the statistics from the index itself or by using a dictionary file. Due to a limitation in the current physical layout of an index on disk, computing the statistics directly from the index is computationally expensive on the scale required by this node transform – computationally, every word in every node requires a relevancy statistic computation. As such, a dictionary file is the far better choice. A dictionary file contains representative, precomputed word frequency statistics which can be extracted efficiently. The supplied utility `DumpLiteralsAndOccurrences` can generate a dictionary file. This utility must be run in the same directory as an existing index, and the index must not currently be in use. A dictionary file named `literals.dat` will be created. **This process can take an extremely long time to run.**

DictionaryFile Property

Type: `string`

Default: `null`

Read Only: `false`

The full path and file name of a dictionary file representative of the index. If `null`, then statistics will be computed in real-time from the index itself; this will dramatically increase indexing time.

Structured Storage Property Extractor

Type: `Daxat.Ess.Nodes.NodeTransforms.StructuredStoragePropertyExtractor`

Assembly: `Daxat.Ess.Nodes.NodeTransforms.StructuredStoragePropertyExtractor`

File: `Daxat.Ess.Nodes.NodeTransforms.StructuredStoragePropertyExtractor.dll`

The Structured Storage Property Extractor responds to `FileContent`. Some files contain standard properties, sometimes referred to as *Ole properties*. This includes metadata such as summary information, access and modification times, etc.

The extracted properties are written to the node stream as `ValueContent` and, as such, will not be indexed unless processed by another, downstream node transform, such as `PropertyTagger`.

Uri Reader

Type: `Daxat.Ess.Nodes.NodeTransforms.UriReader`

Assembly: `Daxat.Ess.Nodes.NodeTransforms.UriReader`

File: `Daxat.Ess.Nodes.NodeTransforms.UriReader.dll`

The Uri Reader node transform responds to `UriContent` by actively accessing the remote content and adding it to the node stream. Content marked with the `text/plain` MIME type is added to the node stream as `TextContent`. All other content types are written to a temporary file on disk and that temporary file is written to the node stream as `FileContent`. As such, it is necessary to have a downstream `FileContent` handling node transform, such as the `IFilter Document Converter`.

Uri Reader respects `robots.txt` if it exists; it identifies itself to the remote server with the user agent

Mozilla/5.0 (compatible; Daxatbot/1.0; <http://www.daxat.com/bot.html>)

In the event that the remote server indicates that the crawled uri has been moved to a new uri, the new uri will be placed into the current node's metadata under the `UriReader.UriReaderResponseUriGuid` key.

Scorer Reference

How to Use the Included Scorer.

The Daxat Extensible Search Server ships with several scorers which may be used to order search results in meaningful and powerful ways.

Simple Scorer

Type: Daxat.Ess.Scoring.SimpleScorer
 Assembly: Daxat.Ess.Scoring.SimpleScorer
 File: Daxat.Ess.Scoring.SimpleScorer.dll

ScoreObject: `String`

The Simple Scorer uses ScoreObject as a metadata key; the score is set to the associated metadata value. As such, `SearchResult.Metadata[ScoreObject]` must resolved to an `Comparable` type, and all of the values must be comparable to one another.

SimpleRelevancyScorer

Type: Daxat.Ess.Scoring.Relevancy.SimpleRelevancyScorer
 Assembly: Daxat.Ess.Scoring.Relevancy.SimpleRelevancyScorer
 File: Daxat.Ess.Scoring.Relevancy.SimpleRelevancyScorer.dll

ScoreObject: `Daxat.Ess.Scoring.Relevancy.RelevancyCriteria`

The Simple Relevancy Scorer is similar to the standard TF*IDF relevancy algorithm, with a few select differences. Whereas TF*IDF increases the term score for every occurrence in a node of a given term t , this algorithm does not permit any one term to contribute indefinitely. Instead, terms are subjected to a geometric series which scales their weight by

$$\sum \frac{1}{2^{tf_{ij}}}$$

where tf_{ij} is the number of occurrences of term i in node j . In the limit the value of series approaches 2, so no one term can contribute more than twice its weight to the final relevancy score.

Additionally, the inverse of the total distance separating the relevancy terms in the node is added to the score. For example, consider two nodes, both containing one instance each of the words 'apple' and 'pie'. In the first node, let the words be next to one another. In the second node, let the words be extremely far apart, with 'apple' occurring near the beginning of the node and 'pie' occurring near the end of the node. All other things being equal, when scored against the relevancy terms 'apple' and 'pie', the first node will score higher because the words are located closer together.

IDF is also computed differently by this algorithm. Whereas traditional IDF is based upon the number of nodes in the index, this algorithm computes IDF based upon the number of word positions in the index.

RelevancyCriteria Support

Property	Supported?
Terms	Yes
Regions	Yes
IncludeTermsFromQuery	No (Always treated as <code>true</code>)
NormalizeScores	Yes

AdvancedRelevancyScorer

Type: Daxat.Ess.Scoring.Relevancy.AdvancedRelevancyScorer
 Assembly: Daxat.Ess.Scoring.Relevancy.AdvancedRelevancyScorer
 File: Daxat.Ess.Scoring.Relevancy.AdvancedRelevancyScorer.dll

ScoreObject: Daxat.Ess.Scoring.Relevancy.RelevancyCriteria

The Advanced Relevancy Scorer computes relevancy in the vein of the Okapi BM25 algorithm of Robertson and Walker, with additional support for Daxat Extensible Enterprise Search features such as weighted named regions.

RelevancyCriteria Support

Property	Supported?
Terms	Yes
Regions	Yes
IncludeTermsFromQuery	No (Always treated as true)
NormalizeScores	Yes

Troubleshooting

Solutions to Common Problems.

One of the design goals of the Daxat Extensible Search Server is to create a search engine which simply works and works simply. However, even the best systems can fail. In particular, ESS inherits several nuances of the Microsoft .NET framework. This appendix addresses the most common problematic situations and provides workarounds or advice on how to deal with these issues.

Cannot Connect To Server

Clients connect to the Daxat Extensible Search Server via standard .NET remoting. As such, the communication *channel* between the client and the server must be clear. By default, the Daxat Extensible Search Server communicates over the TCP channel, port 20869. If a firewall or other device is blocking this traffic between the client and the server, communication will fail.

The .NET remoting configuration of the server is defined within the ESS Service.CLRHost.exe.config file, located within the application directory (typically C:\Program Files\Daxat\Extensible Search Server 2005). The server uses standard .NET remoting configuration (found within the <system.runtime.remoting> section of the configuration file). If it is necessary to change the remoting configuration, it is extremely important to stop the Daxat ESS service first. Once the service is stopped, the configuration file may be edited. Upon restart, the server will bind to the new channel.

If the server is bound to more than one IP address, it may be necessary to bind the ESS to one of the available IP addresses. To do so, augment the <channel> element with a `bindTo` element. For example, to bind the server to IP address 10.47.121.31, change the <channel> element to `<channel ref="tcp" port="20869" bindTo="10.47.121.31">`.

Too Much/Too Little Exception Data at Client

When a client calls one of methods hosted on the server and that method throws an exception, the amount of exception data returned to the client is partially controlled by the .NET Framework. The standard configuration element <customErrors> determines if exceptions are filtered before being passed onto the client.

For example, if, during development, no meaningful exception information is reaching the client, change the mode of the <customErrors> element to "off".

IFilter Document Converter Not Converting Documents

In order for the IFilter Document Converter to work, the appropriate IFilters must be installed and configured. This node transform uses both the registry's MIME database and the registry's file extension mappings to determine which IFilter to use to convert a particular document.

A standard installation of Microsoft Windows normally includes IFilters for standard Microsoft document types. If Microsoft Office documents are not being converted, it may be necessary to reload the Microsoft Office IFilters. This can be accomplished by installing the Index Server Windows component. Also, Microsoft occasionally releases updated Office IFilters on their web site.

In order to convert PDF documents, a PDF IFilter is required. Adobe has a freely available PDF IFilter on their web site.